MSc in Computer Science 2020-21

Project Dissertation

Project Dissertation title: Deep learning for tabular data in healthcare

Term and year of submission: Trinity Term 2021

Candidate Number: 1047662

Word count (excluding contents, figures, tables, bibliography): 28907

Additional pages (contents, figures, tables, bibliography): 28 pages

Acknowledgements

I wish to express my gratitude to Professor David Clifton and the AI in Healthcare Group in the Institute of Biomedical Engineering, for giving me the chance to join this important research programme with real-world significance to the current Covid-19 pandemic. I am particularly grateful to the CURIAL research team - Andrew Soltan, Yang Yang, Omid Rohanian and Jenny Yang – for our weekly meetings, their support in data access and advice on aims and methods of this project.

I would also like to thank Dr Ani Calinescu, for her feedback, advice and support on this project and dissertation. Her insightful questions and comments have been invaluable in shaping my approach to this dissertation.

Finally, I would like to thank my family, my parents and my sister, for their constant support over the course of this year.

Abstract

Background: Machine learning applications on tabular healthcare data have significant untapped potential to diagnose diseases early and personalise treatment for patients but tabular healthcare data presents a number of challenges, including heterogenous feature types with varying distributions, extreme class imbalance and missing data. These opportunities and challenges are exemplified by Covid-19 pandemic datasets, which offer the potential to rapidly screen for the disease in hospitals. Synthetic tabular data generation, using conditional tabular generative adversarial networks (CTGAN) and tabular variational autoencoders (TVAE), is a solution to some of these challenges. However, their robustness to imbalanced and missing data has not been systematically investigated. Moreover, in recent years, several extensions to variational autoencoders have been proposed improving their performance on imaging data, but these have yet to be investigated on tabular data in combination with TVAE. In machine learning applications for predictive tasks on tabular data, tree ensemble methods still dominate but deep learning is desirable, as it can facilitate end-to-end multi-modal pipelines and online learning; this is of particular interest in the healthcare industry. Recent work has highlighted promising model families but there has been no comprehensive comparison of them or evaluation of their robustness to imbalanced and missing data. Moreover, a new school of thought has emerged, which focusses on regularisation. The combination of specialised architectures with multiple classes of deep learning regularisation techniques could improve the performance of the former, but no work has investigated this.

Aims, objectives and contributions: The main contribution of Part 1 of this study is the novel application of an extension to variational autoencoders – normalising flows – to the new domain

of tabular data, and investigation of whether this improves the quality of generated synthetic electronic health record data. The aim is to advance the current state-of-the-art performance in synthetic tabular data generation. The main contribution of Part 2 of this study is the novel application of regularisation cocktails to 14 specialised deep learning architectures for tabular data, and investigation of whether this improves performance in predicting Covid-19 status on electronic health record data. The aim is to advance the current state-of-the-art performance of deep learning models for predictive tasks on tabular data. This study additionally performs a systematic evaluation of the robustness of generative and predictive models to imbalanced and missing data. The aim is to elucidate the utility of these models on imperfect real-world datasets.

Methods: Data of hospital admissions during the Covid-19 pandemic was collected from electronic health records of 4 NHS trusts. Sample sizes ranged from 1800 to 217200 patients. CTGAN, TVAE and TVAE with planar, Sylvester, non-linear independent component estimation and real-valued non-volume preserving flow were trained to generate synthetic Covid-19 hospital admission data. Quality of synthetic data was evaluated on statistical, machine learning detection and efficacy metrics. 14 state-of-the-art deep learning models from differentiable tree, attention, feature interaction and regularisation model families, with and without regularisation cocktails (consisting of weight decay, dropout, snapshot ensembling, batch normalisation, stochastic weight averaging and Lookhead optimiser), were trained to predict Covid-19 status from demographic, vital sign and blood test data at hospital admission. Classification performance was quantified with area under the ROC curve.

Results: All flow types, except for planar flow, improved the generative performance of TVAE, increasing statistical metrics by 1 – 3%, machine learning detection by 1 – 7%, and machine

learning efficacy by 1 – 2%. Statistical and machine learning detection metrics were maintained when CTGAN and TVAE, including with normalising flows, were trained on imbalanced and missing data. Best predictive performances for Covid-19 status were achieved by feature interaction-based models (AUC > 0.8), in particular Wide and Deep and product-based neural networks (AUC 0.84 – 0.90), and self-attention and intersample attention transformer (AUC 0.83 – 0.88). Regularisation cocktails improved performance of almost all models: AUC increased by 1-4% for differentiable tree-based, 1-10% for attention-based and 3-4% for feature interaction-based models. Imbalanced training data degraded AUC of most models, but performance improvement associated with regularisation cocktails was maintained and even augmented.

Conclusions: TVAE with normalising flows improves the state-of-the-art performance of deep generative models on tabular data. These models are robust to real-world datasets with imbalanced and missing data, highlighting their potential for the healthcare sector. Specialised deep learning models can match and surpass the performance of gradient boosted decision trees in predictive tasks on tabular data, enabling the combination of high performance, multi-modal pipelines and online learning, which is key in approximating clinician-like decision-making processes in the healthcare sector. Regularisation cocktails contribute to high performance for all families of specialised architectures, advancing state-of-the-art performance of deep learning predictive models on tabular data, and aids performance in the setting of imbalanced training data.

Abbreviations

AUC: area under ROC curve

BCE: binary cross entropy

BN: batch normalisation

CIN: compressed interaction network

CS: Chi-squared test

CTGAN: conditional tabular generative adversarial network

DCN: deep and cross network

DeepFM: deep factorisation machine

DNDT: deep neural decision tree

EHR: electronic health record

ELBO: evidence lower bound

FM: factorisation machine

GAN: generative adversarial network

GBDT: gradient boosted decision tree

KL: Kullback-Leibler divergence

KS: Kolmogorov–Smirnov test

MLP: multi-layer perceptron

MLR: muddling labels for regularisation

NICE: non-linear independent component estimation

NODE: neural oblivious decision ensemble

ODT: oblivious decision tree

PNN: product-based neural network

RealNVP: real-valued non-volume preserving transformation

RLN: regularisation learning network

SAINT: self-attention and intersample attention transformer

SE: snapshot ensembles

SELU: scaled exponential linear unit

SGD: stochastic gradient descent

SNN: self-normalising neural network

SVC: support vector machine classifier

SWA: stochastic weight averaging

TGAN: tabular generative adversarial network

TVAE: tabular variational autoencoder

VAE: variational autoencoder

WGAN: Wasserstein generative adversarial network

xDeepFM: extreme deep factorisation machine

Contents

Chapter 1: Introduction **10**

1.1 Motivation 10

1.2 Challenges 10

1.3 Motivation and Challenges: The Case Study of Covid-19 12

1.4 Deep learning-based synthetic tabular data generation 13

1.4.1 Existing work 13

1.4.2 Aims, objectives and contributions 15

1.5 Deep learning-based prediction on tabular data 16

1.5.1 Existing work 16

1.5.1 Aims, objectives and contributions 19

1.6 Structure of dissertation 21

Chapter 2: Deep generative models for synthetic tabular data **22**

2.1 Synthetic data 22

2.2 Generative adversarial networks 23

2.2.1 Conditional tabular GAN 24

2.3 Variational autoencoders 26

2.3.1 Tabular VAE 29

2.4 Normalising flows 30

2.4.1 Planar flows 33

2.4.2 Sylvester flows 33

2.4.3 Non-linear Independent Component Estimation 35

2.4.4 Real-Valued Non-Volume Preserving transformation 36

2.5 Summary 37

Chapter 3: Deep learning models for predictive tasks on tabular data 38

3.1 Differentiable tree-based models 38

3.1.1 Neural oblivious decision ensembles 38

3.1.2 Quantum Forest 41

3.1.3 Deep neural decision trees 42

3.2 Attention-based models 44

3.2.1 TabNet 44

3.2.2 TabTransformer 47

3.2.3 Self-attention and intersample attention transformer 49

3.3 Feature interaction-based models 51

3.3.1 Wide and Deep network 54

3.3.2 Deep Factorisation Machine 55

3.3.3 Deep and Cross network 57

3.3.4 Extreme Deep Factorisation Machine 58

3.3.5 Product-based neural network 60

3.4 Regularisation-based models 62

3.4.1 Deep learning regularisation techniques 62

3.4.2 Regularisation-based architectures 70

3.5 Summary 76

Chapter 4: Experimental Methods 80

4.1 Data 80

4.2 Part 1: Synthetic tabular data generation 82

4.2.1 Models 82

4.2.2 Training 85

4.2.3 Metrics 86

4.3 Part 2: Deep learning for prediction on tabular data 87
4.3.1 Models 87
4.3.2 Training 92
4.3.3 Metrics 94
4.4 Summary 94

Chapter 5: Results 95
5.1 Part 1: Synthetic tabular data generation 95
5.2 Part 2: Deep learning for prediction on tabular data 99
5.3 Summary 107

Chapter 6: Discussion 108
6.1 Part 1: Synthetic tabular data generation 109
6.2 Part 2: Deep learning for prediction on tabular data 111
6.3 Limitations 119

Chapter 7: Conclusion 121
7.1 Contributions 121
7.2 Future work 122

Appendices 123
A Hyperparameter search 124
B Source code 131

Bibliography 132

Chapter 1: Introduction

1.1 Motivation

Tabular datasets typically consist of independent and identically distributed samples (rows) represented as a vector of features (columns). They are one of the most ubiquitous data types and are relied on in a number of real-world domains, of which healthcare is one. Most significantly, electronic health records (EHR) are underpinned by storage in a tabular form. While machine learning has been increasingly applied to healthcare imaging datasets, EHR data represent a vast and relatively untapped source of healthcare data. Machine learning applications on EHR data have the potential to screen and diagnose diseases early, monitor deterioration, stratify patients to personalised treatment strategies, monitor treatment response and provide real-time decision support to healthcare professionals, among many other applications (Wong et al., 2018; Zheng et al., 2017; Bronsert et al., 2020; Kogan et al., 2020; Luz et al., 2020; Wong et al., 2018; Golas et al., 2018; Mandair et al., 2020; Martinez et al., 2020). Thus, optimising machine learning applications for tabular data is paramount to maximising its potential to revolutionise the healthcare domain.

1.2 Challenges

However, tabular data presents a number of unique challenges which have traditionally limited the success of deep learning approaches that have performed well on imaging and natural language. Firstly, tabular data typically consists of heterogenous feature types (continuous, ordinal and categorical) with a wide variety of distributions, in contrast to imaging and natural language where datapoints have uniformly continuous and discrete distributions, respectively. In addition, continuous features often follow multi-modal non-Gaussian distributions, and distributions of discrete features are often significantly imbalanced between major and minor

categories (Xu et al., 2019). Secondly, there is no prior knowledge from the data structure e.g. positional information which could be exploited to infer associations (as individual rows and columns are arbitrarily ordered); indeed, in a table, any set of features might be either independent or correlated so models must be able to discover correlations without relying on locality, in contrast to imaging (where pixels in close proximity can be inferred to be correlated) or natural language (where tokens in close proximity have association). Thirdly, compared to imaging, the relative importance of different features in tabular data is considerably more variable.

Further challenges are associated with machine learning on tabular data specifically in the healthcare domain. Firstly, diseases and conditions of interest in predictive tasks are typically uncommon in the population, thus healthcare datasets are characterised by significant class imbalance, often more extreme than 1:10. This is especially the case in application domains of disease screening and early diagnosis. This can result in lack of training opportunities for minor classes. A second key challenge is missing data and data sparsity. When data is collected routinely in EHRs, many fields are often missing per patient, due to targeted clinical investigation, lack of time and resource, patient refusal, etc. When data is collected prospectively, non-participation and dropout rates can be high. Thirdly, access to healthcare datasets is typically limited by data privacy regulation owing to data being personally sensitive. As data cannot be freely shared or released into the public domain, this makes model development across multiple datasets and external verification of model performance difficult. Finally, machine learning applications in the healthcare domain require interpretability, as it is necessary for healthcare professionals to understand how a prediction was reached to verify the accuracy of its reasoning and to gain insights into the most important contributing factors, given the critical nature of clinical decisions and their safety implications. Thus, the “black box”

nature of deep learning methods, where feature importance is often difficult to assess given high-dimensional and implicit representations, poses a barrier to their wider use.

1.3 Motivation and Challenges: The Case Study of Covid-19

The importance of tabular data and the challenges associated with it has been exemplified by datasets collected for Covid-19. Rapid screening of Covid-19 in patients admitted to hospital is crucial for robust infection control among hospital patients but is hindered by slow turnaround of gold-standard Covid-19 PCR tests and lack of access to mass high-fidelity rapid Covid-19 tests – many rapid testing options have insufficient sensitivity in real-world settings (Dinnes et al., 2021) and cannot be performed at scale. Vital signs and blood tests conducted routinely on patients admitted to hospital typically have a much faster turnaround than Covid-19 tests, so could be leveraged for earlier identification of patients who might be Covid-19 positive, within the first hour of admission. Studies have demonstrated that specific blood-based markers, for example lymphocytes, basophils, eosinophils, C-reactive protein, ferritin, troponin and D-dimer are perturbed in Covid-19 infection, and thus could be markers that indicate viral presence (Wynants et al., 2020; Petrilli et al., 2020). Applying machine learning to this tabular EHR data in order to predict Covid-19 status represents a promising approach to rapid triage of Covid-19 in hospitals, at no additional cost.

However, this task faces many of the challenges that beset machine learning on tabular healthcare data. The data will consist of a mix of discrete features, such as gender and ethnicity, and continuous features, such as vital signs and blood test parameters. Discrete features such as ethnicity have highly imbalanced distributions with a “White” majority class of more than 90%. One would expect relative feature importance to be highly variable, given that Covid-19

stimulates a primarily immune response which is reflected by blood-based markers (tests such as blood cell counts will be more important than tests for kidney or liver function). A chief issue will be class imbalance, as Covid-19 prevalence is low among hospital admissions. EHR datasets used in this study encompassing the entire Covid-19 pandemic records an average prevalence of 5 – 10%, and this is likely to decrease substantially as the population becomes fully vaccinated. In a "post" pandemic world, it is highly likely that prevalence will persist at below 1% (Telenti et al., 2021).

1.4 Deep learning-based synthetic tabular data generation

1.4.1 Existing work

For several challenges associated with machine learning on tabular data in the healthcare domain, generating synthetic datasets that closely resemble real datasets is a potential solution. Synthetic data generation can be used to (1) increase the minority class in a training dataset, as a form of data augmentation, and thus ameliorate issues of class imbalance (Che et al., 2017), (2) learn the distribution of the data to impute missing data (Yoon et al., 2018; Hammad Alharbi & Kimura, 2020; Dong et al., 2021; Camino et al., 2019; Pereira et al., 2020), and (3) fully anonymise a sensitive dataset, enabling it to be more widely disseminated for collaborative use while mitigating the risk of privacy violations (Yoon et al., 2020; Bae et al., 2019).

Traditionally, synthetic data generation has focused on taking features as random variables and modelling multivariate probability distributions, followed by sampling from these joint distributions. Discrete variables can be modelled by Bayesian networks (Avino et al., 2018; Zhang et al., 2017) while continuous variables can be modelled by copulas (Patki et al., 2016; Sun et al., 2019). However, the fidelity of synthetic data generated in this manner is hindered by

limitations in complexity of multivariate distributions imposed by computational time and memory constraints.

In recent years, significant advances have been made in deep learning-based generative models, primarily generative adversarial networks (GANs) and variational autoencoders (VAEs), which learn probability distributions more flexibly and thus can generate higher quality synthetic samples than statistical models (Goodfellow et al., 2014; Kingma & Welling, 2014). GANs and VAEs have a history of success in the imaging domain but have been extended to tabular data including in the healthcare sector for the purpose of generating EHR data. Two prominent examples are medGAN, which combines a GAN and autoencoder, to generate data with high-dimensional heterogenous feature types (Choi et al., 2017) and ehrGAN, which was used for augmentation of training data (Che et al., 2017). However, the challenges associated with tabular data already mentioned poses several significant problems for the modelling of its probability distribution. The first is the need to model continuous and discrete feature distributions; for GANs this requires using different non-linear activations on the output (Xu et al., 2019). Secondly, multi-modal non-Gaussian distribution of continuous features can lead to mode collapse and vanishing gradient, and imbalanced discrete features can result in minor classes, which exert little effect on the overall distribution of the data, being missed in generated samples (Xu et al., 2019). Thirdly, if individual datapoints in real data is in the form of sparse one hot encoded vectors while synthetic datapoints are dense vectors representing probability distribution over classes, these may be trivially distinguished based on sparseness (Xu et al., 2019).

Recent state-of-the-art GAN models such as conditional tabular GAN (CTGAN) have been proposed to address these challenges. Tabular VAEs (TVAE) have simultaneously been

proposed and demonstrated to outperform CTGAN (Xu et al., 2019). While CTGAN possesses a number of extensions relative to, not only the original GAN, but also later developments in the form of Wasserstein GAN (WGAN) and tabular GAN (TGAN) (Arjovsky et al., 2017; Xu & Veeramachaneni, 2018), which makes it well-suited for the idiosyncrasies of tabular data, TVAE is based on the original VAE and has not been extended. In particular, as will be outlined in Chapter 2, while the generative network of TVAE has been adapted for tabular feature types, its inference network is still equivalent to that of the original VAE. Nevertheless, in recent years, a number of extensions have been proposed to VAEs which have systematically improved their performance on imaging datasets (Rezende & Mohamed, 2015; Higgins et al., 2017; Rainforth et al., 2018), but these have yet to be investigated on tabular datasets in combination with TVAE.

Another remaining gap in the evaluation of CTGAN and TVAE is their robustness to datasets with imbalanced and missing data, which has not been systematically investigated, despite this being one of the main challenges in real-world applications in industries such as healthcare.

1.4.2 Aims, objectives and contributions

Thus, in Part 1 of this study, the main contribution is the novel application of an extension to VAEs – normalising flows – to the new domain of tabular data. Normalising flows is a technique to learn a richer approximate posterior and thereby address one of the key, if not the most important, limitation of VAEs (Rezende & Mohamed, 2015). I create a new solution for synthetic tabular data generation by extending the vanilla TVAE inference network to make use of four different types of flows – planar, Sylvester, non-linear independent component estimation (NICE) and real-valued non-volume preserving transformation (RealNVP) (Rezende

& Mohamed, 2015; van den Berg et al., 2018; Dinh et al., 2014; 2017). I assess whether this approach improves generative performance by empirically investigating whether they improve the quality of synthetic data generated from real EHR datasets of patients admitted to hospital during the Covid-19 pandemic, in comparison to vanilla TVAE and CTGAN. To the best of my knowledge, this is the first work to apply normalising flows to VAEs on tabular data. The aim of this extension is to advance the current state-of-the-art performance in synthetic tabular data generation.

The second contribution of Part 1 of this study is the novel systematic application of CTGAN and TVAE to imbalanced and missing data. I investigate the performance of CTGAN, vanilla TVAE and TVAE with normalising flow on real EHR datasets with imbalanced and missing data, to evaluate their robustness. The aim of this is to comprehensively elucidate the utility and potential of these generative models on imperfect real-world datasets and the healthcare sector.

1.5 Deep learning-based prediction on tabular data

1.5.1 Existing work

For supervised predictive tasks on tabular data, shallow learning in the form of tree-based ensemble methods e.g. random forest and gradient boosted decision trees (GBDT) are the current dominant and most widely used approaches; leading methods in this space include XGBoost, CatBoost and LightGBM (Friedman, 2001; Chen & Guestrin, 2016; Prokhorenkova et al., 2017; Ke et al., 2017). In addition to state-of-the-art performance, other benefits which make them a preferred technique are their efficiency in representing decision boundaries common in tabular data, selection of features of high importance, and explainability conferred by tracking tree and node structure (Lundberg et al., 2018). Practically, they are also easy to

implement and fast to train, without requiring significant hyperparameter tuning, and thus are popular in real-world applications. However, tree-based methods have drawbacks. They learn by greedy splitting to construct trees (local optimisation) so may not learn globally optimal solutions. They are not differentiable so cannot be trained with gradient optimisation, which prevents their inclusion in pipelines that are trained end-to-end, and they are not suitable for online learning. Moreover, they require manual feature engineering.

In recent years, deep learning has enjoyed remarkable success in predictive tasks in a number of domains, including computer vision, audio and natural language processing, and has become a state-of-the-art method that has enabled significant advances in these (He et al., 2016; Devlin et al., 2018; Lai et al., 2015; van den Oord et al., 2016; Amodei et al., 2016). This has owed in large part to specialised architectures, such as CNNs and RNNs, which are well adapted to encode data into meaningful representations in these domains. There has been significant interest in the machine learning community to extend deep learning to tabular data as it can sidestep many of the challenges faced by decision trees. Deep learning models can be incorporated into end-to-end pipelines trained with gradient-based optimisation methods, from which arises numerous benefits such as construction of multi-modal pipelines that leverage imaging and text data alongside tabular data, combining models most suited to each data type. Such pipelines hold promise in healthcare as a way of integrating the insights from patient symptom reports, biochemical and imaging investigations, to make more realistic predictions about diseases, in a manner that more closely replicates the real decision-making processes of clinicians. Deep learning methods can also be continually trained on streaming data, which is needed for EHR data that is collected in real time. This is especially important in the Covid-19 pandemic, which is rapidly evolving as the population becomes vaccinated; models for predictive tasks on hospital Covid-19 admissions will need to be continuously refined. Other

advantages of deep learning are automatic feature engineering and greater suitability for large datasets (Goodfellow et al., 2016; Hestness et al., 2017).

Theoretically, the representative power of deep learning methods confers them potential to outperform tree-based approaches. However, the success of deep learning methods in imaging and natural language domains have not extended thus far to tabular data, many of the reasons of which are the challenges associated with tabular data already mentioned. Deep learning models do not demonstrate consistent performance advantage over leading GBDT methods; it has been found to be dataset-dependent (Zhou & Feng, 2017; Miller et al., 2017; Lay et al., 2018; Feng & Zhou, 2018; Ke et al., 2018). Certainly, there is no dominant deep learning architecture for tabular data. However, the domain remains underexplored and the increasing number of works in this area have highlighted a few promising architectures which improve the performance of deep learning on tabular data. The main families of current state-of-the-art deep learning models for predictive tasks on tabular data, as identified in a recent review by Gorishniy et al. (2021), are: differentiable tree-based, attention-based and feature interaction-based models. However, the performance of these has been evaluated with different datasets; there has been no comprehensive comparison of all model families on one task. Due to this, it is difficult to pinpoint which models or families perform better than others. Moreover, none of the state-of-the-art models have been evaluated in the healthcare domain, despite its importance and the abundance of tabular datasets in this domain.

A new school of thought has recently emerged which focusses on regularisation, with simple but well-regularised deep learning models shown to outperform specialised architectures (Kadra et al., 2021; Shavitt & Segal., 2018; Lounici et al., 2021; Klambauer et al., 2017). It has been proposed that regularisation cocktails (the joint and simultaneous application of multiple classes

of deep learning regularisation techniques) could also benefit specialised architectures (Kadra et al., 2021). However, no work has systematically investigated the combination of specialised architectures with a range of regularisation techniques.

Another remaining gap in the evaluation of state-of-the-art deep learning models for predictive tasks on tabular data is that no work has systematically examined their robustness, in particular to datasets with imbalanced classes or missing data, which are common in the healthcare domain. Moreover, it has been suggested regularisation cocktails need to be tested under all data modalities, including imbalanced and missing data (Kadra et al., 2021).

1.5.1 Aims, objectives and contributions

To this end, Part 2 of this study performs a large-scale evaluation of all 14 models on a real EHR dataset of patients admitted to hospital during the Covid-19 pandemic. Performance is externally validated on 3 additional Covid-19 datasets from different NHS trusts. To the best of my knowledge, this is the widest systematic comparative study to date of deep learning models for predictive tasks on tabular data, surpassing the 4-5 compared in two recent reviews (Gorishniy et al., 2021; Shwartz-Ziv & Armon, 2021). It also represents the first application of these models to the healthcare domain. The aim of this extensive comparison is to identify the models that consistently perform well and make recommendations on those that should be preferred for tabular data. I also aim to definitively answer the question of whether any of these models surpass the performance of GDBT (which have been previously applied by others in the research group) on this typical tabular healthcare dataset, which has high applicability to other EHR data.

The main combination of Part 2 of this study is the novel application of regularisation cocktails to specialised deep learning models for tabular data, a new but naturally coherent approach to improving the state-of-the-art performance of these models. I extend the architecture and training of 14 current state-of-the-art deep learning models for prediction on tabular data (Popov et al., 2019; Chen, 2020; Yang et al., 2018; Arik & Pfister, 2019; Huang et al., 2020; Somepalli et al., 2021; Cheng et al., 2016; Guo et al., 2017; Wang et al., 2017; Lian et al., 2018; Qu et al., 2016; Shavitt & Segal, 2018; Lounici et al., 2021; Klambauer et al., 2017) with combinations of three families of regularisation techniques – weight decay, model averaging (dropout and snapshot ensembles) and implicit regularisation (batch normalisation, stochastic weight averaging and Lookahead optimiser) (Kadra et al., 2021; Srivastava et al., 2014; Huang et al., 2017; Ioffe & Szegedy, 2015, Izmailov et al., 2018; Zhang et al., 2019). I empirically investigate whether performance in the task of predicting Covid-19 status from EHR datasets of patients admitted to hospital during the Covid-19 pandemic is improved, relative to the base deep learning models. To the best of my knowledge, this is the first work to combine regularisation cocktails and specialised deep learning models for tabular data. The aim of this extension is to advance the current state-of-the-art performance of deep learning models for predictive tasks on tabular data.

The second contribution of Part 2 of this study is the novel systematic application of all 14 state-of-the-art deep learning models, with and without regularisation, to imbalanced and missing data. I investigate the performance of models on real EHR datasets with imbalanced and missing data, to evaluate their robustness. The aim of this is to comprehensively elucidate the utility and potential of these predictive models on imperfect real-world datasets and the healthcare sector.

1.6 Structure of dissertation

This dissertation will start by reviewing current state-of-the-art deep generative models for synthetic tabular data in Chapter 2, followed by a review of state-of-the-art deep learning models for predictive tasks on tabular data, in Chapter 3. The experimental work is presented in detail in Chapter 4 and the results of these empirical investigations are given in Chapter 5. Chapter 6 presents a discussion of the findings and Chapter 7 concludes with suggestions of further work.

Chapter 2: Deep generative models for synthetic tabular data

This chapter outlines the requirements for synthetic tabular data in Section 2.1, reviews the current state-of-the-art deep learning models developed to generate synthetic tabular data in Sections 2.2 and 2.3 and presents the approach of normalising flows in Section 2.4.

2.1 Synthetic data

The requirements of synthetic tabular data are that it must closely match the real data in feature distributions and preserve the associations between features, whilst ensuring no information leakage from real data. Quality of synthetic data generated from real data is usually evaluated in four ways: statistical, machine learning detection, machine learning efficacy and privacy (SDV, n.d.).

1. Statistical evaluation compares corresponding individual columns in the real and synthetic data, for example the distributions of continuous and discrete features.
2. Machine learning detection evaluates the difficulty of the task of training a machine learning classifier to separate real from synthetic data (which are given different binary labels).
3. Machine learning efficacy assesses whether it is possible to substitute the real data for the synthetic data in a machine learning task, using the synthetic data as the training data for a predictive task and measuring the performance of the trained model on a test dataset of real data.

4. Privacy evaluation assesses whether it is possible to predict sensitive features in the real data, after seeing the synthetic data, by training an adversarial model to predict sensitive features in the synthetic data and measuring its performance on the real data.

2.2 Generative adversarial networks

Deep generative models have traditionally been limited by the challenges associated with estimating intractable probability distributions present in data. GANs present a new approach for generating data using an adversarial process (Goodfellow et al., 2014). The core idea is to train two models, a generative model G to learn the distribution of the data (it does this by sampling from a prior $p_z(\mathbf{z})$ and learning a transformation to map this to the real data distribution with $G(\mathbf{z})$ representing the mapping) and a discriminative model D that distinguishes whether the datapoints are real or generated with $D(\mathbf{x})$ representing the probability that $\mathbf{x}$ originated from the real rather than generated data. The two models compete in a minimax two player game, with G being trained to generate data samples which "fool" the discriminator (maximising the error of D) and D being trained to make the correct prediction with regard to the origin of real and generated data samples (minimising the error of D). D provides training signal to G to generate data samples in regions that are more likely to be classed as real data and which are therefore less distinguishable from real data. Eventually, a unique solution is reached in which G generates data that replicates exactly the distribution of the real data and D classifies datapoints with 50% accuracy. This avoids the traditional difficulties in inference, obviating the need for approximate inference networks or Monte Carlo Markov chain sampling. G and D are typically multi-layer perceptrons (MLPs) so can be trained with backpropagation, with sampling by forward propagation.

The training process and objective of GANs is as follows:

1. Train D to maximise the probability of classifying the real data $\mathbf{x}$ and the generated data from G correctly
2. Train G to minimise $\log\left(1 - D\left(G(\mathbf{z})\right)\right)$
3. Objective function:

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

In practice, the algorithm alternates between k steps which train D followed at the end by one step which trains G.

2.2.1 Conditional tabular GAN

CTGAN modifies the base GAN model to address the challenges posed by tabular data, and is the current state-of-the-art GAN for the tabular domain, performing significantly better than other GANs and Bayesian methods (Xu et al., 2019). It extends GAN with mode-specific normalisation, which converts continuous values into a bounded vector representation, to address the challenge of multi-modal non-Gaussian continuous feature distributions. This is in contrast to min-max normalisation traditionally used by GANs. CTGAN incorporates a conditional generator and training-by-sampling to address the challenge of imbalanced discrete features: the conditional generator has the role of generating synthetic datapoints conditioned on a discrete feature and training-by-sampling samples synthetic and real data by log frequency of each class, thereby ensuring all classes in a discrete feature are sampled evenly during training. However, CTGAN also ensures the real data distribution is recovered. This is in contrast to a traditional GAN which does not have any mechanism to ameliorate imbalanced discrete features: in these cases, random sampling of real data results in insufficient representation of minor classes and thus generated data that lacks certain classes, but resampling of real data

results in a learnt distribution for the generated data that is different from the real distribution (Xu et al., 2019).

Mode-specific normalisation involves representing values of a continuous feature as a vector of a mode and mode value. The distribution of individual continuous features are first modelled as a variational Gaussian mixture model: for example, for a three mode distribution, the value of the modes can be represented as η_1, η_2 and η_3 and the Gaussian mixture model as $\mathbb{P}_{C_i}(c_{i,j}) = \sum_{k=1}^{3} \mu_k \mathcal{N}(c_{i,j}; \eta_k, \phi_k)$ where C_i is the continuous feature, $c_{i,j}$ is the feature value, μ_k is the weight of the mode and ϕ_k is the standard deviation of the mode. Using this fitted model, the probability of the value of a feature arising from each mode is computed as probability densities $\rho_k = \mu_k \mathcal{N}(c_{i,j}; \eta_k, \phi_k)$. The mode with the highest probability density is selected, thus informing the one hot vector of the mode: for example $\beta_{i,j} = [0, 0, 1]$ if the third mode was selected. The value of the feature within the mode is computed by normalisation of the value of the feature relative to the value and standard deviation of the mode: $\alpha_{i,j} = \frac{c_{i,j} - \eta_3}{4\phi_3}$. The continuous feature can then be represented as $\alpha_{i,j} \oplus \beta_{i,j}$, where $\oplus$ is a concatenation operator.

The core idea of the conditional generator is that it fixes the value of a particular discrete feature i.e. k^* is the value of the discrete feature D_{i^*}. The generator seeks to learn generated data with a conditional distribution i.e. $\mathbb{P}_G(\text{row}|D_{i*} = k^*)$ that matches the conditional distribution $\mathbb{P}(\text{row}|D_{i*} = k^*)$ in the real data. In this way, the correct row distribution can be computed as $\mathbb{P}(\text{row}) = \sum_{k \in D_{i*}} \mathbb{P}_G(\text{row}|D_{i*} = k^*)\, \mathbb{P}(D_{i*} = k)$. CTGAN achieves this using a conditional vector, generator loss and training-by-sampling. The conditional vector is simply a means of coding the choice of a particular discrete feature D_{i^*} and its value k^*. In essence, discrete features are coded into one hot mask vectors. For example, for two discrete features D_1 and D_2

taking values $D_1 = \{1,2,3\}$ and $D_2 = \{1,2\}$, the condition $D_2 = 1$ is represented by vectors $\mathbf{m}_1 = [0,0,0]$ and $\mathbf{m}_2 = [1,0]$. The final conditional vector is then derived as $\mathbf{m}_1 \oplus \ldots \oplus \mathbf{m}_{N_d}$, hence in this example $cond = [0,0,0,1,0]$. Training the conditional generator requires an additional component to the loss compared to the generator of a traditional GAN as the conditional generator can produce any one hot mask vector for discrete features, not necessarily the correct *cond*; this is addressed by adding the cross entropy between the generated and true *cond* as a penalty to the generator loss. Finally, having constructed the *cond* vector, the motivation behind the conditional generator is to appropriately sample it so as to evenly explore all values in discrete features, in spite of their infrequency. This is achieved by training-by-sampling, which involves random selection of a discrete feature, such that each feature is selected with equal probability. Following this, a probability mass function is constructed across all values that the feature can take, with the probability mass associated with each value being the log of the frequency of the value.

2.3 Variational autoencoders

Currently, variational inference underlies the most realistic of generative models, especially for imaging and text, and can scale to large datasets (Kingma & Welling, 2014; Rezende et al., 2014; Hoffman et al., 2013; Gregor et al., 2014; 2015). The challenge that variational inference seeks to tackle is the following: dataset $\mathbf{X}$ with datapoints $\mathbf{x}_i$ is generated by a process based on unobserved latent variable $\mathbf{z}$. A value $\mathbf{z}_i$ is first generated from a prior distribution $p_\theta(\mathbf{z})$ and the value $\mathbf{x}_i$ is then generated from the conditional distribution $p_\theta(\mathbf{x}|\mathbf{z})$. Inference requires finding the marginal likelihood $p_\theta(\mathbf{x})$ which requires marginalisation of latent variables $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{z})\, p_\theta(\mathbf{x}|\mathbf{z})\, dz$; this integration is typically intractable. Finding the posterior distribution of latent variable $\mathbf{z}$, $p_\theta(\mathbf{z}|\mathbf{x}) = \frac{p_\theta(\mathbf{x}|\mathbf{z})\, p_\theta(\mathbf{z})}{p_\theta(\mathbf{x})}$, is also typically intractable. To generate synthetic data it is necessary to replicate the process of data generation and thus maximum likelihood or

maximum a posteriori estimation of the parameters θ is needed – this too requires marginalisation over the latent variables $\mathbf{z}$ which is intractable (Kingma & Welling, 2014). Variational inference is an approach for efficient inference in the face of these intractable distributions. The core idea is to reformulate the inference problem to an optimisation problem by learning parameters of an approximation to the true intractable posterior $p_\theta(\mathbf{z}|\mathbf{x})$, which is done by introducing a parameterised variational family $q_\phi(\mathbf{z})$, then learning the parameters ϕ that gives the best approximation to the true posterior based on minimising KL(q||p) i.e. $\phi^* = \arg\min KL(q_\phi(\mathbf{z})||p_\theta(\mathbf{z}|\mathbf{x}))$ (Jordan et al., 1999). It is not possible to work directly with $\arg\min KL(q_\phi(\mathbf{z})||p_\theta(\mathbf{z}|\mathbf{x}))$ as the posterior $p_\theta(\mathbf{z}|\mathbf{x})$ is unknown but, as the marginal likelihood $p_\theta(\mathbf{x})$ is independent of ϕ, the optimisation problem can be reformulated to $\phi^* = \arg\min KL(q_\phi(\mathbf{z})||\, p_\theta(\mathbf{z},\mathbf{x})) = \arg\min \mathbb{E}_{q_\phi(\mathbf{z})}[\log \frac{q_\phi(\mathbf{z})}{p_\theta(\mathbf{z},\mathbf{x})}]$. This can be written:

$$\arg\max \mathbb{E}_{q_\phi(\mathbf{z})}[\log \frac{p_\theta(\mathbf{z},\mathbf{x})}{q_\phi(\mathbf{z})}] = \arg\max \log p_\theta(\mathbf{x}) - KL(q_\phi(\mathbf{z})||p_\theta(\mathbf{z}|\mathbf{x}))$$

where the first term is the log marginal likelihood and the second is the Kullback-Leibler (KL) divergence between the true and approximate posterior. It is known as the negative free energy or evidence lower bound (ELBO), as it is the lower bound on the log marginal likelihood (as KL divergence is non-negative), also demonstratable by Jensen's inequality: $\mathbb{E}_{q_\phi(\mathbf{z})}\left[\log \frac{p_\theta(\mathbf{z},\mathbf{x})}{q_\phi(\mathbf{z})}\right] \leq \log \mathbb{E}_{q_\phi(\mathbf{z})}\left[\frac{p_\theta(\mathbf{z},\mathbf{x})}{q_\phi(\mathbf{z})}\right] = \log\, p_\theta(\mathbf{x})$. The ELBO can be alternatively written as:

$$\arg\,\max\, \mathbb{E}_{q_\phi(\mathbf{z})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - KL(q_\phi(\mathbf{z})||p_\theta(\mathbf{z}))$$

where the first term is the expected reconstruction error and the second is the KL divergence between the prior and posterior approximation (a regulariser keeping the posterior approximation close to the prior). The goal is to maximise the ELBO with regard to both variational parameters ϕ and generative model parameters θ.

In order for variational inference to be successful, two issues must be addressed. The first is the need for rich flexible approximate posterior distributions that can capture the true posterior and the second is the need for efficient computation of the derivatives of the ELBO i.e. $\nabla_{\phi}\mathbb{E}_{q_{\phi}(\mathbf{z})}[\log p_{\theta}(\mathbf{x}|\mathbf{z})]$. These are in part addressed by the two key characteristics of variational inference approaches: amortised inference and the reparameterization trick.

The motivation for amortised variational inference is that if a single $q_{\phi}(\mathbf{z})$ is shared across all datapoints it cannot simultaneously be a good fit for all of them. It is theoretically possible to use a different ϕ for each datapoint but the whole dataset would need to be iterated over to update ϕ as θ is updated. Amortised variational inference is a solution which involves learning a mapping from datapoints $\mathbf{x}$ to latent variables $\mathbf{z}$ and variational parameters ϕ using a distribution of the form $q_{\phi}(\mathbf{z}|\mathbf{x})$ which is the inference network (Kingma & Welling, 2014; Rezende et al., 2014; Gershman & Goodman, 2014). Then, learning ϕ corresponds to learning a mapping rather than a particular variational approximation. Using an inference network is advantageous as it avoids the need to learn variational parameters for each data point, with a set of global shared variational parameters ϕ computed instead. This allows amortisation of the cost of inference, making inference far more efficient. A widely used choice of parameterised mapping is diagonal Gaussian distributions of the form $q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(z \mid \mu_{\phi}(\mathbf{x}), \sigma_{\phi}(\mathbf{x}))$, where the mean $\mu_{\phi}(\mathbf{x})$ and standard deviation $\sigma_{\phi}(\mathbf{x})$ functions are specified with deep neural networks.

VAEs combine variational inference with deep learning. In VAEs, both the inference network $q_{\phi}(\mathbf{z}|\mathbf{x})$ (encoder) which maps samples $\mathbf{x}$ to latent representation $\mathbf{z}$ and the generative network $p_{\theta}(\mathbf{x}|\mathbf{z})$ (decoder) which maps the latent representation $\mathbf{z}$ to samples $\mathbf{x}$ are neural networks

(Kingma and Welling, 2014). A common choice for VAEs is to set the prior of latent variable $\mathbf{z}$ to be a centered isotropic multivariate Gaussian distribution $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{z} \mid 0, \mathbf{I})$ and the variational approximate posterior $q_\phi(\mathbf{z}|\mathbf{x})$ to be a multivariate Gaussian distribution with diagonal covariance matrix $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z} \mid \mu_\phi(\mathbf{x}), diag(\sigma_\phi^2(\mathbf{x})))$ where $\mu_\phi(\mathbf{x})$ and $\sigma^2{}_\phi(\mathbf{x})$ are the posterior approximation mean and standard deviation and are a function of datapoints $\mathbf{x}$, with the function being a neural network parameterised with weights ϕ. The distributions for the generative model $p_\theta(\mathbf{x}|\mathbf{z})$ are chosen to suit the data type: multivariate Gaussian distribution for continuous data and a Bernoulli distribution for binary data, whose parameters are a function of latent variables $\mathbf{z}$, with the function being a neural network.

2.3.1 Tabular VAE

Tabular VAE adapts the conventional VAE to generate data of mixed feature types (Xu et al., 2019). The inference network $q_\phi(\mathbf{z}|\mathbf{x})$ is unchanged from the conventional VAE and the variational approximate posterior is set as a multivariate Gaussian distribution with diagonal covariance matrix $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu, \sigma\mathbf{I})$. The generative network $p_\theta(\mathbf{x}|\mathbf{z})$ is modified in order to appropriately model different feature types. For N_c continuous features, random variables α_i representing the value of a feature within a mode and β_i the one hot vector indicating the mode (where $1 \leq i \leq N_c$) are used and assumed to follow Gaussian distributions with different means and variances and a categorical probability mass function, respectively. For N_d discrete variables, random variables $\mathbf{d}_i$ (where $1 \leq i \leq N_d$) are used and assumed to follow a categorical probability mass function. In the generative network, tanh is used as a non-linear activation function to generate α_i while softmax is used to generate β_i and $\mathbf{d}_i$. $p_\theta(\mathbf{x}|\mathbf{z})$ is then derived as a joint distribution of all α_i, β_i and $\mathbf{d}_i$ variables: $p_\theta(\mathbf{x}|\mathbf{z}) = \prod_{i=1}^{N_c} \mathbb{P}(\hat{\alpha}_i = \alpha_i) \prod_{i=1}^{N_c} \mathbb{P}\big(\hat{\beta}_i = \beta_i\big) \prod_{i=1}^{N_d} \mathbb{P}(\hat{\mathbf{d}}_i = \mathbf{d}_i)$ where $\hat{\alpha}_i$, $\hat{\beta}_i$ and $\hat{\mathbf{d}}_i$ are random variables.

2.4 Normalising flows

Although VAEs have enjoyed substantial success as deep generative models, one of the core limitations of variational inference is the choice of variational posterior approximation. The crux of variational inference is the estimation of an intractable posterior by a class of known parametric probability distributions, within which the best approximation to the true posterior is found. The true posterior can only be recovered if it is within this chosen variational family. However, available choices for families of variational posterior approximations are typically simple e.g. diagonal covariance Gaussians, for the purposes of efficient inference, so are not sufficiently rich to capture the complex true posterior. Indeed, the disadvantage of variational methods is that even in the asymptotic regime, the true posterior is often not recovered. Previous work has identified that limited posterior approximations do have a significant negative impact: two observed problems have been underestimation of the variance of the posterior distribution which results in poor predictions, and biases in maximum likelihood and maximum a posteriori estimates of model parameters θ which harms generative performance (Turner & Sahani, 2011).

Much work has therefore focussed on improving posterior approximation by designing more complex and flexible variational families that can capture the true posterior (Nalisnick et al., 2016; Salimans et al., 2015; Tran et al., 2015), for which there is evidence of improved inference and VAE performance (Mnih & Gregor, 2014). However, designing expressive multi-modal posterior approximations which are still tractable and scalable is a challenge. Approaches proposed have included using mixture models for approximate posteriors but this limits scalability, a key advantage of variational inference, as for each parameter update, gradients for each mixture component must be computed which is computationally expensive (Jaakkola & Jordan, 1998; Gershman et al., 2012).

Normalising flows is a new approach to construct arbitrarily complex, flexible, tractable and scalable approximate posteriors, which have been shown to outperform simple approximate posteriors and competitor approaches for posterior approximation on imaging datasets (Rezende & Mohamed, 2015). The core idea is that rich, complex and multi-modal probability densities can be learnt by starting with a simple base probability density e.g. independent Gaussian and applying a sequence of transformations with computable inverses and Jacobians. The degree of complexity can be controlled by the number of transformations i.e. the flow length. In the asymptotic regime, the posterior approximations that can be specified using normalising flows can recover the true posterior, thereby obviating a major limitation of variational inference.

The normalising flow approach is formalised as follows (Rezende & Mohamed, 2015). Let f be the transformation and g be its inverse, $g = f^{-1}$. For latent variable $\mathbf{z}$ with distribution $\mathrm{q}(\mathbf{z})$, the latent variable after a transformation is $\mathbf{z}' = f(\mathbf{z})$ with density $q(\mathbf{z}') = q(\mathbf{z}) \left|\det \frac{\partial f^{-1}}{\partial \mathbf{z}'}\right| = q(\mathbf{z}) \left|\det \frac{\partial f}{\partial \mathbf{z}}\right|^{-1}$ by applying change of variables chain rule (inverse function theorem). Analogously, when a sequence of transformations is applied to latent variable $\mathbf{z}_0$ with distribution $\mathrm{q}_0(\mathbf{z}_0)$, the final latent variable after the transformations is $\mathbf{z}_K = f_k \circ \ldots \circ f_2 \circ f_1(\mathbf{z}_0)$ with density $q(\mathbf{z}_K) = q(\mathbf{z}_0) \left|\det \frac{\partial f_1}{\partial \mathbf{z}_0}\right|^{-1} \ldots \left|\det \frac{\partial f_K}{\partial \mathbf{z}_{K-1}}\right|^{-1}$ and log density $\log q_K(\mathbf{z}_K) = \log q_0(\mathbf{z}_0) - \sum_{k=1}^{K} \log \left|\det \frac{\partial f_k}{\partial \mathbf{z}_{k-1}}\right|$. $\mathrm{q}_K(\mathbf{z}_K)$ is then used as the approximate posterior $\mathrm{q}_\phi(\mathbf{z}|\mathbf{x})$. A key advantage of normalising flows is that expectations under the complex density q_K can be computed using expectations under the simple density q_0: $\mathbb{E}_{q_K}[h(\mathbf{z})] = \mathbb{E}_{q_0}[h(f_K \circ f_{K-1} \circ \ldots \circ f_1(\mathbf{z}_0))]$. Thus, to sample from the complex density, it is sufficient to sample from the base density, so q_K itself does not need to be known. The effect of the flows are contractions and

expansions on the base density. The Jacobian determinant of the transformation reflects the degree of local volume expansion around the data. The challenge is then to choose an appropriate set of transformations. Designing invertible parametric transformation is not difficult (one could make use of neural networks) but computing the Jacobian of functions with high-dimensional domains and the determinant of large matrices is computationally expensive: it typically has complexity $O(D^3)$ where D is the hidden dimension. For the normalising flow approach to be effective, transformations associated with efficient computation of the Jacobian determinant are needed.

The normalising flow approach modifies the ELBO that is optimised (using the fact that expectations under the complex density can be computed using expectations under the simple density):

$$F(\theta, \varphi) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log q_\phi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{x}, \mathbf{z})] = \mathbb{E}_{q_K(\mathbf{z}_K)}[\log q_K(\mathbf{z}_K) - \log p_\theta(\mathbf{x}, \mathbf{z}_K)] =$$

$$\mathbb{E}_{q_0(\mathbf{z}_0)}[\log q_0(\mathbf{z}_0) - \log p_\theta(\mathbf{x}, \mathbf{z}_K)] - \mathbb{E}_{q_0(\mathbf{z}_0)}[\sum_{k=1}^{K} \log \left|\det \frac{\partial f_k}{\partial \mathbf{z}_{k-1}}\right|].$$

Additional terms have only linear time complexity.

The normalising flow method also necessitates modification to amortised variational inference. Flow parameters are typically also data-dependent i.e. functions of the datapoints **x**. The inference network is therefore used as a mapping from datapoints **x** to the parameters of the base density $q_0 \sim \mathcal{N}(\mu, \sigma^2)$ as well as flow parameters λ – the flow parameters are generated by a hypernetwork attached to the inference network.

The following section reviews the types of normalising flows that are investigated in this work.

2.4.1 Planar flows

Planar flows use transformations of the form $f(\mathbf{z}) = \mathbf{z} + \mathbf{u}h(\mathbf{w}^T\mathbf{z} + b)$ where $\mathbf{u}, \mathbf{w} \in \mathbb{R}^D$, $b \in \mathbb{R}$ and h is a smooth element-wise non-linear function (Rezende & Mohamed, 2015). The flow parameters are $\mathbf{u}$, $\mathbf{w}$ and b. The transformation is invertible and the Jacobian determinant of the transformation can be computed using the matrix determinant lemma: $\frac{\partial f}{\partial \mathbf{z}} = h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w} \Rightarrow \det\frac{\partial f}{\partial \mathbf{z}} = \det(I + \mathbf{u}h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w}^T) = 1 + \mathbf{u}^T h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w}$. This can be computed in linear time O(D). Thus, if the base density is $q_0(\mathbf{z}_0)$ and the final density after transformations is $q_K(\mathbf{z}_K)$, $\log q_K(\mathbf{z}_K) = \log q_0(\mathbf{z}_0) - \sum_{k=1}^{K} \log\left|1 + \mathbf{u}_k^T h'\left(\mathbf{w}_k^T\mathbf{z}_{k-1} + b_k\right)\mathbf{w}_k\right|$. The sequence of contractions and expansions resulting from the transformations is perpendicular to the hyperplane $\mathbf{w}^T\mathbf{z} + b = 0$.

For planar flows, the number of parameters is $2EDK + EK$ where E is the number of output units of the inference network, D is the dimension of latent variables $\mathbf{z}$ and K is the number of flows. The EDK terms arises from the flow parameters $\mathbf{u}$ and $\mathbf{w}$ while the EK term is due to b. Of the four flow types studied in this work, planar flows has the fewest parameters.

2.4.2 Sylvester flows

Sylvester flow is a generalisation of planar flow (van den Berg et al., 2018). The transformation in planar flows is equivalent to a MLP layer with one unit and a skip connection which creates a single neuron bottleneck, limiting the flexibility of the transformation. Sylvester flows replaces

this with a MLP layer with M hidden units and a skip connection. The transformation is of the form $f(\mathbf{z}) = \mathbf{z} + \mathbf{A}h(\mathbf{B}\mathbf{z} + \mathbf{b})$ with $\mathbf{A} \in \mathbb{R}^{D\times M}$, $\mathbf{B} \in \mathbb{R}^{M\times D}$, $\mathbf{b} \in \mathbb{R}^{M}$, and M ≤ D. The flow parameters are $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{b}$. An issue is that this transformation is not generally invertible so a special case, with orthogonal and triangular matrices as $\mathbf{A}$ and $\mathbf{B}$, is used to ensure invertibility: $f(\mathbf{z}) = \mathbf{z} + \mathbf{Q}\mathbf{R}h(\widetilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z} + \mathbf{b})$ where $\mathbf{Q}$ is a orthogonal D x M matrix with columns that form an orthonormal set of vectors, $\mathbf{R}$ and $\widetilde{\mathbf{R}}$ are upper triangular M × M matrices and h is a smooth non-linear function with positive derivative. For invertibility of the transformation, the diagonal entries of $\mathbf{R}$ and $\widetilde{\mathbf{R}}$ must satisfy $r_{ii}\tilde{r}_{ii} > -\frac{1}{\|h\prime\|_\infty}$ and $\widetilde{\mathbf{R}}$ must be invertible. The Jacobian determinant can be computed efficiently using Sylvester determinant identity. For the general transformation this is: $\frac{\partial f}{\partial \mathbf{z}} = diag\big(h'(\mathbf{B}\mathbf{z} + \mathbf{b})\big)\mathbf{B}\mathbf{A} \Rightarrow \det\frac{\partial f}{\partial \mathbf{z}} = \det(\mathbf{I}_M + diag\big(h'(\mathbf{B}\mathbf{z} + \mathbf{b})\big)\mathbf{B}\mathbf{A})$. For the special case that is invertible, this is: $\det\Big(\mathbf{I}_M + diag\Big(h'\big(\widetilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z} + \mathbf{b}\big)\Big)\widetilde{\mathbf{R}}\mathbf{Q}^T\mathbf{Q}\mathbf{R}\Big) = \det(\mathbf{I}_M + diag\Big(h'\big(\widetilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z} + \mathbf{b}\big)\Big)\widetilde{\mathbf{R}}\mathbf{R})$. This can be computed in O(M) time as $\widetilde{\mathbf{R}}\mathbf{R}$ is upper triangular, so can be computed by the product of the diagonal elements: $\prod_{i=1}^{M}(1 + h'\big(\widetilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z} + \mathbf{b}\big))_i\widetilde{\mathbf{R}}\mathbf{R}_{i,i}$. Finally, the orthogonality of $\mathbf{Q}$ as it is updated is maintained by a iterative procedure to construct orthogonal matrices $\mathbf{Q}^{k+1} = \mathbf{Q}^k(\mathbf{I} + \frac{1}{2}\Big(\mathbf{I} - {\mathbf{Q}^k}^T\mathbf{Q}^k\Big))$ which converges if $\Big\|{\mathbf{Q}^0}^T\mathbf{Q}^0 - \mathbf{I}\Big\|_2$ where $\|\mathbf{X}\|_2 = \lambda_{max}(\mathbf{X})$, the largest singular value of $\mathbf{X}$. This procedure is differentiable so gradients with respect to $\mathbf{Q}_0$ can be calculated.

For Sylvester flows, the number of parameters is $KE \times (MD + 2M^2 + M)$ where E, D and K are as already described. The MD term owes to the flow parameter $\mathbf{Q}$, M^2 terms is due to $\mathbf{R}$ and $\widetilde{\boldsymbol{R}}$ and M is accounted for by $\mathbf{b}$. Of the four flow types studied in this work, Sylvester flows has the most parameters.

2.4.3 Non-linear Independent Component Estimation

Non-linear Independent Component Estimation (NICE) and real-valued non-volume preserving transformation (RealNVP) are both bijective transformations which are trivially invertible and have tractable Jacobian determinant (Dinh et al., 2014; 2017). These can be designed to be highly non-linear to learn complex high-dimensional densities. The core idea is that part of the input undergoes a transformation which is easily invertible but which depends on the rest of the input in a complex way. The transformations have triangular Jacobian, making use of the property whereby triangular matrices have efficiently computable determinants (the determinant is the product of the diagonal elements). When a sequence of transformations are applied, the inverse is the composition of each layer's inverse and the Jacobian determinant is the product of each layer's Jacobian determinant.

The core idea underlying the NICE transformation is the coupling layer in which the latent variable $\mathbf{z}$ is partitioned into two disjoint subsets $\mathbf{z}_1$ and $\mathbf{z}_2$, one subset is transformed and the other is kept unchanged: $\mathbf{y}_1 = \mathbf{z}_1$ and $\mathbf{y}_2 = g(\mathbf{z}_2, m(\mathbf{z}_1))$, where m is a function e.g. a neural network and g is the coupling law (Dinh et al., 2014). This transformation is trivially invertible: $\mathbf{z}_1 = \mathbf{y}_1$ and $\mathbf{z}_2 = g^{-1}(\mathbf{y}_2, m(\mathbf{y}_1))$. The Jacobian of the coupling layer is $\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{bmatrix} I_d & 0 \\ \frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_1} & \frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_2} \end{bmatrix}$. The determinant of the Jacobian is $\det \begin{bmatrix} I_d & 0 \\ \frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_1} & \frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_2} \end{bmatrix} = \frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_2}$. NICE uses an additive coupling law $g(a, b) = a + b$, such that $\mathbf{y}_1 = \mathbf{z}_1$ and $\mathbf{y}_2 = \mathbf{z}_2 + m(\mathbf{z}_1)$. This is trivially invertible as $\mathbf{z}_1 = \mathbf{y}_1$ and $\mathbf{z}_2 = \mathbf{y}_2 - m(\mathbf{y}_1)$. The inverse of the transformation has computational complexity equal to the forward transformation and does not necessitate computing m so this can be arbitrarily complex and difficult to invert. The Jacobian of the transformation has a zero upper triangular part so the transformation has unit Jacobian determinant (equal to 1). Thus, NICE is a volume-

preserving flow. Given that additive coupling layers have Jacobian determinant of 1, a diagonal scaling matrix $\mathbf{S}$ is multiplied to the output of each layer. The actual transformation is thus $\mathbf{y}_1 = \mathbf{S}_1\mathbf{z}_1$ and $\mathbf{y}_2 = \mathbf{S}_2(\mathbf{z}_2 + m(\mathbf{z}_1))$ where $diag(\mathbf{S}_1, \mathbf{S}_2) = \mathbf{S}$. The determinant of the Jacobian becomes $\det(\mathbf{S}) = \prod_{i=1}^{D} \mathbf{S}_{i,i}$. This has the effect of placing varying importance on different dimensions of the latent variable. As one subset of $\mathbf{z}$ is unchanged with application of a coupling layer, layers must be composed and the subsequent layer must apply the transformation to the alternative subset relative to the previous layer so that all components of $\mathbf{z}$ are transformed. In addition, components of $\mathbf{z}$ are mixed with random permutation prior to separation into subsets so that partitionings of $\mathbf{z}$ vary and different subsets of variables undergo transformation.

For NICE, the number of parameters is KLN^2 where L is the number of layers in the MLP, N is the average number of units in each layer, and K is the number of flows.

2.4.4 Real-Valued Non-Volume Preserving transformation

RealNVP similarly makes use of a coupling layer but with an affine coupling law where $g(a, b) = a \odot b_1 + b_2$ (Dinh et al, 2017). The transformation is $\mathbf{y}_1 = \mathbf{z}_1$ and $\mathbf{y}_2 = \mathbf{z}_2 \odot \exp\big(s(\mathbf{z}_1)\big) + t(\mathbf{z}_1)$, where s and t are scale and translation functions e.g. neural networks. Again, this is trivially invertible: $\mathbf{z}_1 = \mathbf{y}_1$ and $\mathbf{z}_2 = \mathbf{y}_2 - t(\mathbf{y}_1) \odot \exp(-s(\mathbf{y}_1))$. The inverse of the transformation can be computed with the same computational complexity as the forward transformation and does not necessitate computing the inverse of s or t so these can be arbitrarily complex. The Jacobian of the transformation has $\frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_2} = diag(\exp[s(\mathbf{z}_1)])$ where $diag$ is the diagonal matrix with elements corresponding to the vector $\exp[s(\mathbf{z}_1)]$. The Jacobian is a triangular matrix so the determinant can be computed as $\exp(\sum_i s(\mathbf{z}_1)_i)$. Computation of the Jacobian does not require computing the Jacobian of s or t, hence the complexity of s and t are

not limited. The partitioning of $\mathbf{z}$ into two subsets is done using a binary mask b: $\mathbf{y} = b \odot \mathbf{z} + (1 - b) \odot (\mathbf{z} \odot \exp\big(s(b \odot \mathbf{z})\big) + t(b \odot \mathbf{z}))$, specifically a checkerboard pattern mask. These increase the variation in the partitionings of $\mathbf{z}$.

For RealNVP, the number of parameters is $2KLN^2$ as it makes use of two MLPs (s and t), where K, L and N are as already described.

2.5 Summary

Table 1 summarises the key aspects of the four types of normalising flow used in this study. This chapter reviewed deep generative models for tabular data. A review of deep learning predictive models is presented in the next chapter.

Table 1. Summary of four types of normalising flows.

	Transformation	Jacobian determinant
Planar[1]	$\mathbf{z} + \mathbf{u}h(\mathbf{w}^T\mathbf{z} + b)$	$1 + \mathbf{u}^T h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w}$
Sylvester[2]	$\mathbf{z} + \mathbf{Q}\mathbf{R}h(\widetilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z} + \mathbf{b})$	$\prod_{i=1}^{M}(1 + h'\big(\widetilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z} + \mathbf{b}\big))_i \widetilde{\mathbf{R}}\mathbf{R}_{i,i}$
NICE[3]	$(\mathbf{S}_1\mathbf{z}_1, \mathbf{S}_2\big(\mathbf{z}_2 + m(\mathbf{z}_1)\big))$	$\prod_{i=1}^{D}\mathbf{S}_{i,i}$
RealNVP[4]	$(\mathbf{z}_1, \mathbf{z}_2 \odot \exp\big(s(\mathbf{z}_1)\big) + t(\mathbf{z}_1))$	$\exp\,(\sum_i s(\mathbf{z}_1)_i)$

[1]Rezende & Mohamed, 2015; [2]van den Berg et al., 2018; [3]Dinh et al., 2014; [4]Dinh et al., 2017.

Chapter 3: Deep learning models for predictive tasks on tabular data

This chapter serves as a comprehensive review of the main families of current state-of-the-art deep learning models developed for predictive tasks on tabular data (Gorishniy et al., 2021).

3.1 Differentiable tree-based models

One family of deep learning models that has gained prominence for tabular data are differentiable trees (Popov et al., 2019; Chen, 2020; Yang et al., 2018). The development of the architecture is inspired by the success of decision trees on tabular data. The core idea of these models is to use neural networks to mimic decision trees but obviate the drawbacks of the latter, primarily their lack of differentiability. This is achieved by using smooth decision functions in tree nodes.

3.1.1 Neural oblivious decision ensembles

Neural oblivious decision ensembles (NODE) is based on CatBoost, which uses oblivious decision trees (ODTs) (Popov et al., 2019). ODTs use the same splitting feature and threshold in all nodes at the same depth. Because of this, they are weaker learners than conventional decision trees, but have the effect of reducing overfitting when they are ensembled. Moreover, inference is made efficient as binary splits can be computed in parallel rather than sequentially with conventional decision trees. The first key advantage of NODE is that it leverages ODTs but makes decision functions in nodes and decision tree routing differentiable, such that

backpropagation of gradients is possible. NODE achieves this using the entmax transformation for "soft" splitting decision functions in nodes. Entmax is similar to softmax but borrows concepts from sparsemax in order to transform vectors of continuous values to sparse discrete probability distributions (where, unlike softmax, a majority of probabilities are 0). However, it results in smoother decision functions than sparsemax so is more compatible with gradient-based optimisation. By enabling the learning of sparse decisions, entmax confers the model with the correct inductive bias (the same as shallow tree-based methods), but maintains the benefits of being able to learn these using standard gradient descent approaches. The second key advantage of NODE over shallow ODTs is that it facilitates multi-layer hierarchical representation. The architecture consists of stacking multiple layers made up of ensembles of ODTs, creating a "deep" decision tree. Layers can be trained end-to-end with gradient-based optimisation methods. This enables learning of both shallow and deep decisions with complex dependencies. NODE has been demonstrated to consistently outperform leading GBDT packages (Popov et al., 2019).

The NODE model is as follows (Popov et al., 2019). Each layer has m ODTs of depth d. The input **x** with dimension n is received by all trees, which splits the input along splitting features $f_1, f_2, \ldots, f_d$ and compares each feature to a learnable splitting threshold $b_1, b_2 \ldots, b_d$. In the traditional ODT, each tree outputs one of 2^d possibilities (arising from all combinations of d splits): the output from an individual tree is $h(\mathbf{x}) = R[\mathbb{1}(f_1(\mathbf{x}) - b_1), \ldots, \mathbb{1}(f_d(\mathbf{x}) - b_d)]$ where $\mathbb{1}(.)$ is the Heaviside function and R is a d-dimensional response tensor. However in the NODE model, the splitting feature choice function f_i and comparator function $\mathbb{1}(f_i(\mathbf{x}) - b_i)$ is substituted for continuous functions to render the tree differentiable. The splitting feature choice function is instead a weighted sum of features, where weights are derived by applying entmax to a learnable feature selection matrix **F**: $f_i(\mathbf{x}) = \sum_{j=1}^{n} x_j \cdot entmax(\mathbf{F}_{ij})$. Thus the entmax

transformation is also leveraged for feature selection. The Heaviside function is instead a two class entmax: $c_i(\mathbf{x}) = entmax([(f_i(\mathbf{x}) - b_i), 0])$. The outer product of all c_i is computed to generate choice tensor C. The output from an individual tree is the weighted sum of response tensor R with weights derived from the choice tensor C: $h(\mathbf{x}) = \sum_{i_1 \ldots i_d \in \{0,1\}^d} R_{i_1 \ldots i_d} \cdot C_{i_1 \ldots i_d}(\mathbf{x})$. The output from a NODE layer consists of the outputs of individual trees concatenated i.e. $[h_1(\mathbf{x}), \ldots, h_m(\mathbf{x})]$. A DenseNet architecture is used for the full NODE model, with each layer receiving input that is the outputs of all previous layers concatenated. The final prediction is computed from the average of all layers. Earlier layers have the role of feature selection with later layers using these for the final prediction. Model parameters that are trained are F, b and R.

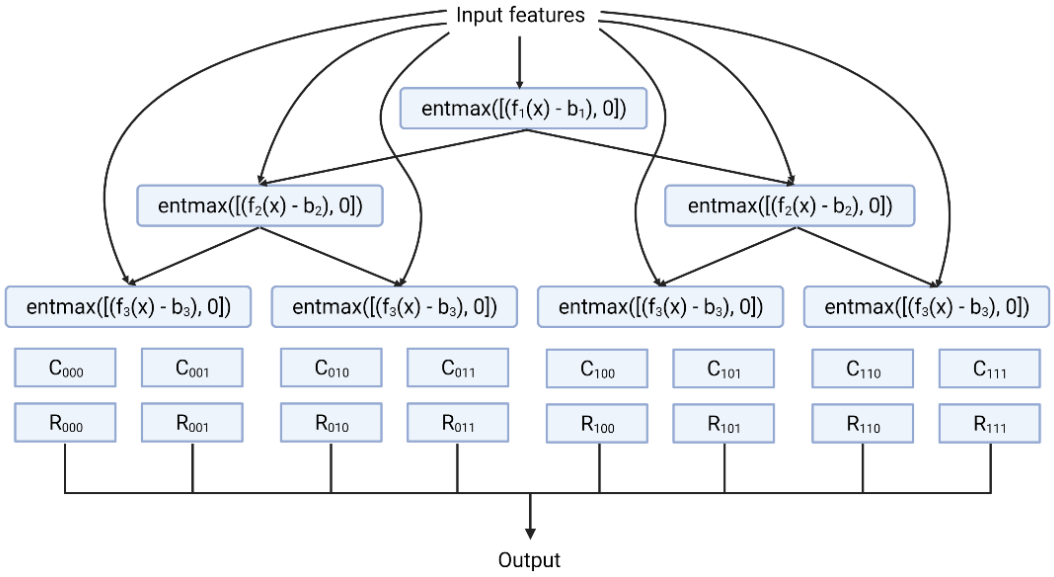

Figure 1. Architecture of individual tree in NODE. Architecture is shown with tree depth of 3. f_i: splitting feature choice function; b_i: splitting threshold; C: choice tensor; R: response tensor (adapted from Popov et al., 2019).

3.1.2 Quantum Forest

Quantum Forest is the more general counterpart to NODE (Chen, 2020). It also uses the basic component of differentiable trees, ensembling these to create a layer and stacks layers to create a deep "forest". However, it uses conventional non-oblivious decision trees as opposed to ODTs. Similar to NODE, it combines the advantages of both trees and neural networks – sparse decisions that confer the appropriate inductive bias and differentiability which allows training by stochastic gradient descent (SGD). Like NODE which uses a learnable feature selection matrix with entmax transformation, Quantum Forest has a sparse attention mechanism for feature selection so decisions only rely on a few important features. Although, Quantum Forest uses an entmax transformation as part of this feature selection, it is suggested entmax does not enforce a sufficiently high degree of sparsity. The attention mechanism can be improved by data-aware initialisation, whereby feature importance is first estimated using GBDT packages e.g. LiteMORT and used to initialise the attention weights, as opposed to randomly initialising these. This makes the attention mechanism sparser and improves attention on the most important features. Quantum Forest has been demonstrated to outperform GBDT (Chen, 2020).

The Quantum Forest model is as follows (Chen, 2020). Similar to NODE, the splitting feature choice function is substituted by a weighted sum of features with weights derived from entmax applied to a learnable attention vector $\mathbf{A}$, that assigns different weights to different features of $\mathbf{x}$. The Heaviside gating function of classical decision trees is substituted by a sigmoid function. Thus the gating function within nodes of the tree is $g(\mathbf{A}, \mathbf{x}, b) = \sigma(entmax(\mathbf{A})\mathbf{x} - b)$ where b is a learnable threshold. These two modifications render the tree differentiable. Each node directs its input to two child nodes with probability computed by the gating function. The probability of the input being directed to each final leaf node is the product of the probabilities of all nodes in the path from the input to the final leaf node: $p = \prod_n g_n$ for $n \in$

$\{n_1, n_2, \ldots, n_d\}$ where d is the depth of the tree. The response at leaf nodes is denoted by Q. The output from an individual tree is the weighted sum of the response Q with weights being the probabilities p: $y(\mathbf{x}) = \sum_j p_j Q_j(\mathbf{x})$. Model parameters that are trained are A, b and Q.

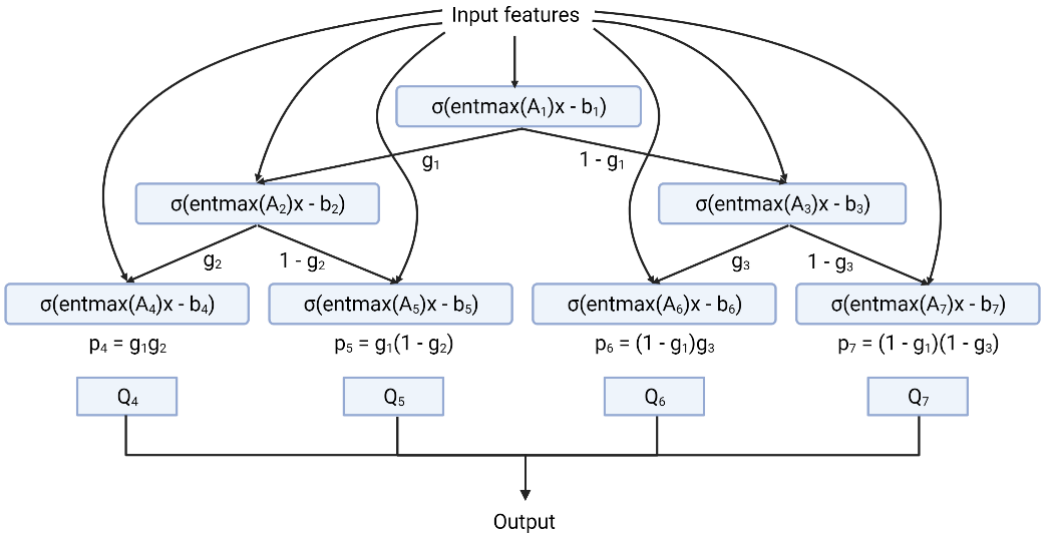

Figure 2. Architecture of individual tree in Quantum Forest. Architecture is shown with tree depth of 3. $\mathbf{A}_i$: attention vector; b_i: splitting threshold; g_i: output of gating function; Q: response tensor (adapted from Chen, 2020).

3.1.3 Deep neural decision trees

Deep neural decision trees (DNDT) use a neural network to mimic a decision tree (Yang et al., 2018). Like NODE and Quantum Forest, this confers the advantage of being differentiable and thus trainable end-to-end with SGD rather than resorting to greedy splitting which can be globally sub-optimal as is the case in classical decision trees. With DNDT, it is possible to simultaneously learn the structure and parameters of the tree with backpropagation only, unlike classical decision trees which require different strategies for these e.g. splitting and score matrix. This confers DNDT with the capacity to find more optimal solutions. However, as it simulates the functions of a decision tree, DNDT inherits their key advantage of intrinsic

interpretability. DNDT has been shown to outperform simple neural networks (Yang et al., 2018).

The DNDT model is as follows (Yang et al., 2018). The key aspect of DNDT is the use of a soft binning split decision function which is an approximation to the hard binning function in classical decision trees, but which renders the DNDT differentiable. In the hard binning function, a continuous scalar x is converted to a vector representing the index of bins to which x is assigned; the cut points of the bins, which can be termed $[\beta_1, \beta_2, \dots, \beta_n]$ if there are $n+1$ bins, are learnable parameters. In contrast, the soft binning function uses a one layer neural network with softmax: $f(x) = softmax(\frac{wx+b}{\tau})$ with $w = [1, 2, \dots, n+1]$, $b = [0, -\beta_1, -\beta_1 - \beta_2, \dots, -\beta_1 - \beta_2 - \cdots - \beta_n]$ and τ being a temperature factor. This produces an output that is close to a one hot encoding of the output when applying the hard binning function to x. For an continuous input $\mathbf{x}$ with D features, the soft binning function is applied to each feature x_d independently. $\mathbf{z}$ is then computed as $\mathbf{z} = f_1(x_1) \otimes f_2(x_2) \otimes \dots \otimes f_D(x_D)$ where $\otimes$ is the Kronecker product; $\mathbf{z}$ is close to a one hot vector representing the index of the final leaf node to which the input $\mathbf{x}$ is directed. Finally, linear classifiers at each leaf node are used to classify the inputs directed there. Model parameters that are trained are the cut points of bins and classifiers at leaf nodes. DNDT implicitly performs feature selection as for some unimportant features all cut points are inactive (cut points are pushed outside the boundary of the data). These features are then ignored and do not affect the prediction.

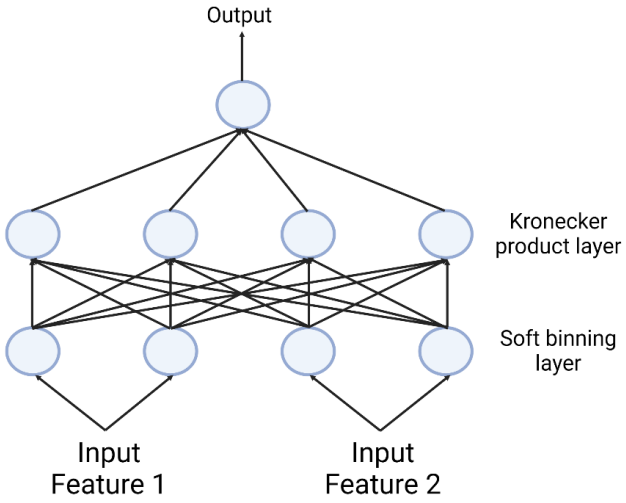

Figure 3. DNDT architecture. Architecture is shown for 2 input features (adapted from Yang et al., 2018).

3.2 Attention-based models

Another family of models that have become popular for tabular data is attention-based models (Arik & Pfister, 2019; Huang et al., 2020; Somepalli et al., 2021).

3.2.1 TabNet

TabNet uses deep neural networks to learn hyperplane decision boundaries that resemble those learnt by decision trees (Arik & Pfister, 2019). This captures the advantages of deep learning such as gradient-based optimisation and integration into end-to-end learning pipelines, while maintaining performance akin to decision trees. To achieve this, feature selection is paramount for appropriate inductive bias. TabNet uses sequential attention to capture salient features. This involves a multi-step architecture with each step selecting a few important features, then using these chosen features to inform a prediction, which contributes to the overall prediction. The attention mechanism in each step is based on a sparsemax function, which is used to encode features into sparse learned masks, thereby selecting only a small subset of features. This is in

contrast to transformer-based models which use self-attention. The advantage of a learnable mask is soft feature selection with sparsity that can be controlled, unlike hard thresholds where a feature is either chosen or not. The feature selection process is also differentiable and can be trained end-to-end. Feature selection in TabNet is instance-wise i.e. features selected can vary for different input datapoints, unlike global methods such as forward selection and Lasso regularisation which select features for a whole dataset. The attention mechanism improves learning efficiency as the capacity of the model is only used for the most relevant features. It also confers interpretability as it is possible to understand which features are important, how they are combined and their contribution to the learned model. Once features are selected, these undergo highly non-linear processing and given the network depth and number of decision steps, complex dependencies can be learnt. In essence, the TabNet architecture facilitates an alternating process of reweighing of features and feedforward networks, which maximises learning capacity but avoids overfitting as a result of feature selection. Having a single model that performs feature selection and output mapping leads to more compact representations, which is a key advantage of TabNet. TabNet has been shown have superior performance to competing models on a range of open source tabular datasets (Arik & Pfister, 2019).

The TabNet model is as follows (Arik & Pfister, 2019). The overall structure is one of N decision steps. Each step takes as input the processed features of the previous step, performs feature selection using an attentive transformer which outputs a learnable mask $\mathbf{M}[\mathbf{i}] \in \mathbb{R}^{BxD}$, where i is the decision step, B is the number of samples and D is the number of features, inputs this mask into a feature transformer which outputs a processed feature representation that is aggregated for the overall decision and that serves as input to the next step. The computation within the attentive transformer is: $\mathbf{M}[\mathbf{i}] = sparsemax(\mathbf{P}[\mathbf{i}-\mathbf{1}] \cdot h_i(\mathbf{a}[\mathbf{i}-\mathbf{1}]))$ where $\mathbf{a[i-1]}$ is the input (processed features from previous step), h_i is a trainable fully connected layer and

$\mathbf{P[i-1]}$ is the prior scale that represents the extent a particular feature has been previously used. $\mathbf{P}[\mathbf{i}] = \prod_{j=1}^{i}(\gamma - \mathbf{M}[\mathbf{j}])$ where γ is a relaxation parameter that determines the number of decision steps a feature can be used in. The sparsity of the mask is further controlled by adding sparsity regularisation, $L_{sparse} = \sum_{i=1}^{N_{steps}} \sum_{b=1}^{B} \sum_{d=1}^{D} \frac{-\mathbf{M}_{b,d}[\mathbf{i}] \log(\mathbf{M}_{b,d}[\mathbf{i}] + \varepsilon)}{N_{steps} \cdot B}$, to the loss with coefficient λ_{sparse}. The mask provides information on feature importance: $\mathbf{M}_{\mathrm{b,d}}[\mathbf{i}]$ indicates the importance of the dth feature of the bth sample in step i and a mechanism exists to combine masks at different steps (weighted by the importance of each step in the overall decision) to obtain an aggregate feature importance mask. The selected features are processed in a feature transformer which consists of fully connected layers that are shared across all decision steps and layers which are specific step dependent. The output of the feature transformer is then split for the decision step output and input to subsequent step: $[\mathbf{d}[\mathbf{i}], \mathbf{a}[\mathbf{i}]] = f_i(\mathbf{M}[\mathbf{i}] \cdot \mathbf{f})$, where $\mathbf{d[i]} \in \mathbb{R}^{BxN_d}$ and $\mathbf{a[i]} \in \mathbb{R}^{BxN_a}$. The overall decision is the aggregation of the output of each decision step $\mathbf{d_{out}} = \sum_{i=1}^{N_{steps}} ReLU(\mathbf{d}[\mathbf{i}])$. Model parameters that are trained are the weights and biases of the fully connected layers of the attentive and feature transformer.

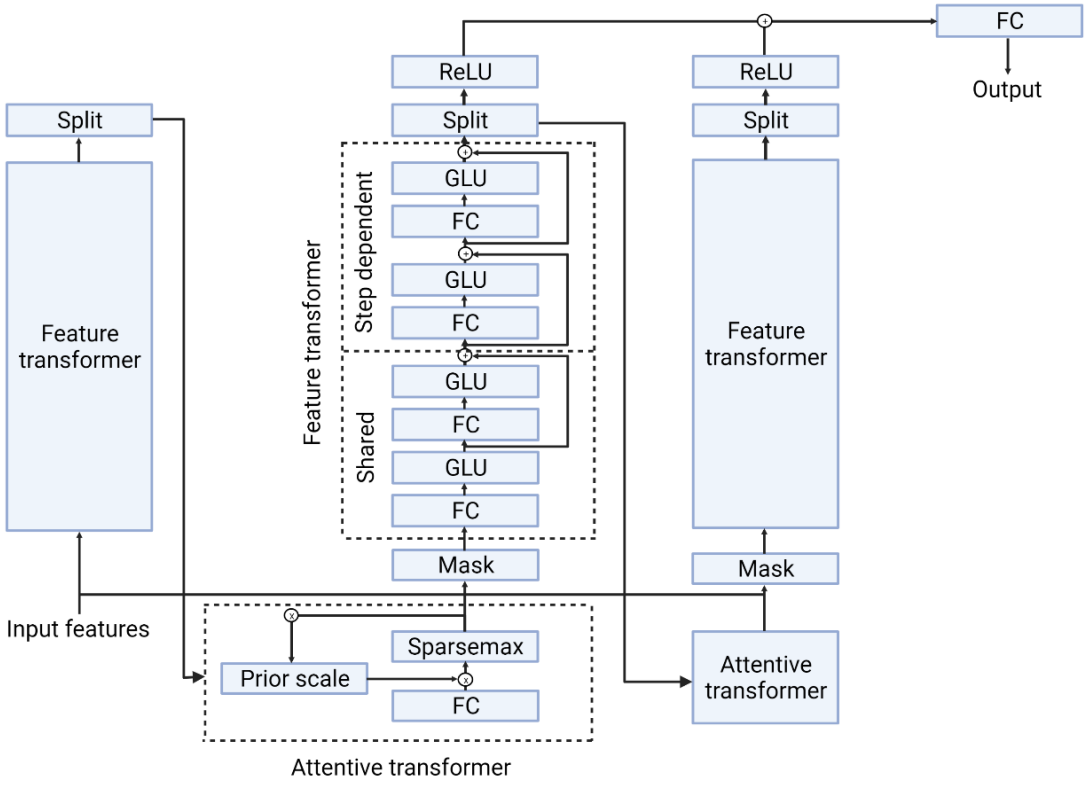

Figure 4. TabNet architecture. Architecture is shown for 2 decision steps. FC: fully connected layer; GLU: gated linear unit activation function; ReLU: rectified linear unit activation function (adapted from Arik & Pfister, 2019).

3.2.2 TabTransformer

TabTransformer is inspired by self-attention based transformers which have enabled state-of-the-art performance of natural language processing models, with their ability to create word embeddings (Vaswani et al., 2017). TabTransformer uses multi-head attention-based transformer layers to learn robust and efficient contextual embeddings of categorical features (Huang et al., 2020). Features that are associated result in embedding vectors close in space and similar classes within a feature are also close in embedding space, so contextual embeddings are interpretable. TabTransformer has been demonstrated to outperform other deep learning approaches and to equal the performance of GBDT (Huang et al., 2020).

The TabTransformer model is as follows (Huang et al., 2020). The input $\mathbf{x}$ is split into $\{\mathbf{x}_{cat}, \mathbf{x}_{cont}\}$ where $\mathbf{x}_{cat}$ represents all categorical features and $\mathbf{x}_{cont}$ all continuous features. Each of the categorical features are embedded into a parametric embedding, $e_{\phi i}(x_i)$, with a column embedding layer. $\mathbf{E}_{\phi}(\mathbf{x}_{cat})$, denoting the set of embeddings of all the categorical features, is fed into a stack of N transformer layers which transforms the parametric embeddings into contextual embeddings. Each transformer layer is composed of multi-head self-attention and feedforward layers. In the self-attention layer, input parametric embeddings are projected onto $\mathbf{K} \in \mathbb{R}^{mxk}$, $\mathbf{Q} \in \mathbb{R}^{mxk}$ and $\mathbf{V} \in \mathbb{R}^{mxv}$ matrices to generate key, query and value vectors, such that the matrices comprise the key, query and value vectors of all embeddings; k and v are the dimension of key and value matrices and m is the number of input embeddings. Each input embedding attends to all other embeddings through an attention head $Attention(\mathbf{K}, \mathbf{Q}, \mathbf{V}) = \mathbf{A} \cdot \mathbf{V}$ where attention matrix $\mathbf{A} = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{k}})$. Thus, $\mathbf{A} \in \mathbb{R}^{mxm}$ determines the degree of attention of each embedding to other embeddings. The output of the attention head is then fed to the feedforward layers. Finally, the contextual embeddings of categorical features are concatenated with $\mathbf{x}_{cont}$ and the resulting vector is fed into an MLP which predicts the target. Model parameters that are trained are ϕ for column embedding, θ for transformer layers, and weights and biases of the MLP.

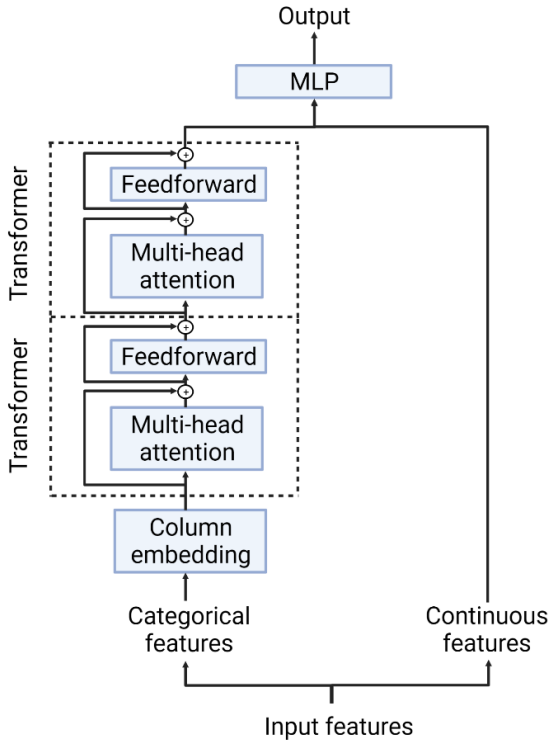

Figure 5. TabTransformer architecture. Architecture is shown for 2 transformer layers. Multi-head attention: multi-head attention layer; Feedforward: fully connected feedforward layer; MLP: multi-layer perceptron (adapted from Huang et al., 2020).

3.2.3 Self-attention and intersample attention transformer

Self-Attention and Intersample Attention Transformer (SAINT) extends the TabTransformer model with contextual embedding of all features, both categorical and continuous, by projecting all features into a high-dimensional dense vector embedding and feeding this into the transformer block (Somepalli et al., 2021). Like TabTransformer, the transformer block performs self-attention where different features within a datapoint attend to each other. However, SAINT introduces intersample attention ("row" attention) where different datapoints attend to one another, enabling superior representations and classification of datapoints. In essence, attention is performed over both rows and columns and is hierarchical, with attention computed first between features of a datapoint and then between entire datapoints. Intersample attention in SAINT is inspired by row and column attention in the Axial Transformer for

localised attention on imaging (Ho et al., 2019) and graph attention networks for attention over neighbouring nodes on graphs (Velickovic et al., 2017). SAINT has been demonstrated to perform better than XGBoost, CatBoost and LightGBM and previous deep learning methods (Somepalli et al., 2021).

The SAINT model is as follows (Somepalli et al., 2021). The input $\mathbf{x}$ is fed into an embedding layer. As feature types are heterogenous, different embedding methods are used. The embeddings are then input into L transformer layers. Each transformer layer comprises a self-attention transformer block followed by an intersample attention transformer block. The self-attention block is equivalent to the transformer layer in TabTransformer, featuring multi-head self-attention and feedforward layers. The intersample attention block is also similar but uses a multi-head intersample attention layer instead. In this layer, the embeddings for each feature in a datapoint are concatenated and attention is computed over datapoints. Attention maps of which features and datapoints attend to others provide information on which are used most prominently in decisions. The contextual embedding output of the intersample attention block is input into an MLP to predict the target.

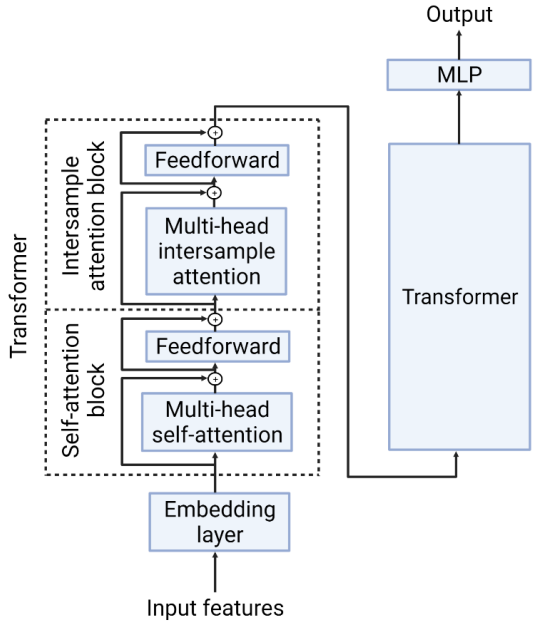

Figure 6. SAINT architecture. Architecture is shown for 2 transformer layers. Multi-head self-attention: multi-head self-attention layer; Multi-head intersample attention: multi-head intersample attention layer; Feedforward: fully connected feedforward layer; MLP: multi-layer perceptron (adapted from Somepalli et al., 2021).

3.3 Feature interaction-based models

A third family of models is based on modelling feature interactions (Cheng et al., 2016; Guo et al., 2017; Wang et al., 2017; Lian et al., 2018; Qu et al., 2016). Learning higher order feature interactions is paramount for good predictive performance, especially interactions between discrete features. This is typically done by modelling pairwise feature interactions with cross product transformation of features (Cheng et al., 2016; Guo et al., 2017; Wang et al., 2017; Lian et al., 2018; Qu et al., 2016). Modelling both low- and high-order interactions further improves performance, enabling both memorisation and generalisation. Memorisation refers to learning of feature pairs that frequently co-occur and how they relate to the target, but this cannot generalise to feature interactions that are unseen in training data. This is addressed by generalisation which explores feature pairs that are rare or have not previously occurred (Cheng et al., 2016). A major challenge for modelling feature interactions is the need to manually search for and engineer

these cross product features, which is time consuming, non-exhaustive due to the infeasibility of enumerating all cross feature combinations, and requires domain expertise as features are typically task specific. Thus, there is a need for models which learn feature interactions automatically (Cheng et al., 2016; Guo et al., 2017; Wang et al., 2017; Lian et al., 2018; Qu et al., 2016). However, there are challenges associated with the task of such models. If data comprises multi-field discrete variables, one hot encoding can lead to high-dimensional sparse inputs, on which it is difficult to learn cross features as few are observed, thereby limiting capacity of models. Additionally, models would easily overfit on high-dimensional sparse inputs. Thus, there is a need for low-dimensional dense representations of inputs (Cheng et al., 2016; Guo et al., 2017; Wang et al., 2017; Lian et al., 2018; Qu et al., 2016). A second challenge for models is the need to capture different orders of interactions, low and high for memorisation and generalisation, without overt bias towards one or the other (Cheng et al., 2016).

One such model that has been proposed for automatic learning of feature interactions is factorisation machines (FMs) (Rendle, 2010; 2012). These are suited for sparse inputs. FMs learn low-dimensional dense embeddings for sparse features, embedding each feature x_i into a latent vector $\mathbf{v}_i$, which represents its interactions with other features. Following embedding, FMs model pairwise feature interactions using the inner product of the features' latent vectors. This inner product is used to weight the cross feature i.e. $\langle \mathbf{v}_i \mathbf{v}_j \rangle x_i x_j$. Interactions between all pairs of features are modelled and the prediction of the target is: $y(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{v}_i^T \mathbf{v}_j \, x_i x_j$ where w_0 is the bias and w_i is the weight of the ith feature. A key drawback of FMs is that they only model first- and second-order feature interactions and are impractical to scale to higher order interactions, as they have polynomial time complexity with large numbers of parameters associated with significant computational cost, so they have

limited expressiveness (Guo et al., 2017). However, modelling of feature interactions is explicit which renders FMs interpretable. Another disadvantage of FMs is that they model all feature interactions with the same weight (they are not able to distinguish the importance of feature interactions). Features have different degrees of relevance and modelling interactions with less relevant features can introduce noise, to the detriment of performance (Lian et al., 2018; Blondel et al., 2016; Xiao et al., 2017).

In recent years, deep neural networks have become popular in modelling feature interactions automatically (Cheng et al., 2016; Guo et al., 2017; Wang et al., 2017; Lian et al., 2018; Qu et al., 2016). Similar to FMs, they must also use a low-dimensional dense embedding. Dense embeddings result in non-zero predictions for all pairwise feature interactions so can aid generalisation to feature pairs that are rare or unseen previously. However, as there may be no interactions between most features, over-generalisation is a drawback (Cheng et al., 2016). Deep neural networks have become popular due to their expressivity; they are able to model higher order feature interactions and capture complex and selective interactions. They can also be trained end-to-end. However, traditionally, feature interactions are learnt implicitly by deep neural networks, so lack interpretability of which feature interactions make an important contribution (Wang et al., 2017; Lian et al., 2018). Deep neural networks also do not learn certain types of feature interactions efficiently; for example, “add” operations in MLP layers are less suited to learning feature interactions in multi-field discrete data than “product” operations (Qu et al., 2016). Thus, current state-of-the-art models propose explicit feature interaction learning and different approaches to incorporating cross product features into a deep neural network.

3.3.1 Wide and Deep network

Wide and Deep networks leverage both shallow linear and deep feedforward neural network components (Cheng et al., 2016). The linear model uses cross product feature inputs while the deep neural network model embeds features into low-dimensional dense latent vectors and learns highly non-linear feature interactions between embeddings. By combining these two components, Wide and Deep inherits the advantages of each. Chiefly, the combination allows Wide and Deep to learn both low-order and high-order feature interactions, and thus achieve memorisation and generalisation, with the wide linear model memorising feature interactions through their cross features and the deep neural network generalising to rare or unseen features interactions through their low-dimensional dense embeddings. Wide and Deep has been shown to significantly improve performance compared to models with only a wide or deep component (Cheng et al., 2016).

The Wide and Deep model is as follows (Cheng et al., 2016). The wide component is a linear model: $y = \mathbf{w}^T\mathbf{x} + b$ where y is the target, $\mathbf{x}$ is the input, $\mathbf{w}$ is the weights and b is the bias. Features of $\mathbf{x}$ include both raw and cross product features. The deep component is a feedforward neural network. Sparse high-dimensional features are embedded into low-dimensional dense vectors. These embedding vectors are fed into the feedforward neural network. The computation in each hidden layer is: $\mathbf{a}^{l+1} = ReLU(\mathbf{W}^l \cdot \mathbf{a}^l + \mathbf{b}^l)$ where $\mathbf{a}$ is the activation, $\mathbf{W}$ is the weights and $\mathbf{b}$ is the bias. The wide and deep component outputs are combined in a weighted sum to make the final prediction: $P(Y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}_{wide}^T[\mathbf{x}, \phi(\mathbf{x})] + \mathbf{w}_{deep}^T\mathbf{a}^{lf} + b)$ where $\mathbf{w}_{\text{wide}}$ is the wide model weights, $\phi(\mathbf{x})$ is the cross product transformation of the original features, $\mathbf{w}_{\text{deep}}$ is the weights applied on the final activations of the deep neural network and b is the bias. Model parameters that are jointly trained are therefore, $\mathbf{w}$ and b in the linear model, $\mathbf{W}$ and $\mathbf{b}$ in the deep neural network and $\mathbf{w}_{\text{wide}}$, $\mathbf{w}_{\text{deep}}$ and b in the final weighted sum.

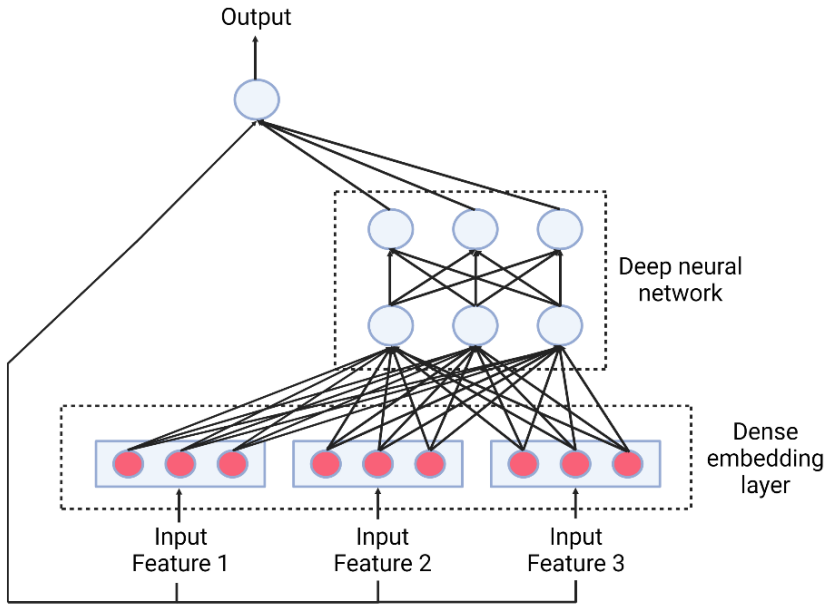

Figure 7. Wide and Deep architecture. Architecture is shown for 3 input features (adapted from Cheng et al., 2016).

3.3.2 Deep Factorisation Machine

Deep Factorisation Machine (DeepFM) combines shallow FM and deep neural network components (Guo et al., 2017). These use the same input and feature embedding vector. This avoids the need for manual feature engineering, as learning can occur directly from the raw input features. Like Wide and Deep, DeepFM is also able to learn both low- and high-order feature interactions, with low-order interactions modelled by the FM component and high-order interactions modelled by the deep neural network component. This enables both memorisation and generalisation. DeepFM has been shown to systematically improve performance over other state-of-the-art models and models with only a FM or deep component (Guo et al., 2017).

The DeepFM model is as follows (Guo et al., 2017). For each feature x_i, a scalar w_i is the weight representing the importance of the feature and a learnable latent embedding vector $\mathbf{v}_i$ represents its interactions with other features. In the FM, the inner product of latent vectors is used to weight feature interactions. The output of the FM is $y_{FM}(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + \sum_{i=1}^{d} \sum_{j=i+1}^{d} \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i \cdot x_j$. In the deep neural network, an embedding layer converts the features into low-dimensional dense vectors. The weights for this embedding layer are the latent vectors $\mathbf{v}_i$ used in the FM. The output of the embedding layer is then fed into the feedforward neural network which uses the following computation in each layer: $\mathbf{a}^{l+1} = \sigma(\mathbf{W}^l \cdot \mathbf{a}^l + \mathbf{b}^l)$, where $\mathbf{a}$ is the activation, $\mathbf{W}$ is the weights and $\mathbf{b}$ is the bias. The final prediction combining the FM and deep component is $y(\mathbf{x}) = \sigma(y_{FM}(\mathbf{x}) + y_{Deep}(\mathbf{x}))$ where $y_{FM}(\mathbf{x})$ is the output of FM component and $y_{Deep}(\mathbf{x})$ is the output of deep component. Model parameters that are jointly trained are $\mathbf{w}$ and $\mathbf{v}_i$ in the FM and $\mathbf{W}$ and $\mathbf{b}$ in the deep neural network.

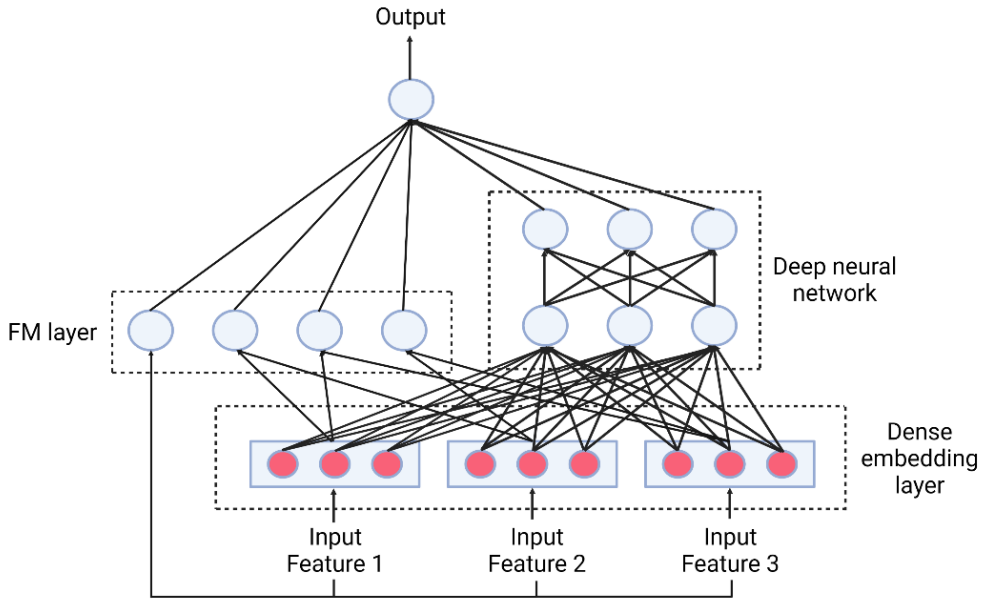

Figure 8. DeepFM architecture. Architecture is shown for 3 input features (adapted from Guo et al., 2017).

3.3.3 Deep and Cross network

Deep and Cross network (DCN) jointly trains a cross network and deep feedforward neural network (Wang et al., 2017). The cross network has the function of performing feature crossing, and does this by taking the outer product of feature embedding vectors at the bit-wise level. The cross network models all cross features up to a degree that is determined by the layer depth as each layer learns higher order interactions based on the interactions from previous layers, so it allows efficient learning of bounded-degree feature interactions. The cross network is motivated by FMs, extending it from a single layer to a deep structure which can model high-degree cross features and higher order interactions. It also avoids the need for manual cross feature engineering. By leveraging both the cross and deep neural networks, Deep and Cross enjoys the benefits of both. The cross network can learn some types of feature interaction that deep neural networks are unable to, and models feature interactions explicitly, but deep neural networks can learn higher order non-linear interactions which the cross network does not have capacity for. Deep and Cross has been shown to have state-of-the-art performance superior to other models, on both sparse and dense inputs (Wang et al., 2017).

The DCN model is as follows (Wang et al., 2017). DCN begins with an embedding layer transforming features into low-dimensional dense vectors. This embedding layer uses a learnable embedding matrix of weights. The embedding vectors then are stacked into one vector, $\mathbf{x}_0$, and fed into the cross network and deep neural network in parallel. Each cross layer performs feature crossing on its input (which is the output of the previous cross layer) and $\mathbf{x}_0$, and adds back its input after feature crossing: $\mathbf{x}_{l+1} = \mathbf{x}_0\mathbf{x}_l^T\mathbf{w}_l + \mathbf{b}_l + \mathbf{x}_l$ where $\mathbf{x}_l$ is the output from the *lth* cross layers and $\mathbf{w}$ and $\mathbf{b}$ are the weight and bias of the *lth* layer. As each layer crosses the output of the previous layer with $\mathbf{x}_0$, the degree of the cross features is proportional to the layer depth. The weight $\mathbf{w}_l$ of the cross feature $\mathbf{x}_0\mathbf{x}_l^T$ is the multiplication of weights w_i

associated with each x_i. The deep neural network is a fully connected feedforward network where the computation in each layer is: $\mathbf{a}^{l+1} = ReLU(\mathbf{W}^l \cdot \mathbf{a}^l + \mathbf{b}^l)$, where $\mathbf{a}$ is the activation, $\mathbf{W}$ is the weights and $\mathbf{b}$ is the bias. The final combination layer combines the outputs from the cross and deep neural networks: $p = \sigma([\mathbf{x}_{L1}^T, \mathbf{a}_{L2}^T]\mathbf{w}_{logits})$ where $\mathbf{x}_{L1}$ is the output from the cross network, $\mathbf{a}_{L2}$ is the output from the deep network and $\mathbf{w}_{logits}$ is the weight vector for the combination layer. Model parameters that are jointly trained are $\mathbf{w}_l$ and $\mathbf{b}_l$ in the cross network, $\mathbf{W}$ and $\mathbf{b}$ in the deep neural network and $\mathbf{w}_{logits}$ in the final combination layer.

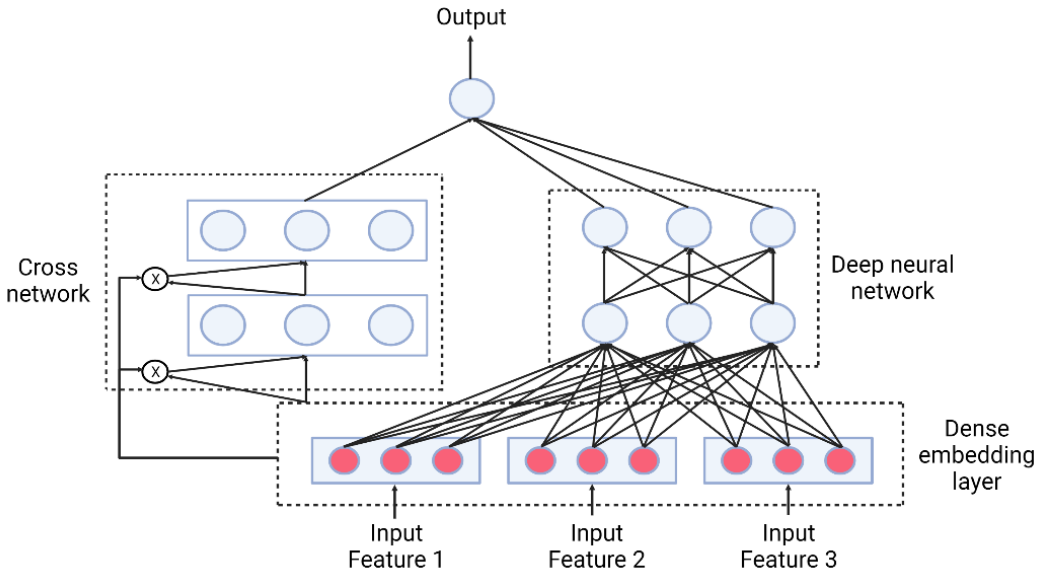

Figure 9. DCN architecture. Architecture is shown for 3 input features (adapted from Wang et al., 2017).

3.3.4 Extreme Deep Factorisation Machine

eXtreme Deep Factorization Machine (xDeepFM) combines a deep neural network and Compressed Interaction Network (CIN) module (Lian et al., 2018). The CIN performs feature crossing by taking the outer product of a hidden layer and the original feature matrix at vector-wise level. Like the cross network of Deep and Cross, the degree of interactions is determined

by the CIN layer depth so CIN learns bounded-degree feature interactions. CIN is based on and extends the cross network of Deep and Cross, and also draws on the architecture of CNNs and RNNs. It also avoids the need for manual feature searching or engineering. CIN and deep neural networks have distinct properties that are complementary to one another. By jointly training them, xDeepFM can learn both low- and high-order feature interactions and both implicit and explicit feature interactions: CIN learns explicit high-order interactions while the deep neural network learns high- and low-order interactions implicitly. xDeepFM has been shown to consistently outperform a number of other state-of-the-art models on real-world datasets (Lian et al., 2018).

The xDeepFM model is as follows (Lian et al., 2018). xDeepFM uses an embedding layer to convert raw features into a low-dimensional dense vector. The embedding layer outputs a matrix $\mathbf{X}^0 \in \mathbb{R}^{mxD}$ where the *ith* row is the embedding vector of the *ith* feature, m is the number of embedding vectors and D is the dimension of the embedding vectors. $\mathbf{X}^0$ is fed into the CIN network. The computation in a CIN layer is: $\mathbf{X}_{h,*}^k = \sum_{i=1}^{H_{k-1}} \sum_{j=1}^{m} \mathbf{W}_{ij}^{k,h} (\mathbf{X}_{i,*}^{k-1} \circ \mathbf{X}_{j,*}^0)$ where H_k is the number of embedding vectors in the *kth* layer, $\mathbf{X}^k$ is the output of the *kth* layer and $\mathbf{W}^{k,h}$ is the parameter matrix for the *hth* embedding vector in the *kth* layer. Each layer takes the interaction of the output of the previous layer $\mathbf{X}^{k-1}$ and $\mathbf{X}^0$, so the degree of interactions scales with the layer depth. The outputs of all CIN layers are pooled, by sum pooling on each $\mathbf{X}^k$, $p_i^k = \sum_{j=1}^{D} \mathbf{X}_{i,j}^k$, concatenation into a pooling vector for each layer $\mathbf{p}^k = [p_1^k, p_2^k, \dots, p_{H_k}^k]$ and finally concatenation of pooling vectors of all K hidden layers into $\mathbf{p}^+ = [\mathbf{p}^1, \mathbf{p}^2, \dots, \mathbf{p}^K]$. The final prediction is made by combining CIN and deep neural network outputs: $y = \sigma(\mathbf{w}_{linear}^T \mathbf{x} + \mathbf{w}_{dnn}^T \mathbf{a}_{dnn}^k + \mathbf{w}_{cin}^T \mathbf{p}^+ + b)$ where $\mathbf{x}$ is the raw features, $\mathbf{a}_{dnn}^k$ is the output of the deep neural network, $\mathbf{p}^+$ is the output of the CIN, $\mathbf{w}_{linear}^T$ is the weight for the raw features, $\mathbf{w}_{dnn}^T$ is the weight for the deep neural network output, $\mathbf{w}_{cin}^T$ is the weight for the CIN output and b is the

bias. Thus model parameters jointly trained are $\mathbf{W}_{ij}^{k,h}$ in the CIN, the weights and biases of the deep neural network, and $\mathbf{w}_{linear}^{T}$, $\mathbf{w}_{dnn}^{T}$, $\mathbf{w}_{cin}^{T}$ and b of the final output unit.

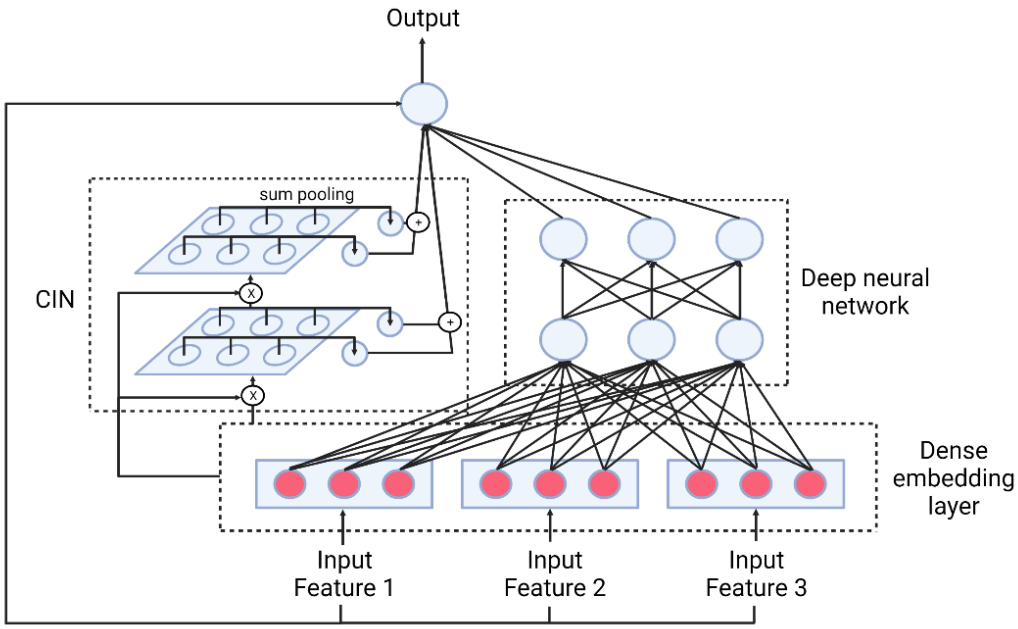

Figure 10. xDeepFM architecture. Architecture is shown for 3 input features (adapted from Lian et al., 2018).

3.3.5 Product-based neural network

Product-based neural networks (PNNs) introduce a product layer to model inter-field feature interactions, followed by fully connected MLP layers to learn higher order feature interactions (Qu et al., 2016). The product layer uses both inner and outer product operations (it is a concatenation of inner and outer products) which provides a strategy for rule representation. PNNs largely capture high-order feature interactions, as they utilise a deep feedforward neural network. They have been shown to offer consistently superior performance to other state-of-the-art models (Qu et al., 2016).

The PNN model is as follows (Qu et al., 2016). PNNs use a layer to embed features and generate low-dimensional dense embedding vectors. The embedding layer uses the weight matrix $\mathbf{W}_i$ for feature x_i. In the inner product neural network, pairwise feature interaction is computed as the inner product of the embedding vectors of features: $g(\mathbf{v}_i, \mathbf{v}_j) = \langle \mathbf{v}_i, \mathbf{v}_j \rangle$ and these terms are collated into a square matrix $\mathbf{p}$. The product layer also generates linear $\mathbf{z} = (\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_N)$. Once $\mathbf{p}$ and $\mathbf{z}$ are obtained, $\mathbf{l}_p$ is calculated as $\mathbf{l}_p = (l_p^1, l_p^2, \dots, l_p^{D_1})$ where $l_p^n = \mathbf{W}_p^n \odot \mathbf{p} = \sum_{i=1}^{N} \sum_{j=1}^{N} (\mathbf{W}_p^n)_{i,j} \mathbf{p}_{i,j}$. $\mathbf{l}_z$ is calculated as $\mathbf{l}_z = (l_z^1, l_z^2, \dots, l_z^{D_1})$ where $l_z^n = \mathbf{W}_z^n \odot \mathbf{z} = \sum_{i=1}^{N} \sum_{j=1}^{M} (\mathbf{W}_z^n)_{i,j} \mathbf{z}_{i,j}$. In these computations, $\mathbf{W}_p^n$ and $\mathbf{W}_z^n$ are weights in the product layer. The outer product neural network uses similar computations to the inner product, except that feature interaction is computed as the outer product of embedding vectors of features: $g(\mathbf{v}_i, \mathbf{v}_j) = \mathbf{v}_i \mathbf{v}_j^T$. Thus for each element of $\mathbf{p}$, $\mathbf{p}_{i,j}$ is a square matrix. $\mathbf{p}$ is then defined as $\mathbf{p} = \sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{v}_i \mathbf{v}_j^T$. The product layer connects to the deep neural network. The inputs are $\mathbf{l}_p$ and $\mathbf{l}_z$ and computation in the first hidden layer is: $\mathbf{l}_1 = ReLU(\mathbf{l}_p + \mathbf{l}_z + \mathbf{b}_1)$ where $\mathbf{b}_1$ is the bias. In the second hidden layer $\mathbf{l}_2 = ReLU(\mathbf{W}_2 \mathbf{l}_1 + \mathbf{b}_2)$ and the final prediction is made by: $y = \sigma(\mathbf{W}_3 \mathbf{l}_2 + \mathbf{b}_3)$, where $\mathbf{W}$ and $\mathbf{b}$ are the weights and biases. Model parameters that are trained are $\mathbf{W}_p^n$ and $\mathbf{W}_z^n$ in the product layers, and $\mathbf{W}$ and $\mathbf{b}$ of the deep neural network.

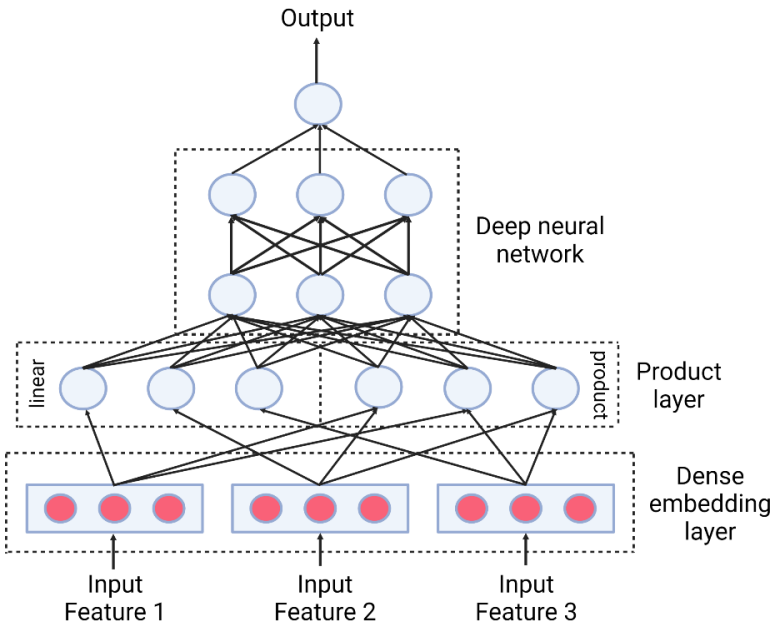

Figure 11. PNN architecture. Architecture is shown for 3 input features (adapted from Qu et al., 2016).

3.4 Regularisation-based models

A new school of thought is gaining traction in deep learning for predictive tasks on tabular data, which is to apply more sophisticated regularisation. This has come in two forms – use of recently developed deep learning regularisation techniques in combination and specialised regularisation-based architectures (Kadra et al., 2021; Shavitt & Segal, 2018; Lounici et al., 2021; Klambauer et al., 2017).

3.4.1 Deep learning regularisation techniques

The concept of regularisation cocktails – that is the joint and simultaneous application of a set of deep learning regularisation techniques from different families – has been advanced (Kadra et al., 2021). Regularisation cocktails have been shown to significantly improve the performance of simple MLPs on tabular data prediction tasks; their use can surpass the performance of state-

of-the-art specialised architectures enumerated in this chapter thus far, as well as the most successful GBDT methods such as XGBoost.

Three families of regularisers, in particular, are important and the most widely used: weight decay (either L_1 or L_2), which is the classical regularisation approach based on penalising the size (norms) of weights (Tibshirani, 1996; Tikhonov, 1943); model averaging, predominantly dropout (Srivastava et al., 2014) and, more recently, snapshot ensembles (Huang et al., 2017); and implicit regularisation, which includes batch normalisation (Ioffe & Szegedy, 2015) and, more recently, stochastic weight averaging (Izmailov et al., 2018) and Lookahead optimiser (Zhang et al., 2019). The following section is an in-depth exploration of the latter 5 regularisation techniques developed in recent years for deep neural network architectures.

3.4.1.1 Model averaging

Deep neural networks can learn highly non-linear relationships between input and output and have the potential to be extremely expressive. However, this can lead to overfitting, especially with limited data. Combining the predictions of ensembles of models trained with different initialisations and parameter settings often significantly increases robustness and improves performance, but for large neural networks, averaging outputs during testing or training multiple neural networks for the purpose of model averaging can be computationally expensive, time consuming and be prohibitively costly. Additionally, combining predictions of models relies on models being sufficiently different; training neural networks of different architectures or on different training datasets is difficult owing to time needed to optimise hyperparameters and insufficient data (Srivastava et al., 2014; Huang et al., 2017).

3.4.1.1.1 Dropout

The core idea of dropout is to randomly and independently drop units and their connections in a neural network with a fixed probability during training (Srivastava et al., 2014). This is equivalent to randomly sampling a thinned network (which consists of all units retained after dropout) from 2^n possible sub-networks where n is the number of units in the network, with thinned networks sharing weights. During testing, it is infeasible to average prediction from many thinned networks. Instead a single unthinned neural network, without dropout, is used with the weights of the network equal to weights from training scaled down (the trained weights of a unit is multiplied by the probability it had of being retained during training). This represents an efficient technique for attaining a weighted mean of predictions of an exponential number of networks. Dropout improves generalisation and reduces overfitting. This is because it prevents co-adaptations of units in neural networks as its stochasticity renders the presence of any individual unit unreliable. Alternatively, dropout may be effective as it ensembles many different networks. It has been shown to be applicable to diverse domains and produce state-of-the-art results across a wide variety of tasks in vision, speech and text recognition, significantly outperforming classical neural networks (Srivastava et al., 2014).

Dropout is formulated as follows (Srivastava et al., 2014). The computation within a feedforward neural network is $\mathbf{z}^{l+1} = \mathbf{W}^{l+1}\mathbf{a}^{l} + \mathbf{b}^{l+1}$ and $\mathbf{a}^{l+1} = f(\mathbf{z}^{l+1})$ where $\mathbf{z}$ is the preactivation, $\mathbf{a}$ is the activation, $\mathbf{W}$ is the weights, $\mathbf{b}$ is the bias and f is an activation function. Dropout modifies activation $\mathbf{a}$ by: $r_j^l \sim \text{Bernoulli}(p)$, such that $\mathbf{r}^l$ is a vector of independent Bernoulli random variables each with probability p of being 1, and $\tilde{\mathbf{a}}^l = \mathbf{r}^l * \mathbf{a}^l$, such that $\tilde{\mathbf{a}}^l$ is the thinned output and is used instead of $\mathbf{a}^l$ as the input into the next layer of the feedforward neural network. During training, backpropagation of the derivatives of the loss is only done on

the thinned subnetwork. During testing, weights of the neural network without dropout are scaled as $\mathbf{W}_{test}^{l} = p\mathbf{W}^{l}$.

3.4.1.1.2 Snapshot ensembles

The motivation of snapshot ensembling (SE) is to learn and combine multiple neural network models, without any additional training cost (Huang et al., 2017). This makes use of the non-convex nature of training a deep neural network, with many local minima. These minima are diverse and contain unique information; while different local minima may be associated with the same degree of error, they are non-overlapping in the errors they make so there is a utility in ensembling models at different local minima. The core idea of SE is to train a neural network to converge at multiple intermediate local minima along its gradient-based optimisation path, leveraging the ability of SGD to readily converge and escape from local minima. Each time the model converges, the network weights are saved (a "snapshot" is taken). For ensembling to be effective, individual snapshots must all have low error, hence the model must be trained quickly in few epochs if the total training time is to remain unchanged. To achieve rapid convergence, a cyclic learning rate schedule following a cosine function is used, where the learning rate is quickly lowered to encourage model convergence to a local minima and then raised after a snapshot is taken to perturb the model to escape the local minima, giving it the opportunity to converge towards another minima (Loshchilov & Hutter, 2017); strong perturbation is important to increase the diversity of local minima. During testing, the ensemble prediction is the average of the predictions of the last few snapshots as increasingly better models with lower error are obtained over the course of training. The SE approach has been shown to significantly improve state-of-the-art performance and is comparable to other ensemble models (Huang et al., 2017). It is a technique that is network architecture and task agnostic.

SE is formulated as follows (Huang et al., 2017). The cyclic cosine annealing learning rate schedule has the form $\alpha(t) = \frac{\alpha_0}{2}\left(\cos\frac{\pi\, mod(t-1,\lfloor T/M \rfloor)}{\lfloor T/M \rfloor} + 1\right)$ where α is the learning rate, α_0 is the initial learning rate, t is the epoch number, T is the total number of epochs and M is the number of snapshots. A large initial learning rate is annealed to a smaller learning rate which is close to 0 over the course of a cycle, with the large learning rate allowing the model to escape from local minima and the small learning rate allowing the model to converge to local minima.

3.4.1.2 Implicit

3.4.1.2.1 Batch normalisation

The motivation for batch normalisation (BN) is internal covariate shift which refers to the change in distributions of layer inputs during training of deep neural networks as the inputs are affected by the parameters of previous layers which change as they are updated (Shimodaira, 2000). This especially occurs in deep neural networks as small changes in parameters can become amplified through many layers. It poses a problem for the efficient training of neural networks as weight parameters of the layers have to continually adapt to compensate for changes in the distribution of the input. Additionally, through many layers, inputs can drift into saturated regions of activation functions resulting in vanishing gradients and slow convergence. To counteract these effects, ReLU activation is used, weights must be initialised carefully and small learning rates must be used (as large learning rates can lead to large parameters, exploding gradients and divergence) (Glorot & Bengio, 2010; Saxe et al., 2013). BN addresses these issues by normalising layer inputs (activations of previous layers) – that is, fixing the mean and variance of these to zero mean and unit variance, in order to mitigate internal covariate shift (Ioffe & Szegedy, 2015). It does this by applying a differentiable transformation on inputs, which is incorporated into the network architecture so that gradient-based optimisation methods can be still be applied. It uses additional parameters to scale the transformed inputs, to maintain

the representation ability of the network. Normalisation is applied over mini-batches as application across the entire training set for every update of weights would be computationally expensive. This also allows SGD to be used. By maintaining a stable distribution for inputs, BN significantly accelerates convergence and training speed. BN also improves gradient propagation behaviour, avoiding saturation and vanishing gradient, and makes it much less dependent on the scale of parameters or their initialisation (it can be shown that scale does not affect the Jacobian of layers). This allows use of less well-tuned initialisation, higher learning rates and saturating non-linear activation functions without divergence, which also contributes to faster convergence. BN additionally has a regularisation effect, as datapoints are seen in conjunction with others in the mini-batch and the network produces non-deterministic values for a datapoint, which aids generalisation. BN has been shown to significant improve state-of-the-art performance (Ioffe & Szegedy, 2015).

BN is formulated as follows (Ioffe & Szegedy, 2015). Each feature x^k is normalised independently to zero mean and unit variance: $\hat{x}^k = \frac{x^k - \mathbb{E}[x^k]}{\sqrt{Var[x^k]}}$. However, normalising the inputs can change the representation in the layer. To compensate, the normalised value undergoes an affine transformation $y^k = \gamma^k \hat{x}^k + \beta^k$ using learnable parameters γ^k and β^k, so that the BN transform can be an identity transform which preserves the representation ability of the network. Computation of mean and variance of each activation is done on mini-batches of size m. Hence the overall BN transform has the following steps:

1. Computing the mini-batch mean: $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$
2. Computing the mini batch variance: $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
3. Normalisation: $\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$

4. Affine transformation: $y_i = \gamma \hat{x}_i + \beta$.

The BN transform is used in network layers. The computation within a network unit is typically $\mathbf{a} = f(\mathbf{W}\mathbf{z} + \mathbf{b})$ where **a** is the activation, **z** is the pre-activation, f is the non-linear activation function and **W** and **b** are weights and biases of the model. With the BN transform this is: $\mathbf{a} = f(BN(\mathbf{W}\mathbf{z}))$ and $\hat{\mathbf{a}} = \gamma\mathbf{a} + \beta$. During backpropagation in training, the gradients of the loss with respect to the BN transform parameters are computed in addition to other derivatives: $\frac{\partial \ell}{\partial \hat{x}_i}$, $\frac{\partial \ell}{\partial \mu_B}$, $\frac{\partial \ell}{\partial \sigma_B^2}$, $\frac{\partial \ell}{\partial x_i}$, $\frac{\partial \ell}{\partial \gamma}$ and $\frac{\partial \ell}{\partial \beta}$. During testing, activations are not normalised – the following steps are applied:

1. $\mathbb{E}[x] = \mathbb{E}_B[\mu_B]$
2. $Var[x] = \frac{m}{m-1}\mathbb{E}_B[\sigma_B^2]$
3. $y = \frac{\gamma}{\sqrt{Var[x] + \varepsilon}} \cdot x + (\beta - \frac{\gamma \mathbb{E}[x]}{\sqrt{Var[x] + \varepsilon}})$.

3.4.1.2.2 Stochastic weight averaging

The motivation of stochastic weight averaging (SWA) is that SGD traverses a weight space for neural networks that correspond to high performance but does not typically reach the optimal points in this weight space. SWA is a method to compute an average of the weights attained along the sequence of gradient-based optimisation steps, in order to move to the optimal points in the weight space (Izmailov et al., 2018). These averaged weights are then used for the network. SWA finds wider optimal solutions than conventional SGD which results in better generalisation (this is because the training and test error surfaces do not exactly align but wider optima would be more likely to remain optimal in both as it is robust to the small shift between the surfaces). Conventional SGD converges to optima near the boundary of wide regions of optimal weights and, with asymmetry of the loss function, SGD typically converges near the

periphery of sharp ascents in the training loss. In contrast, SWA converges in the centre of optimal regions as well as in flat region of the training loss. These convergence points are more robust as larger perturbations are needed to increase error. Again, because of the shift between the train and test error surfaces, centred points, although are associated with slightly worse training error, result in better test error and better generalisation. SWA has been shown to substantially improve performance of state-of-the-art models on a range of datasets and it can be used with a host of different network architectures (Izmailov et al., 2018).

SWA is formulated as follows (Izmailov et al., 2018). The cyclical learning rate schedule used is: $\alpha(i) = \big(1 - t(i)\big)\alpha_1 + t(i)\alpha_2$ where $t(i) = \frac{1}{c}\,(mod(i-1, c) + 1)$, α_1 and α_2 are learning rates and c is the averaging period. This has the effect of decreasing the learning rate from α_1 to α_2 linearly.

3.4.1.2.3 Lookahead

Lookahead is a novel form of optimisation algorithm, that updates two sets of weights: it generates a sequence of "fast weights" using a conventional optimiser such as SGD or Adam in its inner loop and then uses the final fast weights to inform the direction of update of "slow weights" in its outer loop (Zhang et al., 2019). This method essentially performs weight averaging during training, as opposed to SWA which performs it at the end. Lookahead reduces variance (as higher variance in the inner loop updates is reset by the outer loop update), improves learning stability and accelerates convergence (Lookahead uses a large learning rate in the inner loop to allow fast weights to make quick progress in learning but uses slow weights to smooth out oscillations for rapid convergence, avoiding the pitfalls of conventional optimisation methods which can oscillate and be slow to converge). It also improves generalisation.

Lookahead has been shown to significantly improve results on a range of architectures and datasets (Zhang et al., 2019). It is applicable to any standard optimisation method.

Lookahead is formulated as follows (Zhang et al., 2019). Slow weights ϕ and fast weights θ are maintained. Fast weights in the inner loop are updated by: $\theta_{t,i+1} = \theta_{t,i} + A(L, \theta_{t,i-1}, d)$ where A is the standard optimisation method selected, L is the objective function and d is the mini-batch of training data. This update is iterated k times, following which the slow weights are updated using the rule: $\phi_{t+1} = \phi_t + \alpha(\theta_{t,k} - \phi_t) = \alpha[\theta_{t,k} + (1-\alpha)\theta_{t-1,k} + \cdots + (1-\alpha)^{t-1}\theta_{0,k}] + (1-\alpha)^t\phi_0$, where α is the slow weights step size. In essence, the slow weights are updated by linear interpolation with fast weights, using an exponential moving average of fast weights that places greater focus on the later fast weights in the k updates but with some contribution from earlier fast weights. After slow weights are updated, fast weights are reset to the slow weight value.

3.4.2 Regularisation-based architectures

The second form of regularised models that are actively being explored for deep learning are specialised regularisation-based architectures (Shavitt & Segal, 2018; Lounici et al., 2021; Klambauer et al., 2017). The following section reviews the key approaches.

3.4.2.1 Regularisation learning networks

The core idea of regularisation learning networks (RLNs) is to apply a different learnable regularisation coefficient to each weight in a neural network (Shavitt & Segal, 2018). This would typically result in a large number of regularisation coefficients as hyperparameters which

is not computationally tractable. RLNs introduce a new loss function – counterfactual loss – to efficiently optimise these many regularisation coefficient hyperparameters and learn them together with the weights of the neural network. The effect of RLNs is sparse feature selection, with selected features assigned with large weights, thereby retaining only the most informative features and conferring the correct inductive bias. Additionally, RLNs result in modular regularisation: if certain irrelevant features are strongly regularised, other more relevant features undergo more relaxed regularisation. Thus, RLNs are most suited to input data with highly variable relative feature importance. As they provide meaningful insights into which features are important for prediction, they also confer model interpretability. RLNs have been shown to significantly outperform deep neural networks using other regularisation techniques and perform comparably with GBDTs (Shavitt & Segal, 2018).

The RLN model is as follows (Shavitt & Segal, 2018). In conventional regularisation, the objective function is: $\tilde{L}(\mathbf{X}, \mathbf{W}, \lambda) = L(\mathbf{X}, \mathbf{W}) + \exp(\lambda) \cdot \sum_{i=1}^{n} \|w_i\|$ where L is the loss function, $\mathbf{X}$ is the data, $\mathbf{W}$ is the weights and λ is the regularisation coefficient. In this case, the regularisation coefficient λ is a hyperparameter tuned with a validation dataset. For RLNs, a different regularisation coefficient is associated with each weight in the neural network so the objective function is: $\tilde{L}(\mathbf{X}, \mathbf{W}, \Lambda) = L(\mathbf{X}, \mathbf{W}) + \sum_{i=1}^{n} \exp(\lambda_i) \cdot \|w_i\|$ where $\Lambda = \{\lambda_i\}_{i=1}^{n}$ are the set of regularisation coefficients. There are therefore n regularisation coefficients λ_i, one for each weight w_i, and optimising using cross validation is intractable. The goal is therefore to find an efficient way to optimise Λ. The SGD update for weights is: $w_{t+1,i} = w_{t,i} - \eta \cdot \frac{\partial \tilde{L}(\mathbf{X}_t, \mathbf{W}_t, \Lambda_t)}{\partial w_{t,i}}$ where η is the learning rate of the weights. This can be written as $w_{t+1,i} = w_{t,i} - \eta \cdot (g_{t,i} + r_{t.i})$ where $g_{t,i}$ is the gradient of the loss term, $g_{t,i} = \frac{\partial \mathrm{L}(\mathbf{X}_t, \mathbf{W}_t)}{\partial w_{t,i}}$, and $r_{t,i}$ is the gradient of the regularisation term, $r_{t.i} = \frac{\partial}{\partial w_{t,i}} (\sum_{j=1}^{n} \exp(\lambda_{t,j}) \cdot \|w_{t,j}\|)$. As the latter is zero for every j ≠ i, it

is equivalent to $\exp(\lambda_{t,i}) \cdot \frac{\partial \|w_{t,i}\|}{\partial w_{t,i}}$. The counterfactual loss is then formulated as $L_{CF}(\mathbf{X}_t, \mathbf{X}_{t+1}, \mathbf{W}_t, \Lambda_t, \eta) = L(\mathbf{X}_{t+1}, \mathbf{W}_{t+1})$ where $\mathbf{W}_{t+1}$ depends on $\mathbf{W}_t$, $\mathbf{X}_t$, Λ_t and η as the update rule for weights is $w_{t+1,i} = w_{t,i} - \eta \cdot \frac{\partial \tilde{L}(\mathbf{X}_t, \mathbf{W}_t, \Lambda_t)}{\partial w_{t,i}}$. L_{CF} can then be used in the SGD update for regularisation coefficients: $\lambda_{t+1,i} = \lambda_{t,i} - \nu \cdot \frac{\partial L_{CF}(\mathbf{X}_t, \mathbf{X}_{t+1}, \mathbf{W}_t, \Lambda_t, \eta)}{\partial \lambda_{t,i}} = \lambda_{t,i} + \nu \cdot \eta \cdot g_{t+1,i} \cdot r_{t.i}$ where ν is the learning rate of the regularisation coefficients. Thus, the counterfactual loss allows optimisation of regularisation coefficients while learning the weights of the network simultaneously. With many training steps, regularisation coefficients $\lambda_{t,i}$ tends to vanish, hence a projection step is used with each update of coefficients to normalise the coefficients: $\lambda_{t+1,i} = \lambda_{t,i} + \nu \cdot \eta \cdot g_{t+1,i} \cdot r_{t.i} + (\theta - \frac{\sum_{j=1}^{n} \lambda_{t,i} + \nu \cdot \eta \cdot g_{t+1,i} \cdot r_{t.i}}{n})$ where θ is the normalisation factor of the regularisation coefficients. This leads to regularisation in the network having zero sum game behaviour, with stronger regularisation in one part of the network permitting relaxation in regularisation in other parts. Model parameters that are trained are $\mathbf{W}$ and Λ.

3.4.2.2 Muddling labels for regularisation

The core idea of muddling labels for regularisation (MLR) is penalising memorisation (and thus promoting generalisation) of deep neural networks with a new loss function that incorporates three techniques: Ridge regularisation on the output of the last hidden layer of the neural network and using this to replace the weights of the output layer, random permutations to generate target labels which are not informative, and dithering to introduce noise to target labels (Lounici et al., 2021). The first technique means the weights of the output layer are directly a function of the last hidden layer, leading to a stronger regularisation than simply regularising the weights of the output layer, as is conventional. The second technique prevents overfitting of the model as random permutation of target labels will produce false (x, y) pairs in which it is expected there should be no relationship between x and y; fitting of any relationship would only

occur through memorisation. This is penalised which compels the model to focus on meaningful relationships between x and y in real (x, y) pairs. MLR has been shown to produce higher performance than simple neural networks and tree-based methods on datasets with a range of sample sizes, feature types, tasks i.e. classification and regression and task difficulties (Lounici et al., 2021).

The MLR model is as follows (Lounici et al., 2021). The base structure is a simple feedforward neural network with the following computations in each layer: $\mathbf{A}^0 = \mathbf{x}$ in the input layer, $\mathbf{A}^1 = ReLU(\mathbf{A}^0\mathbf{W}^1 + \mathbf{b}^1)$ in the first hidden layer, $\mathbf{A}^{l+1} = ReLU(\mathbf{A}^l\mathbf{W}^{l+1} + \mathbf{b}^{l+1})$ in subsequent hidden layers, and $\mathbf{A}^L = \mathbf{A}^{L-1}\mathbf{W}^L$ in the output layer, where $\mathbf{A}$ is the activation, $\mathbf{W}$ is the weights and $\mathbf{b}$ is the bias. MLR makes certain modifications. Firstly, ridge regularisation is applied to the output of the final hidden layer $\mathbf{A}^{L\text{-}1}$: $\mathbf{P}(\theta, \lambda, \mathbf{x}) = [(\mathbf{A}^{L-1})^T\mathbf{A}^{L-1} + \lambda\mathbb{I}]^{-1}(\mathbf{A}^{L-1})^T$, where θ represents the learnable weight and bias parameters of the neural network, λ is the learnable regularisation coefficient and $\mathbf{x}$ is the input. $\mathbf{H}$, a "regularised projector" based on $\mathbf{A}^{L\text{-}1}$, is computed as $\mathbf{H}(\theta, \lambda, \mathbf{x}) = \mathbf{A}^{L-1}\mathbf{P}(\theta, \lambda, \mathbf{x})$. The output layer of the neural network is computed as $\mathbf{A}^L = \mathbf{A}^{L-1}\mathbf{W}^L = \mathbf{A}^{L-1}\,\mathbf{P}(\theta, \lambda, \mathbf{x})\mathbf{Y} = \mathbf{H}(\theta, \lambda, \mathbf{x})\mathbf{Y}$ and the final output is $Hardmax(\mathbf{A}^{L-1}(\theta)\mathbf{P}(\theta, \lambda, \mathbf{x})\mathbf{Y})$, where $\mathbf{Y}$ are the target labels. Secondly, permutation operations are applied to target labels $\mathbf{Y}$. Thirdly, dithering is applied by muddling the target through introducing noise of higher variance along less informative eigendirections of $\mathbf{H}$. The MLR loss is therefore: $MLR(\theta, \lambda) = BCE(\mathbf{Y}; (2\mathbf{Y} - 1) + (\mathbb{I}_n - \mathbf{H})\xi + \mathbf{H}(2\mathbf{Y} - 1)) + \frac{1}{T}\sum_{t=1}^{T}|BCE(\mathbf{Y}; \overline{\mathbf{Y}}\mathbb{1}_n) - BCE(\pi^t(2\mathbf{Y} - 1); \pi^t(2\mathbf{Y} - 1) + (\mathbb{I}_n - \mathbf{H})\xi_t + \mathbf{H}\pi^t(2\mathbf{Y} - 1))|$ where $\overline{\mathbf{Y}} = mean(\mathbf{Y})$, $(\pi^t(2\mathbf{Y} - 1))_{t=1}^T$ are T randomly drawn permutations of $2\mathbf{Y} - 1$, and ξ and $(\xi_t)_{t=1}^T$ are randomly drawn noise vectors from $\mathcal{N}(0_n, \mathbb{I})$ which adds noise to $2\mathbf{Y} - 1$ and permutations of $2\mathbf{Y} - 1$. The first term of the MLR loss represents the conventional BCE loss and the second represents the amount of memorisation of the model by comparing the BCE loss

when fitting to uninformative labels to the BCE loss when not fitting the data. Model parameters that are trained are θ and λ.

3.4.2.3 Self-normalising neural networks

BN has become extremely popular in deep neural networks (Ioffe & Szegedy, 2015). It normalises activations in neural network layers to zero mean and unit variance but these can be perturbed through the process of SGD and regularisation such as dropout. If variance grows or diminishes excessively, this can lead to exploding and vanishing gradients respectively during training. In general, perturbations to mean and variance can lead to high fluctuations in training error and reduce the efficacy and speed of learning. The core idea of self-normalising neural networks (SNNs) is to obviate the need for BN and instead use an alternative activation function – the scaled exponential linear unit (SELU) on a feedforward neural network – to achieve the same outcome (Klambauer et al., 2017). SNNs push activations in the network to converge to zero mean and unit variance and maintains this as activations are propagated through many network layers, by drawing them towards a stable fixed point. In the general case, they maintain activation mean and variance within defined intervals provided weights meet certain mild conditions. Additionally, they stabilise variances to prevent exploding and vanishing gradients. SNNs are able to use strong regularisation techniques and are robust to perturbations, avoiding fluctuations in training error. This permits training of deeper models with many layers, increasing the quality of learning. They have been shown to significantly outperform simple feedforward networks with BN and specialised architectures on a range of tasks (Klambauer et al., 2017).

The SNN model is as follows (Klambauer et al., 2017). The base structure is a simple feedforward neural network, where the computation in the first layer is $\mathbf{z} = \mathbf{W}\mathbf{x}$ and $\mathbf{a} = f(\mathbf{z})$

where $\mathbf{z}$ is the pre-activation, $\mathbf{W}$ is the weights, $\mathbf{x}$ is the input, $\mathbf{a}$ is the activation and f is the activation function. For a single activation unit this is $z = \mathbf{w}^T\mathbf{x}$ and $a = f(z)$. Input features x_i have a mean $\mu = \mathbb{E}(x_i)$ and variance $\nu = Var(x_i)$. The preactivation z have mean $\mathbb{E}(z) = \sum_{i=1}^{n} w_i \mathbb{E}(x_i) = \mu\omega$ where $\omega = \sum_{i=1}^{n} w_i$ (sum of weights of all x_i input features) and variance $Var(z) = Var(\sum_{i=1}^{n} w_i x_i) = \nu\tau$ where $\tau = \sum_{i=1}^{n} w_i^2$ (sum of squared weights of all x_i input features). By the central limit theorem, z has a normal distribution $z \sim \mathcal{N}(\mu\omega, \sqrt{\nu\tau})$ with density $p_N(z;\ \mu\omega, \sqrt{\nu\tau})$. The activation a has $\tilde{\mu} = \mathbb{E}(a)$ and $\tilde{\nu} = Var(a)$. The function of SNNs is based on a mapping function g that maps μ and ν from one layer to $\tilde{\mu}$ and $\tilde{\nu}$ in the next layer, for every activation a. There is a stable and attracting fixed point for μ and ν that depend on ω and τ. When the mapping function g is repeatedly applied over a number of layers, μ and ν converge to the fixed point. Moreover, μ and ν always remain bounded by [μ_{min}, μ_{max}] and [ν_{min}, ν_{max}]. The SELU activation function – $selu(x) = \lambda \begin{cases} x & if\ x > 0 \\ \alpha e^x - \alpha & if\ x \leq 0 \end{cases}$ where α and λ are constants – allows the mapping g to have these properties. It has positive and negative values to control the mean, saturation regions and steeper slopes to dampen or amplify the variance when it is large or small respectively, and a continuous curve so there is a fixed point where variance damping and amplification is balanced. The SELU activation dampens variance for negative inputs with the effect stronger for inputs far from 0. In contrast, they increase variance for positive inputs with the effect stronger for inputs close to 0 (this arises as the exponential linear unit is multiplied by $\lambda > 1$ so the slope is greater than one for positive inputs). Therefore the mapping function g which maps μ and ν of a layer to $\tilde{\mu}$ and $\tilde{\nu}$ of the next layer is: $\tilde{\mu}(\mu, \omega, \nu, \tau) = \int_{-\infty}^{\infty} selu(z)\, p_N\left(z;\ \mu\omega, \sqrt{\nu\tau}\right) dz$ and $\tilde{\nu}(\mu, \omega, \nu, \tau) = \int_{-\infty}^{\infty} selu(z)^2\, p_N\left(z;\ \mu\omega, \sqrt{\nu\tau}\right) dz - \tilde{\mu}^2$. Weights ω and τ are initialised as $\omega = 0$ and $\tau = 1$ (they are drawn from a Gaussian distribution with $\mathbb{E}(w_i) = 0$ and $Var(w_i) = \frac{1}{n}$, where n is the number of features. For these weights, (μ, ν) = (0, 1) is a stable and attracting fixed point. During learning, weights ω and τ will deviate from 0 and 1 and no longer be normalised. However the self-normalising property of SNNs can be

maintained if these are bounded $\omega \in [-0.1, 0.1]$ and $\tau \in [0.95, 1.1]$ i.e. close to 0 and 1, in which case there still exists a fixed point $\mu \in [-0.03, 0.07]$ and $\nu \in [0.80, 1.49]$ i.e. (μ, ν) is close to (0, 1). SNNs also push variance of activations into a defined interval. For $\mu \in$ [-1, 1], $\omega \in$ [-0.1, 0.1], $\nu \in$ [3, 16], $\tau \in$ [0.8, 1.25], $\tilde{\nu}(\mu, \omega, \nu, \tau) < \nu$, hence the variance is bounded to $\nu <$ 3, preventing exploding gradients. For $\mu \in$ [-0.1, 0.1], $\omega \in$ [-0.1, 0.1], $\nu \in$ [0.02, 0.16] or $\nu \in$ [0.02, 0.24] and $\tau \in$ [0.8, 1.25] or $\tau \in$ [0.9, 1.25] then $\tilde{\nu}(\mu, \omega, \nu, \tau) > \nu$, hence the variance is bounded to $\nu >$ 0.16 or $\nu >$ 0.24, preventing vanishing gradients. SNNs also use a modified dropout technique: instead of standard dropout where the activation is preserved with probability p and is set to 0 with probability $1 - p$, alpha dropout is used which sets inputs to $-\lambda\alpha$ (the negative saturation value) with probability $1 - p$. Mean and variance is maintained at their original values after alpha dropout using scale and translation transformations. Model parameters that are trained are ω and τ.

3.5 Summary

Table 2 summarises the 14 specialised architectures used in this study. This chapter reviewed deep learning predictive models for tabular data. The empirical investigations of this study is presented in the next chapter.

Table 2. Summary of 14 specialised deep learning architectures for tabular data.

	Model family	Key computation	Trained parameters
NODE[1]	Differentiable tree	$f_i(\mathbf{x}) = \sum_{j=1}^{n} x_j \cdot entmax(F_{ij})$ $c_i(\mathbf{x}) = entmax([(f_i(\mathbf{x}) - b_i), 0])$	*F, b, R*

		$h(\mathbf{x}) = \sum_{i_1 \dots i_d \in \{0,1\}^d} R_{i_1 \dots i_d} \cdot C_{i_1 \dots i_d}(\mathbf{x})$	
Quantum Forest[2]	Differentiable tree	$g(\mathbf{A}, \mathbf{x}, b) = \sigma(entmax(\mathbf{A})\mathbf{x} - b)$ $p = \prod_n g_n$ $y(\mathbf{x}) = \sum_j p_j Q_j(\mathbf{x})$	*A, b, Q*
DNDT[3]	Differentiable tree	$f(x) = softmax(\frac{wx + [0, -\beta_1, \dots, -\beta_1 - \beta_2 - \dots - \beta_n]}{\tau})$ $\mathbf{z} = f_1(x_1) \otimes f_2(x_2) \otimes \dots \otimes f_D(x_D)$	$[\beta_1, \beta_2, \dots, \beta_n]$
TabNet[4]	Attention	$\mathbf{M}[\mathbf{i}] = sparsemax(\mathbf{P}[\mathbf{i}-\mathbf{1}] \cdot h_i(\mathbf{a}[\mathbf{i}-\mathbf{1}]))$ $[\mathbf{d}[\mathbf{i}], \mathbf{a}[\mathbf{i}]] = f_i(\mathbf{M}[\mathbf{i}] \cdot \mathbf{f})$ $\mathbf{d_{out}} = \sum_{i=1}^{N_{steps}} ReLU(\mathbf{d}[\mathbf{i}])$	Weights and biases in FC layers in the attentive and feature transformer
TabTransformer[5]	Attention	$\mathbf{A} = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{k}})$ $Attention(\mathbf{K}, \mathbf{Q}, \mathbf{V}) = \mathbf{A} \cdot \mathbf{V}$	ϕ in column embedding layer, θ in transformer layers, weight and biases in MLP
SAINT[6]	Attention	$\mathbf{A} = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{k}})$ $Attention(\mathbf{K}, \mathbf{Q}, \mathbf{V}) = \mathbf{A} \cdot \mathbf{V}$	ϕ in embedding layer, θ in transformer layers, weight and biases in MLP
Wide and Deep[7]	Feature interaction	Linear model: $y = \mathbf{w}^T\mathbf{x} + b$ Deep neural network: $\mathbf{a}^{l+1} = ReLU(\mathbf{W}^l \cdot \mathbf{a}^l + \mathbf{b}^l)$ $P(Y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}_{wide}^T[\mathbf{x}, \phi(\mathbf{x})] + \mathbf{w}_{deep}^T\mathbf{a}^{lf} + b)$	$\mathbf{w}$ and b in linear model, $\mathbf{W}$ and $\mathbf{b}$ in deep neural network, $\mathbf{w}_{\text{wide}}$, $\mathbf{w}_{\text{deep}}$ and b in the final sum layer
DeepFM[8]	Feature interaction	FM: $y_{FM}(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + \sum_{i=1}^{d} \sum_{j=i+1}^{d} \langle v_i, v_j \rangle x_i \cdot x_j$ Deep neural network: $\mathbf{a}^{l+1} = \sigma(\mathbf{W}^l \cdot \mathbf{a}^l + \mathbf{b}^l)$ $y(\mathbf{x}) = \sigma(y_{FM}(\mathbf{x}) + y_{Deep}(\mathbf{x}))$	$\mathbf{w}$ and v_i in FM, $\mathbf{W}$ and $\mathbf{b}$ in deep neural network
DCN[9]	Feature interaction	Cross network: $\mathbf{x}_{l+1} = \mathbf{x}_0\mathbf{x}_l^T\mathbf{w}_l + \mathbf{b}_l + \mathbf{x}_l$	$\mathbf{w}_l$ and $\mathbf{b}_l$ in cross network, $\mathbf{W}$

		Deep neural network: $\mathbf{a}^{l+1} = ReLU(\mathbf{W}^l \cdot \mathbf{a}^l + \mathbf{b}^l)$ $p = \sigma([\mathbf{x}_{L1}^T, \mathbf{a}_{L2}^T]\mathbf{w}_{logits})$	and $\mathbf{b}$ in deep neural network, $\mathbf{w}_{\text{logits}}$ in final combination layer
xDeepFM[10]	Feature interaction	CIN: $\mathbf{X}_{h,*}^k = \sum_{i=1}^{H_{k-1}} \sum_{j=1}^{m} \mathbf{W}_{ij}^{k,h} (\mathbf{X}_{i,*}^{k-1} \circ \mathbf{X}_{j,*}^0)$ Deep neural network: $\mathbf{a}^{l+1} = ReLU(\mathbf{W}^l \cdot \mathbf{a}^l + \mathbf{b}^l)$ $y = \sigma(\mathbf{w}_{linear}^T \mathbf{x} + \mathbf{w}_{dnn}^T \mathbf{a}_{dnn}^k + \mathbf{w}_{cin}^T \mathbf{p}^+ + b)$	$\mathbf{W}_{ij}^{k,h}$ in CIN, $\mathbf{W}$ and $\mathbf{b}$ in deep neural network, $\mathbf{w}_{linear}^T$, $\mathbf{w}_{dnn}^T$, $\mathbf{w}_{cin}^T$ and b of final output unit
PNN[11]	Feature interaction	Product layer: $g(\mathbf{v}_i, \mathbf{v}_j) = \langle \mathbf{v}_i, \mathbf{v}_j \rangle$ $l_p^n = \mathbf{W}_p^n \odot \mathbf{p} = \sum_{i=1}^{N} \sum_{j=1}^{N} (\mathbf{W}_p^n)_{i,j} \mathbf{p}_{i,j}$ $\mathbf{z} = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N)$ $l_z^n = \mathbf{W}_z^n \odot \mathbf{z} = \sum_{i=1}^{N} \sum_{j=1}^{M} (\mathbf{W}_z^n)_{i,j} \mathbf{z}_{i,j}$ Deep neural network: $\mathbf{l}_1 = ReLU(\mathbf{l}_p + \mathbf{l}_z + \mathbf{b}_1)$ $\mathbf{l}_2 = ReLU(\mathbf{W}_2 \mathbf{l}_1 + \mathbf{b}_2)$ $y = \sigma(\mathbf{W}_3 \mathbf{l}_2 + \mathbf{b}_3)$	$\mathbf{W}_p^n$ and $\mathbf{W}_z^n$ in product layers, $\mathbf{W}$ and $\mathbf{b}$ in deep neural network
RLN[12]	Regularisation	$\tilde{L}(\mathbf{X}, \mathbf{W}, \Lambda) = L(\mathbf{X}, \mathbf{W}) + \sum_{i=1}^{n} \exp(\lambda_i) \cdot \|w_i\|$ $g_{t,i} = \frac{\partial \mathrm{L}(\mathbf{X}_t, \mathbf{W}_t)}{\partial w_{t,i}}$ $r_{t.i} = \exp(\lambda_{t,i}) \cdot \frac{\partial \|w_{t,i}\|}{\partial w_{t,i}}$ $w_{t+1,i} = w_{t,i} - \eta \cdot \frac{\partial \tilde{L}(\mathbf{X}_t, \mathbf{W}_t, \Lambda_t)}{\partial w_{t,i}} = w_{t,i} - \eta \cdot (g_{t,i} + r_{t.i})$ $L_{CF}(\mathbf{X}_t, \mathbf{X}_{t+1}, \mathbf{W}_t, \Lambda_t, \eta) = L(\mathbf{X}_{t+1}, \mathbf{W}_{t+1})$ $\lambda_{t+1,i} = \lambda_{t,i} - \nu \cdot \frac{\partial L_{CF}(\mathbf{X}_t, \mathbf{X}_{t+1}, \mathbf{W}_t, \Lambda_t, \eta)}{\partial \lambda_{t,i}} = \lambda_{t,i} + \nu \cdot \eta \cdot g_{t+1,i} \cdot r_{t.i}$	$\mathbf{W}$, $\Lambda = \{\lambda_i\}_{i=1}^n$
MLR[13]	Regularisation	$\mathbf{P}(\theta, \lambda, \mathbf{x}) = [(\mathbf{A}^{L-1})^T \mathbf{A}^{L-1} + \lambda \mathbb{I}]^{-1} (\mathbf{A}^{L-1})^T$	θ, λ

		$\mathbf{H}(\theta,\lambda,\mathbf{x}) = \mathbf{A}^{L-1}\mathbf{P}(\theta,\lambda,\mathbf{x})$ $\mathbf{A}^{L} = \mathbf{A}^{L-1}\mathbf{W}^{L} = \mathbf{A}^{L-1}\,\mathbf{P}(\theta,\lambda,\mathbf{x})\mathbf{Y} = \mathbf{H}(\theta,\lambda,\mathbf{x})\mathbf{Y}$ $MLR(\theta,\lambda) = BCE(\mathbf{Y};(2\mathbf{Y}-1) + (\mathbb{I}_n - \mathbf{H})\xi + \mathbf{H}(2\mathbf{Y}-1)) + \frac{1}{T}\sum_{t=1}^{T}\lvert BCE(\mathbf{Y};\,\bar{\mathbf{Y}}\mathbb{1}_n) - BCE(\pi^t(2\mathbf{Y}-1);\,\pi^t(2\mathbf{Y}-1) + (\mathbb{I}_n - \mathbf{H})\xi_t + \mathbf{H}\pi^t(2\mathbf{Y}-1))\rvert$	
SNN[14]	Regularisation	$\omega = \sum_{i=1}^{n} w_i$ $\tau = \sum_{i=1}^{n} w_i^2$ $selu(x) = \lambda\begin{cases} x & if\ x > 0 \\ \alpha e^x - \alpha & if\ x \le 0 \end{cases}$ $\tilde{\mu}(\mu,\omega,\nu,\tau) = \int_{-\infty}^{\infty} selu(z)\,p_N\left(z;\,\mu\omega,\sqrt{\nu\tau}\right)dz$ $\tilde{\nu}(\mu,\omega,\nu,\tau) = \int_{-\infty}^{\infty} selu(z)^2\,p_N\left(z;\,\mu\omega,\sqrt{\nu\tau}\right)dz - \tilde{\mu}^2$	$\omega,\ \tau$

[1]Popov et al., 2019; [2]Chen, 2020; [3]Yang et al., 2018; [4]Arik & Pfister, 2019; [5]Huang et al., 2020; [6]Somepalli et al., 2021; [7]Cheng et al., 2016; [8]Guo et al., 2017; [9]Wang et al., 2017; [10]Lian et al., 2018; [11]Qu et al., 2016; [12]Shavitt & Segal, 2018; [13]Lounici et al., 2021; [14]Klambauer et al., 2017.

Chapter 4: Experimental Methods

This chapter presents the experimental methods underlying the study. The study can be divided into two parts: Part 1 which deals with synthetic tabular data generation and Part 2 which deals with deep learning for predictive tasks on tabular data; details of the model architectures, training process and evaluation metrics used for each are given in Sections 4.2 and 4.3, respectively. Prior to that, Section 4.1 explains the datasets used.

4.1 Data

This study used 4 datasets of hospital admissions during the Covid-19 pandemic collected from the EHRs of 4 NHS trusts – Oxford University Hospitals NHS Foundation Trust (Oxford), University Hospitals Birmingham NHS Foundation Trust (Birmingham), Portsmouth Hospitals University NHS Trust (Portsmouth) and Bedfordshire Hospitals NHS Foundation Trust (Bedford). Datasets consisted of anonymised patient records. The Portsmouth and Bedford datasets were collected exclusively within the Covid-19 pandemic period since March 2020, while the Oxford and Birmingham datasets were a mix of pandemic and pre-pandemic patient records. Approval for use of data was obtained from National Health Service Health Research Authority (IRAS ID 281832) and sponsored by the University of Oxford. This study was conducted within the CURIAL research programme (Soltan et al., 2021). The datapoints in each dataset represented one patient encounter. Each dataset consisted of 30 features: age, gender, ethnicity of the patient, 6 vital sign measurements, 20 blood tests results and the Covid-19 status of the patient at the time of hospital admission. Thus, there were 3 discrete and 27 continuous features. Of the 3 discrete features, two were binary and 1 was multiclass (7 classes). The task was to predict the presence (or absence) of Covid-19 infection in the patient based on their age,

gender, ethnicity, vital sign measurements and blood test results, which are routinely elicited and performed when patients attend an emergency department. Datasets ranged in size from several thousand to hundreds of thousands of samples. The number of samples in each were: Oxford - 217199, Birmingham - 205373, Portsmouth - 38717, Bedford - 1859. The prevalence of Covid-19 in all datasets was low. The number and percentage of positive Covid-19 cases in each were: Oxford - 3109 (1.43%), Birmingham - 789 (0.38%), Portsmouth - 2283 (5.90%) and Bedford - 209 (11.24%). Hence, prediction classes were highly imbalanced (<10%), especially in the mixed pandemic and pre-pandemic Oxford and Birmingham datasets. The number of samples with missing data in each dataset was: Oxford 33.1%, Birmingham 75.2%, Portsmouth 18.5% and Bedford 35.4%.

As the Portsmouth dataset included the largest number of Covid-19 positive cases and was collected exclusively during the Covid-19 pandemic period, this was selected to be the main training dataset, with the other 3 used for external validation. This setup allowed assessment of the generalisability of learnt models to a future "post-pandemic" situation when hospital admissions revert to the pre-pandemic situation but with low Covid-19 prevalence (which is represented by the Oxford and Birmingham datasets). The raw Portsmouth dataset with missing data excluded was used as a class-imbalanced training dataset. A 50:50 class-balanced training dataset was also created by performing under-sampling of the majority class (Covid-19 negative cases) in the imbalanced dataset. All models in Part 1 and 2 of the study used the same under-sampled dataset. The sample sizes of the two alternative training datasets were: Portsmouth balanced - 4318, Portsmouth imbalanced - 31537. As the sample sizes differ, the imbalanced dataset was randomly sampled to the same sample size as the balanced dataset, to ensure that training dataset size did not affect the performance of any model. All models in Part 1 and 2 of the study used the same randomly sampled dataset.

4.2 Part 1: Synthetic tabular data generation

4.2.1 Models

Models were implemented in Python v3.6. Python was chosen as it is one of the most well-equipped programming languages for machine learning, with libraries specialised for large-scale data pre-processing, such as NumPy and Pandas, which this study utilised. Python packages also enabled efficient implementation of machine learning algorithms; this study used Scikit-learn, in particular for scaling of data and evaluation metrics. The study also drew on the modularisation and object-oriented programming properties of Python.

4.2.1.1 Conditional tabular GAN

CTGAN was defined in Chapter 2. The generator had two fully connected hidden layers of dimension 256 with BN and ReLU activation function applied. Synthetic data was generated using different activation functions for different feature types; tanh was used to generate α_i values representing the value of a feature within a mode while softmax was used to generate β_i the one hot vector indicating the mode and discrete values $\mathbf{d}_i$. The latent dimension $|\mathbf{z}|$ was 32. The structure was:

1. $h_0 = \mathbf{z} \oplus cond$ where $z \sim \mathcal{N}(0, 1)$
2. $h_1 = h_0 \oplus ReLU(BN(FC_{|cond|+|\mathbf{z}|\to 256}(h_0)))$
3. $h_2 = h_1 \oplus ReLU(BN(FC_{|cond|+|\mathbf{z}|+256\to 256}(h_1)))$
4. $\hat{a}_i = \tanh(FC_{|cond|+|\mathbf{z}|+512\to 1}(h_2))$
5. $\hat{\beta}_i = gumbel_{0.2}(FC_{|cond|+|\mathbf{z}|+512\to m_i}(h_2))$
6. $\hat{\mathbf{d}}_i = gumbel_{0.2}(FC_{|cond|+|\mathbf{z}|+512\to |D_i|}(h_2))$

where h are hidden layer outputs (Xu et al., 2019).

The discriminator also has two fully connected hidden layers of dimension 256 with leaky ReLU function and dropout applied. The structure was:

1. $h_0 = \mathbf{x}_1 \oplus ... \oplus \mathbf{x}_{10} \oplus cond_1 \oplus ... \oplus cond_{10}$
2. $h_1 = dropout(leakyReLU_{0.2}\left(FC_{10|\mathbf{x}|+10|cond|\rightarrow 256}(h_0)\right))$
3. $h_2 = dropout(leakyReLU_{0.2}(FC_{256\rightarrow 256}(h_1)))$
4. $C(.) = FC_{256\rightarrow 1}(h_2)$

where h are hidden layer outputs (Xu et al., 2019).

CTGAN is trained with WGAN loss using Adam optimiser with learning rate of $2e^{-4}$ and weight decay $1e^{-6}$, as per Xu et al. (2019). Batch size was 250 datapoints and training proceeded for 300 epochs.

4.2.1.2 Tabular variational autoencoder

TVAE was defined in Chapter 2. An encoder architecture consisting of two fully connected layers with 128 hidden units with ReLU activation function applied was used. The output of the encoder was fed into two fully connected layers to predict the mean μ and variance σ^2 of $\mathbf{z}$. The latent dimension was 32. The structure is:

1. $h_1 = ReLU(FC_{|\mathbf{x}|\rightarrow 128}(\mathbf{x}))$
2. $h_2 = ReLU(FC_{128\rightarrow 128}(h_1))$
3. $\mu = FC_{128\rightarrow 32}(h_2)$

4. $\sigma^2 = \exp\left(\frac{1}{2} FC_{128\rightarrow 32}(h_2)\right)$

where h are hidden layer outputs (Xu et al., 2019).

The decoder architecture mirrored the encoder, with two fully connected layers of 128 hidden units with ReLU activation function applied. The structure was:

1. $h_1 = ReLU(FC_{128\rightarrow 128}(\mathbf{z}))$
2. $h_2 = ReLU(FC_{128\rightarrow 128}(h_1))$
3. $\bar{\alpha}_i = \tanh\left(FC_{128\rightarrow 1}(h_2)\right)$
4. $\hat{\alpha}_i \sim \mathcal{N}(\bar{\alpha}_i, \delta_i)$ where δ_i are network parameters
5. $\hat{\beta}_i \sim softmax(FC_{128\rightarrow m_i}(h_2))$
6. $\hat{\mathbf{d}}_i \sim softmax(FC_{128\rightarrow |D_i|}(h_2))$

where h are hidden layer outputs (Xu et al., 2019).

TVAE is trained with ELBO loss using Adam optimiser with learning rate $1e^{-3}$ and weight decay $1e^{-5}$, as per Xu et al. (2019). Batch size was 256 datapoints and training proceeded for 300 epochs.

4.2.1.3 Normalising flows

For the planar and Sylvester flow, the output of the TVAE encoder network was fed to separate linear layers which mapped datapoints to the flow parameters, generating the amortised flow parameters; in the case of planar flow, this was $\mathbf{u}$, $\mathbf{w}$ and b and, for Sylvester flow, this was $\mathbf{Q}$,

b, $\mathbf{R}$ and $\widetilde{\mathbf{R}}$ (Rezende & Mohamed, 2015; van den Berg et al., 2018). For Sylvester flow, the number of orthogonal vectors per orthogonal matrix was set to $M = 4$.

For NICE and RealNVP, the flow parameters were not data dependent. The additive coupling layer of NICE used a function m which was a fully connected MLP with 4 hidden layers of dimension 80. A scaling layer was used after each coupling layer. Random permutation for mixing was used (Dinh et al., 2014). The affine coupling layer of RealNVP used scale and translation functions which were both fully connected MLPs with 4 hidden layers of dimension 80. Alternate checkerboard masking was used (Dinh et al., 2017).

For each flow type, 5 flows were used. 10, 20 and 40 flows were investigated and, while performance was maintained, no significant performance improvement was found beyond 5 flows.

4.2.2 Training

Each model was trained on the Portsmouth class-balanced and imbalanced datasets as well as a real-world dataset with imbalanced classes and a significant amount of missing data (the Bedford dataset was chosen for this as it was also an exclusively pandemic dataset so had the most similar distribution to the Portsmouth dataset, allowing meaningful comparison of performance). Each training dataset was split into training and validation sets with ratio 80:20. The same train / validation split (using the same seed) was used for all experiments. After best hyperparameter settings were determined on the validation set, the model was trained with the entire dataset. The experiments for each model on each dataset were repeated 3 times with

different seeds and mean and standard deviations of the metrics over these repeats were reported.

Hyperparameters, including CTGAN and TVAE hidden and latent dimensions, learning rate, weight decay, batch size, number of epochs, and flow specific parameters, were tuned by grid search over defined search spaces specified by the papers which first proposed the models, with initial setting to the default configuration in the relevant paper (Xu et al., 2019; Rezende & Mohamed, 2015; van den Berg et al., 2018; Dinh et al., 2014; 2017). For detailed description of hyperparameter search spaces, see Appendix A. Hyperparameter settings were chosen based on best performance across all metrics on the validation set.

All experiments were run on a 3.00 GHz Intel Core i7 CPU with 16.0 GB RAM.

4.2.3 Metrics

The quality of generated synthetic data was evaluated using three classes of metrics (SDV, 2021).

Firstly, statistical metrics were used. The two sample Kolmogorov–Smirnov (KS) test was used to compare distributions of continuous features; higher values indicated synthetic data more closely matched real data (smaller disparity between observed and expected continuous distribution functions). The counterpart was the Chi-squared (CS) test, which compared the

distribution of discrete features; higher values indicated better synthetic data (higher probability of being sampled from the same distribution as the real data).

Secondly, machine learning detection metrics were used. Machine learning classifiers (logistic regression and support vector machine (SVC)) were trained to predict real vs synthetic data; the metric indicated the difficulty of this task (metric is 1 – AUC of the machine learning classifier). Higher values up to 0.5 (equivalent to classification AUC of 0.5) were better as they indicated the synthetic data was more indistinguishable from the real data.

Thirdly, machine learning efficacy metrics were used. This involved using the synthetic data as the training data for a predictive task and assessing the performance of the trained model on a test dataset of real data; the metric is the AUC of the classification model and higher values were better. Four types of classifiers were trained – decision tree, AdaBoost, logistic regression and MLP - and the average AUC was taken.

4.3 Part 2: Deep learning for prediction on tabular data

4.3.1 Models

Models were implemented in Python v3.6, in either the TensorFlow v2.5 / Keras v2.4.3 or PyTorch v1.7 libraries (TensorFlow, n.d.; Keras, n.d.; PyTorch, n.d.). Both TensorFlow and PyTorch are among the most popular and sophisticated deep learning libraries, enabling efficient construction, compilation and training of models on large datasets. The choice of library was based on the library used in original implementations of the models by the authors who first proposed them, and later existing implementations. Code written specifically for this

study used the PyTorch library. Details of model specific pre-processing, architecture, hyperparameters, regularisation cocktails used and their hyperparameters are given for each model.

NODE used 2 layers, 512 trees per layer, 1024 total trees, tree depth of 6, tree output dimension of 3, QHAdam optimiser, learning rate of $1e^{-3}$ and batch size of 256 (Popov et al., 2019). Regularisation cocktail ingredients included: BN on each layer, dropout of 0.1 in each layer and weight decay of $1e^{-4}$. Leave-one-out encoding was used for discrete features and quantile normal transformation was used for continuous features for stable training and faster convergence, similar to in Popov et al. (2019).

Quantum Forest used 1 layer, 1024 total trees, tree depth of 5, QHAdam optimiser, learning rate of $1e^{-3}$, and batch size of 256 (Chen, 2020). Regularisation cocktail ingredients included: BN on each layer, dropout of 0.1 in each layer, weight decay of $1e^{-8}$, L1 regularisation of 0.01, gate regularisation of 0.1 and Lookahead with k of 5 and slow weights step size of 0.5. Leave-one-out encoding was used for discrete features and quantile normal transformation was used for continuous features, similar to in Chen (2020).

DNDT used one cut point per feature, temperature τ of 0.1, Adam optimiser and learning rate of 0.1 (Yang et al., 2018). Regularisation cocktail ingredients included: weight decay of $1e^{-4}$ and Lookahead with k of 5 and slow weights step size of 0.5. As DNDT did not scale well to larger numbers of features, separate models were trained on subsets of 7 features resulting in 5 models. The final prediction was obtained by majority voting, similar to in Yang et al. (2018).

TabNet used 3 decision steps, N_d and N_a of 256, γ relaxation factor of 1, λ_{sparse} sparsity coefficient of $1e^{-4}$, momentum m_B of 0.9, virtual batch size B of 32, Adam optimiser, learning rate of $1e^{-3}$, and batch size of 256 (Arik & Pfister, 2019). Regularisation cocktail ingredients included: BN on each layer in feature and attentive transformers, dropout of 0.1 in each layer and weight decay of $1e^{-4}$. The attentive transformer consisted of one fully connected layer and sparsemax function. The feature transformer consisted of two shared and two decision step-dependent layers, each of which consisted of a fully connected layer and gated linear units (GLU) non-linear activation functions connected to a normalised residual connection to stabilise variance and learning. No processing is done for continuous features, as per Arik & Pfister (2019).

TabTransformer used categorical embedding dimension of 32, 6 layers, 8 attention heads, self-attention layer dimension of 16, MLP hidden dimension of (4l, 2l) where l is the input size, Adam optimiser, learning rate of $1e^{-3}$, and batch size of 128 (Huang et al., 2020). Regularisation cocktail ingredients included: pre-normalisation on each attention and feedforward layer, dropout of 0.1 in each attention and feedforward layer, BN on each MLP layer, dropout of 0.1 in each MLP layer, weight decay of $1e^{-4}$, SWA with initial learning rate of $1e^{-2}$, averaging period of 5 epochs, starting at 50 epochs, and Lookahead with k of 5 and slow weights step size of 0.5. The feedforward block consisted of two fully connected layers with GLU non-linear activation function. The final MLP consisted of two fully connected hidden layers with RELU non-linear activation function. Categorical features were label encoded, as per Huang et al. (2020).

SAINT used embedding dimension of 32, 6 layers, 8 attention heads, self-attention layer dimension of 16, intersample attention layer dimension of 64, MLP hidden dimension of (4l, 2l) where l is the input size, Adam optimiser, learning rate of $1e^{-3}$, and batch size of 256 (Somepalli et al., 2021). Regularisation cocktail ingredients included: pre-normalisation on each attention and feedforward layer, dropout of 0.1 in each attention layer, dropout of 0.8 in each feedforward layer, BN on each MLP layer, dropout of 0.1 in each MLP layer, weight decay of $1e^{-4}$, and Lookahead with k of 5 and slow weights step size of 0.5. The feedforward block consisted of two fully connected layers with GLU non-linear activation function. The final MLP consisted of two fully connected hidden layers with RELU non-linear activation function. Categorical features were label encoded, as per Somepalli et al. (2021).

Wide and Deep used categorical embedding dimension of 32, 3 hidden layers and 1024 → 512 → 256 hidden units in the deep neural network, Adam optimiser, learning rate of $1e^{-2}$, and batch size of 128 (Cheng et al., 2016). Regularisation cocktail ingredients included: BN in each deep neural network layer, dropout of 0.5 and L2 regularisation of $1e^{-4}$ in each layer, BN and L2 regularisation of $1e^{-4}$ in the linear model, SE with initial learning rate of $2e^{-1}$, 100 total epochs and 5 snapshots, and Lookahead with k of 5 and slow weights step size of 0.5.

DeepFM used categorical embedding dimension of 32, 3 hidden layers and 400 hidden units per layer in the deep neural network, Adam optimiser, learning rate of $1e^{-2}$, and batch size of 128 (Guo et al., 2017). Regularisation cocktail ingredients included: BN in each deep neural network layer, dropout of 0.5 and L2 regularisation of $1e^{-4}$ in each layer, BN and L2 regularisation of $1e^{-4}$ in the linear model, SWA with averaging period of 5 epochs, starting at 10 epochs, and Lookahead with k of 5 and slow weights step size of 0.5.

DCN used categorical embedding dimension of 32, 3 hidden layers and 1024 hidden units per layer in the deep neural network, 6 cross layers, Adam optimiser, learning rate of $1e^{-2}$, and batch size of 128 (Wang et al., 2017). Regularisation cocktail ingredients included: BN in each deep neural network layer, dropout of 0.5 and L2 regularisation of $1e^{-4}$ in each layer, dropout of 0.1 and L2 regularisation of $1e^{-4}$ in each cross layer, SWA with averaging period of 5 epochs, starting at 10 epochs, and Lookahead with k of 5 and slow weights step size of 0.5.

xDeepFM used categorical embedding dimension of 32, 3 hidden layers and 400 hidden units per layer in the deep neural network, 3 hidden layers and 100 hidden units per layer in the CIN network, Adam optimiser, learning rate of $1e^{-2}$, and batch size of 128 (Lian et al., 2018). Regularisation cocktail ingredients included: BN in each deep neural network layer, dropout of 0.5 and L2 regularisation of $1e^{-4}$ in each layer, dropout of 0.1 and L2 regularisation of $1e^{-4}$ in each CIN layer, BN and L2 regularisation of $1e^{-4}$ in the linear model, SWA with averaging period of 5 epochs, starting at 10 epochs, and Lookahead with k of 5 and slow weights step size of 0.5.

PNN used categorical embedding dimension of 32, 3 hidden layers and 400 hidden units per layer in the deep neural network, Adam optimiser, learning rate of $1e^{-2}$, and batch size of 128 (Qu et al., 2016). Regularisation cocktail ingredients included: BN in each deep neural network layer, dropout of 0.5 and L2 regularisation of $1e^{-4}$ in each layer, L2 regularisation of $1e^{-4}$ in the outer product layer, SE with initial learning rate of $2e^{-1}$, 100 total epochs and 5 snapshots, and Lookahead with k of 5 and slow weights step size of 0.5.

RLN used 4 layers, regularisation coefficients learning rate ν of $1e^{6}$, normalisation factor θ of -10, Adam optimiser, weights learning rate η of $1e^{-3}$, and batch size of 128 (Shavitt & Segal, 2018). Regularisation cocktail ingredients included: SWA with averaging period of 5 epochs, starting at 50 epochs.

MLR used 3 layers, 1024 hidden units, 16 permutations, dither of 0.03, Adam optimiser, learning rate of $1e^{-3}$, and batch size of 128 (Lounici et al., 2021). Regularisation cocktail ingredients included: Lookahead with k of 5 and slow weights step size of 0.5.

SNN used 8 hidden layers, 1024 → 512 → 256 hidden units, Adam optimiser, learning rate of $1e^{-3}$, and batch size of 128 (Klambauer et al., 2017). Regularisation cocktail ingredients included: dropout of 0.05 in each layer, weight decay of $1e^{-4}$, SWA with initial learning rate of $1e^{-2}$, averaging period of 5 epochs, starting at 10 epochs, and Lookahead with k of 5 and slow weights step size of 0.5.

4.3.2 Training

All datasets were processed prior to training. All discrete features were one hot encoded and all continuous features were normalised to zero mean and unit variance, unless otherwise stated for specific deep learning models in the previous section.

Each model was trained on the Portsmouth class-balanced and imbalanced datasets. Each training dataset was split into training, validation and test sets: the dataset was first split into

training and test sets with ratio 80:20, and the training set was further split into training and validation sets with ratio 80:20. The same train / validation / test splits (using the same seed) were used for all experiments. The experiments on each model on each dataset with each regularisation status were replicated 3 times using different seeds for the train / validation / test split and the mean and standard deviation of the metrics over these repeats were reported.

All models were trained to minimise cross-entropy loss. Hyperparameters, including those for regularisation cocktails, stated for each model in the previous section were tuned by grid search over defined search spaces specified by the papers which first proposed the models, with initial setting to the default configuration in the relevant paper (Popov et al., 2019; Chen, 2020; Yang et al., 2018; Arik & Pfister, 2019; Huang et al., 2020; Somepalli et al., 2021; Cheng et al., 2016; Guo et al., 2017; Wang et al., 2017; Lian et al., 2018; Qu et al., 2016; Shavitt & Segal, 2018; Lounici et al., 2021; Klambauer et al., 2017; Srivastava et al., 2014; Huang et al., 2017; Ioffe & Szegedy, 2015; Izmailov et al., 2018; Zhang et al., 2019). For detailed description of hyperparameter search spaces, see Appendix A. Hyperparameter settings were chosen based on best AUC on the validation set.

Once models were trained, they were evaluated on the Portsmouth test set and 3 external validation datasets (Bedford, Oxford and Birmingham) and these performance metrics were reported.

All experiments were run on a 3.00 GHz Intel Core i7 CPU with 16.0 GB RAM.

4.3.3 Metrics

To quantify classification performance, area under the ROC curve (AUC) was reported. This is a common metric for binary classification problems; higher AUC corresponds to better performance (Bradley, 1997). It captures how well classes can be separated by a model and is particularly suitable as it is not sensitive to class imbalance (unlike accuracy) so allows fair comparison between the performance on class-balanced and imbalanced training datasets. Additionally, unlike precision or recall, there is no need to set a threshold.

4.4 Summary

This chapter detailed the methods of the empirical investigations. Discussion of experimental results is presented in the next chapter.

Chapter 5: Results

This chapter presents the results of empirical investigations in Part 1 and 2 of the study, in Sections 5.1 and 5.2 respectively.

5.1 Part 1: Synthetic tabular data generation

Tables 3 – 5 gives the results for Part 1 of the study – evaluation of the performance of CTGAN, vanilla TVAE and TVAE with four types of normalising flow on a real EHR dataset of hospital admissions during the Covid-19 pandemic, in terms of quality of synthetic data generated. The results in Table 3 pertain to training of models on a class-balanced dataset, Table 4 gives the results of training on a class-imbalanced dataset and Table 5 gives the results of training on a real-world dataset (Bedford) with class imbalance and missing data. In all cases, the degree of imbalance and missing data is given in Chapter 5.

With the exception of planar flows, all other flow types – Sylvester flow, NICE and RealNVP – improved generative performance of the vanilla TVAE. On statistical metrics, TVAE with normalising flows increased CS test by 1-2% and KS test by up to 1% compared to vanilla TVAE, when trained on the balanced Portsmouth dataset (Table 3). On the imbalanced Portsmouth dataset, the improvement was 2-3% for CS test and up to 1% for KS test (Table 4), while both metrics were equivalent on the real-world Bedford dataset between vanilla TVAE and TVAE with normalising flows (Table 5). TVAE with normalising flows increased the difficulty of the machine learning task to distinguish between real and synthetic data, as shown by higher logistic and SVC detection metrics, compared to the vanilla TVAE. When trained on

the balanced dataset, the increase was 4-7% for logistic and 3-5% for SVC detection; on the imbalanced dataset, the increase was 4-6% for logistic and 1-3% for SVC detection; on the real-world dataset, the increase was 1-3% for logistic and up to 3% for SVC detection. In the case of Sylvester flow, NICE and RealNVP, logistic detection was close to 50% indicating that the real and synthetic data could not be classified correctly more often than by chance. This suggests that normalising flows generates synthetic data which very closely resembles the real data, and one can speculate this represents state-of-the-art performance. Similarly, on machine learning efficacy metrics, average classifier performance when machine learning classifiers were trained with synthetic data was higher for synthetic data generated using TVAE with normalising flows than the vanilla TVAE, with improvements in AUC of up to 1% on the balanced dataset, up to 2% on the imbalanced dataset and 1-2% on the real-world dataset. It is important to note though that the machine learning efficacy metrics are constrained by the prediction task and its difficulty, so this may have limited the size of the differences. Sylvester flow, NICE and RealNVP had comparable performance. Planar flows performed equivalently to the vanilla TVAE across most metrics on all datasets. On the balanced dataset, CTGAN had substantially worse performance across all metrics than vanilla TVAE and TVAE with normalising flows. As CTGAN is considered a state-of-the-art solution for synthetic tabular data generation, this further supports the notion that TVAEs with normalising flows advances the current state-of-the-art, at least on datasets of a similar nature to the EHR datasets used in this study.

CTGAN, vanilla TVAE and TVAE with normalising flows were robust to training on datasets with imbalanced classes and missing data (Tables 4 and 5). Comparing performance of models trained on the different datasets, the quality of synthetic data as measured by statistical and machine learning detection metrics did not appear to be negatively impacted by imbalanced or missing data. The only metric that was substantially lower was machine learning efficacy (training on imbalanced data resulted in 15% reduction in AUC) but this is likely dictated by the

prediction task itself and its increased difficulty when data is imbalanced and there are fewer training opportunities for the minority class. Interestingly, CTGAN demonstrated the best machine learning efficacy of all models when trained on an imbalanced dataset.

Table 3. Generative performance of CTGAN and TVAE with and without normalising flows trained on class-balanced Portsmouth dataset.

	vanilla TVAE	TVAE + Planar flow	TVAE + Sylvester flow	TVAE + NICE	TVAE + Real NVP	CTGAN
CS test mean ± std	0.9587 ± 0.0193	0.9559 ± 0.0154	**0.9701 ± 0.0164**	0.9682 ± 0.0244	0.9605 ± 0.0128	0.8566 ± 0.0671
KS test mean ± std	0.8630 ± 0.0019	0.8596 ± 0.0025	0.8732 ± 0.0039	**0.8750 ± 0.0039**	0.8716 ± 0.0059	0.7816 ± 0.0268
LR detection mean ± std	0.4490 ± 0.0127	0.4487 ± 0.0332	**0.5191 ± 0.0386**	0.4903 ± 0.0096	0.4906 ± 0.0082	0.135 ± 0.0426
SVC detection mean ± std	0.1777 ± 0.0188	0.1438 ± 0.0168	**0.2300 ± 0.0355**	0.2235 ± 0.0341	0.2085 ± 0.0265	0.0287 ± 0.0127
Average classifier mean ± std	0.7674 ± 0.0092	0.7706 ± 0.0031	0.7682 ± 0.0019	0.7670 ± 0.0072	**0.7718 ± 0.0019**	0.7426 ± 0.0081
DT classifier mean ± std	0.7110 ± 0.0080	0.7204 ± 0.0024	0.7269 ± 0.0138	0.7235 ± 0.0130	0.7282 ± 0.0120	0.6950 ± 0.0324
AdaBoost classifier mean ± std	0.7825 ± 0.0142	0.7855 ± 0.0029	0.7804 ± 0.0037	0.7778 ± 0.0049	0.7810 ± 0.0040	0.7604 ± 0.0139
LR classifier mean ± std	0.7849 ± 0.0091	0.7859 ± 0.0055	0.7788 ± 0.0028	0.7825 ± 0.0090	0.7857 ± 0.0035	0.7566 ± 0.0115
MLP classifier mean ± std	0.7912 ± 0.0075	0.7908 ± 0.0053	0.7867 ± 0.0045	0.7844 ± 0.0089	0.7922 ± 0.0068	0.7586 ± 0.0046

*Best performances on each metric are in bold. CS test: Chi-squared test; KS test: Kolmogorov–Smirnov test; SVC: support vector machine classifier; DT: decision tree; LR: logistic regression; MLP: multi-layer perceptron; std: standard deviation.

Table 4. Generative performance of CTGAN and TVAE with and without normalising flows trained on class-imbalanced Portsmouth dataset.

	vanilla TVAE	TVAE + Planar flow	TVAE + Sylvester flow	TVAE + NICE	TVAE + Real NVP	CTGAN
CS test mean ± std	0.9357 ± 0.0171	0.9542 ± 0.0086	0.9537 ± 0.0083	0.9554 ± 0.0067	**0.9679 ± 0.0157**	0.6671 ± 0.0459
KS test mean ± std	0.8695 ± 0.0024	0.8615 ± 0.0026	**0.8740 ± 0.0089**	0.8696 ± 0.0056	0.8726 ± 0.0129	0.7673 ± 0.0289
LR detection mean ± std	0.4491 ± 0.0282	0.4394 ± 0.0128	0.4845 ± 0.0236	0.4939 ± 0.0176	**0.5059 ± 0.0541**	0.0778 ± 0.0272
SVC detection mean ± std	0.2355 ± 0.0407	0.1962 ± 0.0163	0.2517 ± 0.0208	**0.2654 ± 0.0100**	0.2448 ± 0.0265	0.0194 ± 0.0098
Average classifier mean ± std	0.6233 ± 0.0150	0.6144 ± 0.0244	0.6431 ± 0.0196	0.6262 ± 0.0304	0.6394 ± 0.0136	**0.6586 ± 0.0283**
DT classifier mean ± std	0.6127 ± 0.0061	0.5878 ± 0.0192	0.6228 ± 0.0323	0.6051 ± 0.0177	0.6061 ± 0.0167	0.5829 ± 0.0260
AdaBoost classifier mean ± std	0.5836 ± 0.0110	0.5807 ± 0.0263	0.5892 ± 0.0252	0.6063 ± 0.0372	0.6053 ± 0.0207	0.6592 ± 0.0386
LR classifier mean ± std	0.7119 ± 0.0199	0.6813 ± 0.0279	0.7221 ± 0.0097	0.6958 ± 0.0435	0.7221 ± 0.0319	0.7330 ± 0.0084
MLP classifier mean ± std	0.5850 ± 0.0334	0.6079 ± 0.0393	0.6382 ± 0.0313	0.5976 ± 0.0366	0.6243 ± 0.0211	0.6594 ± 0.0552

*Best performances on each metric are in bold. CS test: Chi-squared test; KS test: Kolmogorov–Smirnov test; SVC: support vector machine classifier; DT: decision tree; LR: logistic regression; MLP: multi-layer perceptron; std: standard deviation.

Table 5. Generative performance of CTGAN and TVAE with and without normalising flows trained on real-world Bedford dataset with class imbalanced and missing data.

	vanilla TVAE	TVAE + Planar flow	TVAE + Sylvester flow	TVAE + NICE	TVAE + Real NVP	CTGAN

CS test mean ± std	0.9817 ± 0.0130	0.9839 ± 0.0036	**0.9865 ± 0.0102**	0.9823 ± 0.0104	**0.9865 ± 0.0113**	0.9067 ± 0.0622
KS test mean ± std	**0.8948 ± 0.0057**	0.8765 ± 0.0066	0.8895 ± 0.0040	0.8917 ± 0.0044	0.8866 ± 0.0060	0.7334 ± 0.0030
LR detection mean ± std	0.4830 ± 0.0390	0.4239 ± 0.0111	0.5059 ± 0.0410	0.4938 ± 0.0057	**0.5167 ± 0.0274**	0.0295 ± 0.0111
SVC detection mean ± std	0.2981 ± 0.0176	0.1960 ± 0.0185	0.2904 ± 0.0318	0.3008 ± 0.0033	**0.3248 ± 0.0081**	0.0093 ± 0.0037
Average classifier mean ± std	0.7187 ± 0.0264	0.7174 ± 0.0052	**0.7371 ± 0.0187**	0.7291 ± 0.0094	0.7279 ± 0.0091	0.5165 ± 0.0266
DT classifier mean ± std	0.6714 ± 0.0334	0.6740 ± 0.0267	0.7193 ± 0.0218	0.6888 ± 0.0073	0.6882 ± 0.0196	0.4998 ± 0.0394
AdaBoost classifier mean ± std	0.7060 ± 0.0052	0.6971 ± 0.0250	0.7154 ± 0.0151	0.6931 ± 0.0126	0.6876 ± 0.0185	0.4973 ± 0.0033
LR classifier mean ± std	0.7812 ± 0.0281	0.7758 ± 0.0084	0.7962 ± 0.0206	0.8046 ± 0.0136	0.8063 ± 0.0080	0.5620 ± 0.0646
MLP classifier mean ± std	0.7162 ± 0.0424	0.7227 ± 0.0156	0.7174 ± 0.0214	0.7301 ± 0.0075	0.7297 ± 0.0106	0.5067 ± 0.0112

*Best performances on each metric are in bold. CS test: Chi-squared test; KS test: Kolmogorov–Smirnov test; SVC: support vector machine classifier; DT: decision tree; LR: logistic regression; MLP: multi-layer perceptron; std: standard deviation.

5.2 Part 2: Deep learning for prediction on tabular data

Tables 6 – 7 gives the results for Part 2 of the study – evaluation of the performance of 14 state-of-the-art deep learning models with and without regularisation cocktails on real EHR datasets of hospital admissions during the Covid-19 pandemic, in the task of predicting Covid-19 infection status. The results in Table 6 pertain to training of models on a class-balanced dataset while Table 7 gives the results of training on a class-imbalanced dataset. All trained models

were evaluated on 3 real-world external validation datasets, with class imbalance and missing data. In all cases, the degree of imbalance and missing data is given in Chapter 5.

When trained on the class-balanced dataset, all models performed reasonably with AUC > 0.7 (Table 6). Feature interaction-based models consistently yielded the best performances with AUC > 0.8 in almost all cases on all datasets. Best performances in this class of models were achieved by Wide and Deep and PNN, which both achieved AUC of 0.84 – 0.90 on all test and external validation datasets. These performances match and surpass that of state-of-the-art XGBoost models (previously investigated by other members of the research group), which achieve AUC of 0.85 – 0.88 in the same task. It also surpasses performance of simple MLPs (also previously investigated), which achieve AUC of around 0.75 in the same task. This suggests that relatively simple models which robustly encode feature interactions and are ensembled are most promising. Of other classes of models, SAINT also performed particularly well, achieving AUC of 0.83 – 0.88, suggesting that self and intersample attention is valuable.

Regularisation cocktails almost universally improved performance of the 14 models trained on the class-balanced dataset (Table 6). Larger performance improvements were particularly seen on the 3 external validation datasets, supporting the hypothesis that regularisation cocktails improve the generalisation capacity of specialised architectures. For example, differentiable tree-based models with regularisation cocktails outperformed base models specifically on external validation datasets. AUC of NODE was increased by up to 4%, Quantum Forest by 1% and DNDT by 1-3%. For attention-based models, regularisation cocktails resulted in substantial improvement to TabNet performance with ≥10% increase in AUC on each test set. The performance improvement for TabTransformer was small but for SAINT, there was a 1-8% increase in AUC on each test set. Of the feature interaction-based models, regularisation

cocktails improved performance of all models on all datasets, with AUC improvements of 3 – 4% in general but ranging from 1 – 6%. The performance of some models such as Wide and Deep, PNN and SAINT with regularisation cocktails matches and exceeds the performance of XGBoost as already mentioned; however, these models without regularisation cocktails do not, hence one can speculate that the addition of regularisation cocktails advances the state-of-the-art performance of deep learning models for predictive tasks on tabular data. In contrast, addition of regularisation cocktails made little difference to the performance of regularisation-based architectures: differences in AUC were <1% for RLN and MLR, although for SNN, performance improvements of 1 – 3% were achieved.

The performance of models trained on a balanced dataset were reasonably robust when applied to real-world imbalanced datasets with missing data (Table 6). Performance was generally lower on external validation datasets with imbalanced classes and missing data, compared to the test set held out from the training set which was balanced and had no missing data, but AUC differences were < 0.1 in models, with and without regularisation cocktails.

Table 6. Predictive performance of 14 specialised deep learning architectures trained on class-balanced Portsmouth dataset with and without regularisation cocktails.

	Portsmouth mean ± std	Bedford mean ± std	Oxford mean ± std	Birmingham mean ± std
NODE reg	0.7363 ± 0.0105	0.778 ± 0.0167	0.7531 ± 0.0038	0.7014 ± 0.0025
NODE no reg	0.7966 ± 0.0017	0.7397 ± 0.0114	0.718 ± 0.0042	0.7108 ± 0.0026
Quantum Forest reg	0.7724 ± 0.0081	0.7173 ± 0.0083	0.7317 ± 0.0015	0.7353 ± 0.0127
Quantum Forest no reg	0.7881 ± 0.004	0.7045 ± 0.0028	0.7295 ± 0.0024	0.7209 ± 0.001

DNDT reg	0.699 ± 0.0644	0.6737 ± 0.0699	0.6948 ± 0.0348	0.711 ± 0.0345
DNDT no reg	0.671 ± 0.0603	0.6471 ± 0.0784	0.6623 ± 0.0579	0.7031 ± 0.0326
TabNet reg	0.7603 ± 0.0116	0.77 ± 0.0087	0.728 ± 0.0128	0.7597 ± 0.0091
TabNet no reg	0.6379 ± 0.0243	0.6375 ± 0.0196	0.6392 ± 0.0077	0.6563 ± 0.0103
TabTransformer reg	0.8114 ± 0.0013	0.8165 ± 0.0171	0.7833 ± 0.0027	0.7985 ± 0.0034
TabTransformer no reg	0.8054 ± 0.0038	0.8287 ± 0.0041	0.7808 ± 0.0035	0.7935 ± 0.0072
SAINT reg	**0.8873 ± 0.0181**	**0.8810 ± 0.0257**	**0.8397 ± 0.0055**	**0.8463 ± 0.0242**
SAINT no reg	0.8033 ± 0.1457	0.8713 ± 0.0305	0.7893 ± 0.0265	0.775 ± 0.0459
Wide Deep reg	**0.8787 ± 0.0116**	**0.8990 ± 0.0067**	**0.8451 ± 0.0102**	**0.8540 ± 0.0160**
Wide Deep no reg	0.8677 ± 0.0069	0.8609 ± 0.0053	0.8119 ± 0.0018	0.7991 ± 0.0173
DeepFM reg	0.8894 ± 0.0012	0.8977 ± 0.0081	0.8539 ± 0.0040	0.8178 ± 0.0271
DeepFM no reg	0.8673 ± 0.0052	0.8549 ± 0.0081	0.8055 ± 0.014	0.7889 ± 0.0168
DCN reg	**0.8921 ± 0.0027**	**0.8958 ± 0.0055**	**0.8547 ± 0.0017**	**0.8386 ± 0.0315**
DCN no reg	0.8664 ± 0.0012	0.8584 ± 0.0032	0.815 ± 0.001	0.8089 ± 0.0194
xDeepFM reg	0.8928 ± 0.0026	0.8945 ± 0.0062	0.8579 ± 0.0031	0.8274 ± 0.0118
xDeepFM no reg	0.8685 ± 0.0059	0.8562 ± 0.0065	0.8145 ± 0.0079	0.8149 ± 0.0201
PNN reg	**0.8777 ± 0.0021**	**0.8980 ± 0.0087**	**0.8533 ± 0.0024**	**0.8458 ± 0.0145**
PNN no reg	0.8437 ± 0.038	0.8478 ± 0.0425	0.8029 ± 0.0335	0.819 ± 0.0407
RLN reg	0.8189 ± 0.0040	0.7103 ± 0.0070	0.7626 ± 0.0046	0.7242 ± 0.0094
RLN no reg	0.8191 ± 0.0113	0.7096 ± 0.0107	0.7508 ± 0.0108	0.7258 ± 0.0044

MLR reg	0.8105 ± 0.0103	0.7064 ± 0.0114	0.7688 ± 0.0027	0.7448 ± 0.0057
MLR no reg	0.8096 ± 0.0061	0.7112 ± 0.0107	0.7666 ± 0.0069	0.7426 ± 0.0081
SNN reg	0.7959 ± 0.0253	0.7140 ± 0.0027	0.7610 ± 0.0095	0.7458 ± 0.0142
SNN no reg	0.7861 ± 0.0342	0.683 ± 0.0113	0.737 ± 0.0081	0.7304 ± 0.0086

*Best-performing models are in bold. reg: with regularisation cocktail; no reg: without regularisation cocktail; std: standard deviation.

For the majority of models, training on an imbalanced dataset degraded performance, suggesting they are not robust to imbalanced training data, although the best-performing models were (Table 7). For differentiable tree-based models, significantly lower performance was observed. In some cases, learning was obviated altogether resulting in AUC near 0.5, but it was still possible to achieve AUC of 0.65 – 0.75 on NODE (representing a 0.05 – 0.1 decrease). A similar picture was seen with attention-based models, in particular TabNet. TabTransformer can achieve an intermediate level of performance with AUC 0.65 – 0.7 (representing a 0.15 decrease), but SAINT appears to be more robust to training on imbalanced data, with the model having equivalent performance (AUC 0.85 – 0.9) to that trained on balanced data. Feature interaction-based models were also relatively robust to imbalanced data; modest declines in AUC of 0.03 – 0.05 in general and up to 0.1 were observed. Finally, a significant decrease in performance was seen with regularisation-based architectures, with typical declines in AUC of 0.15 – 0.2 for RLN and MLR and 0.1 for SNN.

Regularisation cocktails also improved performance in the setting of imbalanced training data (Table 7). Moreover, performance improvements conferred by addition of regularisation cocktails tended to be larger when models were trained on imbalanced data compared to balanced data. This can be seen more clearly in the models that are more robust to imbalanced

data. For SAINT, regularisation cocktails led to improvements in AUC of around 5%. For feature interaction-based models, regularisation cocktails led to 5-10% increases in AUC in general which was higher than the 3 – 4% increase when training with balanced data. Regularisation cocktails also resulted in a more noticeable performance improvement in regularisation-based architectures when training on imbalanced compared to balanced data: there was an increase in AUC of up to 3% for RLN and 2-3% for MLR. For the models which lack robustness to imbalanced data, regularisation cocktails can be a mechanism of increasing robustness. For example, for NODE and TabTransformer, the addition of cocktails promoted effective learning, with AUC improvements of 15-25%. However, this effect was not consistent, as shown by the results of Quantum Forest, DNDT and TabNet.

Performance was similar on external validation datasets with imbalanced classes and missing data, compared to the test set held out from the training set which was imbalanced but had no missing data, with decline in performance < 0.05 in the vast majority of models, with and without regularisation cocktails (Table 7). This suggests the robustness of models when applied to missing data was good.

Table 7. Predictive performance of 14 specialised deep learning architectures trained on class-imbalanced Portsmouth dataset with and without regularisation cocktails.

	Portsmouth mean ± std	Bedford mean ± std	Oxford mean ± std	Birmingham mean ± std
NODE reg	0.7424 ± 0.0025	0.6506 ± 0.0022	0.702 ± 0.0047	0.7053 ± 0.0036
NODE no reg	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0
Quantum Forest reg	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0

Quantum Forest no reg	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0
DNDT reg	0.5119 ± 0.0153	0.5093 ± 0.014	0.5144 ± 0.0198	0.5202 ± 0.0314
DNDT no reg	0.5002 ± 0.0153	0.5166 ± 0.0148	0.5344 ± 0.0189	0.5644 ± 0.0316
TabNet reg	0.5641 ± 0.0075	0.5257 ± 0.005	0.5455 ± 0.012	0.5336 ± 0.0117
TabNet no reg	0.5153 ± 0.0127	0.5471 ± 0.0198	0.5527 ± 0.0119	0.5541 ± 0.012
TabTransformer reg	0.6698 ± 0.0146	0.675 ± 0.0479	0.6552 ± 0.0269	0.6529 ± 0.0579
TabTransformer no reg	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0
SAINT reg	**0.854 ± 0.0421**	**0.897 ± 0.0079**	**0.8593 ± 0.0133**	**0.8697 ± 0.0116**
SAINT no reg	0.843 ± 0.03	0.8523 ± 0.02	0.804 ± 0.0578	0.7967 ± 0.0627
Wide Deep reg	0.8258 ± 0.0194	0.8678 ± 0.0056	0.8159 ± 0.0067	0.8422 ± 0.0352
Wide Deep no reg	0.8148 ± 0.0083	0.8233 ± 0.0132	0.7697 ± 0.0163	0.7966 ± 0.0173
DeepFM reg	**0.8459 ± 0.0057**	**0.8700 ± 0.0157**	**0.8252 ± 0.0029**	**0.8702 ± 0.0168**
DeepFM no reg	0.8018 ± 0.0103	0.8249 ± 0.0041	0.7682 ± 0.01	0.7739 ± 0.0021
DCN reg	**0.8489 ± 0.0091**	**0.8769 ± 0.0088**	**0.8258 ± 0.0053**	**0.8917 ± 0.0074**
DCN no reg	0.7909 ± 0.0167	0.8182 ± 0.0168	0.7501 ± 0.0164	0.7917 ± 0.0218
xDeepFM reg	**0.8403 ± 0.0146**	**0.8875 ± 0.0081**	**0.8271 ± 0.0019**	**0.8800 ± 0.0199**
xDeepFM no reg	0.8004 ± 0.0082	0.8137 ± 0.017	0.7475 ± 0.0244	0.7949 ± 0.0182
PNN reg	0.8448 ± 0.0022	0.8673 ± 0.0215	0.8287 ± 0.0147	0.8859 ± 0.0158
PNN no reg	0.8082 ± 0.0211	0.845 ± 0.0499	0.7844 ± 0.019	0.8587 ± 0.0361
RLN reg	0.6041 ± 0.0352	0.5681 ± 0.0299	0.6089 ± 0.0249	0.5906 ± 0.0351

RLN no reg	0.6013 ± 0.0264	0.5445 ± 0.0166	0.594 ± 0.0282	0.5606 ± 0.02
MLR reg	0.6216 ± 0.0466	0.5506 ± 0.0177	0.5714 ± 0.0215	0.59 ± 0.0221
MLR no reg	0.5982 ± 0.0739	0.5331 ± 0.0275	0.5526 ± 0.0298	0.5639 ± 0.0417
SNN reg	0.6903 ± 0.0131	0.6128 ± 0.0118	0.6462 ± 0.0042	0.6367 ± 0.0092
SNN no reg	0.6633 ± 0.0032	0.6211 ± 0.0227	0.6484 ± 0.0104	0.6373 ± 0.0299

*Best-performing models are in bold. reg: with regularisation cocktail; no reg: without regularisation cocktail; std: standard deviation.

The frequencies of use of individual regularisation techniques over the 14 models are shown in Figure 12. This is calculated by considering the best-performing regularisation cocktail configuration for each model and counting the number of times each regularisation method was used within the 14 cocktails. As can be seen weight decay, dropout, BN and Lookahead were the most frequently used.

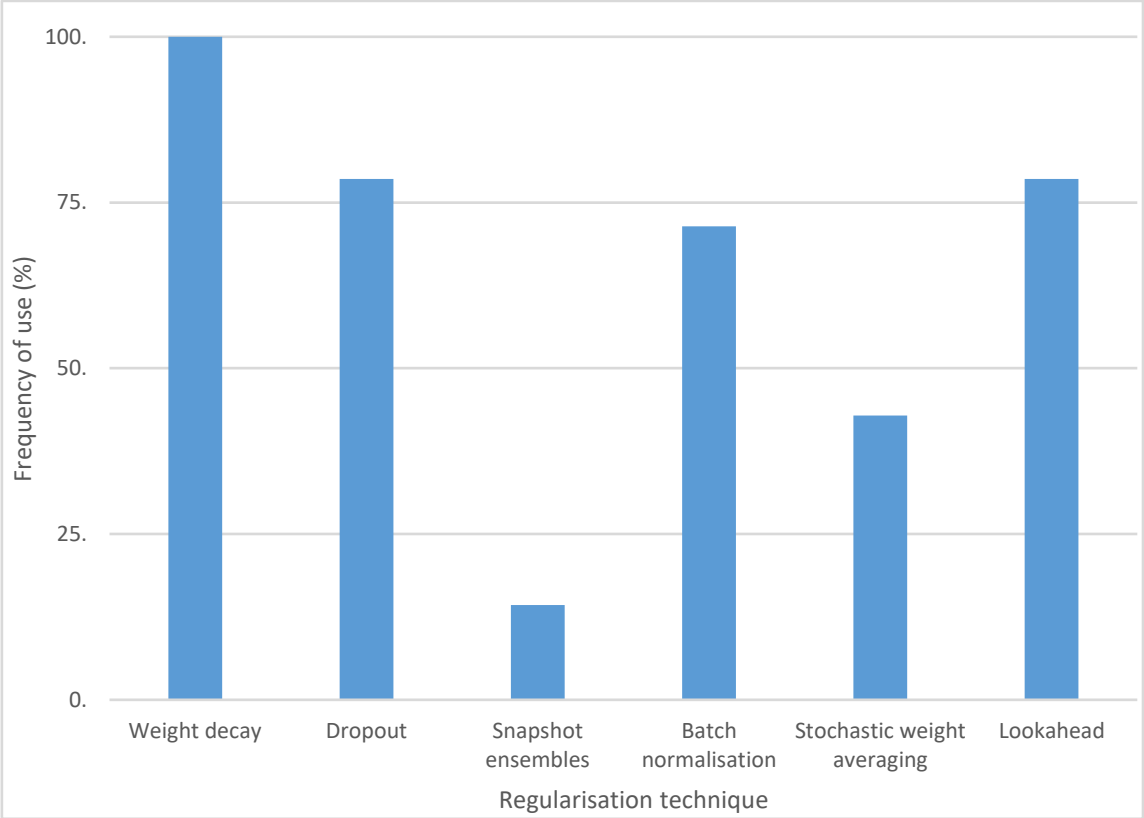

Figure 12. Frequency of use of individual regularisation cocktail ingredients.

5.3 Summary

This chapter detailed the individual results of experiments. Discussion of findings, assessment of the overall approach and joining concepts is presented in the next chapter.

Chapter 6: Discussion

This chapter discusses the findings of the study in relation to the original research aims and objectives detailed in Chapter 1, and reflects on the overall effectiveness of the approach: Sections 6.1 and 6.2 presents these for Part 1 and 2 of the study, respectively. Following this, limitations of the study are given in Section 6.3.

The results of the study addresses the research aims and objectives. The study has demonstrated a new state-of-the-art performance for synthetic tabular data generation, using TVAE with normalising flows. To the best of my knowledge, this is the first work to apply normalising flows to VAEs on tabular data. It has shown that state-of-the-art deep generative models are robust when applied to real-world datasets with imbalanced and missing data, highlighting their potential for the healthcare sector. The study has demonstrated that a handful of specialised deep learning models developed for predictive tasks on tabular data can equal and surpass the performance of state-of-the-art GBDT methods, thus allowing machine learning practitioners to maintain high performance while also leveraging the benefits of deep learning, primarily online learning and integration into multi-modal pipelines. This is especially important in the healthcare sector, where there is continuous collection of multi-modal data in the course of patient care. The study has shown that this high performance is in part due to regularisation cocktails which almost universally improves performance of all families of specialised models, and represents an advance to the state-of-the-art performance of deep learning for predictive tasks on tabular data. To the best of my knowledge, this is the first work to combine regularisation cocktails and specialised deep learning models for tabular data. However, training on imbalanced datasets substantially degrades performance, highlighting the need for caution in

selection and construction of training datasets in sectors such as healthcare where real-world data is often imperfect. Nevertheless, regularisation cocktails can still improve performance in these settings. Finally, the study has shown that models are generally robust to application on datasets with significant imbalance and missing data, suggesting practical use of well-trained models in real-world domains such as healthcare.

6.1 Part 1: Synthetic tabular data generation

TVAE outperformed CTGAN when trained on all datasets. This finding is consistent with Xu et al. (2019), where the authors found that TVAE outperforms CTGAN in the majority of cases. It can be postulated that this is because VAEs directly use data in the generative network whereas the generator of CTGAN does not use data but rather gradients flowing through the discriminator. Additionally, the discriminator and generator in a GAN are updated alternately and must be well synchronised in terms of number of updates in each, to avoid overfitting or mode collapse (Goodfellow et al., 2014). Although the performance of TVAE is superior, this does not mean that it should be the favoured solution in all circumstances. As the generator in GANs does not directly use data during training, data is not copied into the generator's parameters which has privacy benefits (Xu et al., 2019). Privacy metrics were not reported in this study, as the data was fully anonymised, but this is not always the case especially in the healthcare sector, where anonymisation can be expensive and time consuming. Another advantage of GANs is that inference and Monte Carlo Markov chain is not needed, and a wide variety of distributions can be modelled including sharp distributions, in contrast to Monte Carlo Markov chain-based inference methods which require blurry distributions for chains to mix between modes (Goodfellow et al., 2014).

With the exception of planar flows, TVAE with normalising flows outperformed the vanilla TVAE. Sylvester flows represent a generalisation of planar flows which uses a larger bottleneck and thus a single transformation becomes much more flexible in producing a complex distribution, allowing the approximate posterior to be richer, thus improving generative performance (van den Berg et al., 2018). Fewer transformations are required relative to planar flows; 5 flows were used in this study and found to work well. However, the drawback of using Sylvester flow is that it has the largest number of parameters and was slowest to train: the additional computational memory and time requirements may pose a barrier to scalability. NICE also performed well, which may in part owe to its additive coupling transformation being numerically stable as the transformation is piece-wise linear with a coupling function m that is a neural network (Dinh et al., 2014). RealNVP may have performed equally well due to affine coupling having the capacity to generate more complex distributions than additive coupling, thereby allowing the approximate posterior to be more expressive (Dinh et al., 2017). However, for both NICE and RealNVP, a subset of the latent variables remain unchanged in each transformation so more transformations are needed to allow all dimensions to influence one another. Nevertheless, 5 flows were used in this study and found to work well. Planar flows had comparable performance to vanilla TVAE in the majority of cases. It can be hypothesised that this might be due to the simplicity of the transformation. The planar flow transformation is a single unit MLP, hence it imposes a narrow bottleneck allowing distributions to change in one direction at a time only. It is noted by the authors of Rezende & Mohamed (2015) and van den Berg et al. (2018) that, because of this bottleneck, many transformations are required to attain a rich and flexible distribution, especially for high-dimensional latent spaces. However, 10, 20 and 40 flows were experimented with in this study and, while performance was maintained, no significant improvement was found; this might be because a large number of flows makes the inference network deep and difficult to train (van den Berg et al., 2018).

All generative models were robust to training on imbalanced and missing data. This supports the claims of Xu et al. (2019). One of the aims of CTGAN is to better handle discrete features with imbalanced classes, and this is achieved by introducing a conditional generator and training by sampling; the ablation study in Xu et al. (2019) showed these two design elements were critical for imbalanced datasets. The results of this study are in line with that, and this study additionally demonstrates robustness when training on datasets with missing data, which was not previously investigated.

Performance comparison with existing work: It is difficult to compare the model performances achieved in this study with the performance reported by Xu et al. (2019), as the authors only report machine learning efficacy for real datasets, and they give accuracy and F1 score for these, while this study used AUC in order to maintain consistency with Part 2 of the study. However, statistical and machine learning detection metrics in this study were objectively high and close to the maximum value for TVAE with Sylvester, NICE and RealNVP flow. The machine learning efficacy metrics achieved were comparable with intermediate performances in Part 2 of the study, and given the simplicity of the classifiers used, this is likely to be close to the best performance that can be attained on the dataset.

6.2 Part 2: Deep learning for prediction on tabular data

Feature interaction-based models performed strongest. One reason for this may be because they take into account the interactions between variables and in particular categorical variables such as gender and ethnicity which affect a number of blood parameters (e.g. haemoglobin). However, it also demonstrates that simple models ensembled are effective in elevating performance. Another key reason for superior performance may be that these models capture

high-order feature interactions, which allow better generalisation. One of the best-performing models, PNN, predominantly captures high-order interactions, and the core idea of xDeepFM is to capture explicit and implicit high-order interactions (Qu et al., 2016; Lian et al., 2018), while others such as Wide and Deep and DeepFM combine both high- and low-order interactions (Cheng et al., 2016; Guo et al., 2017). Of note is that many of the feature interaction-based models are designed for sparse and high-dimensional inputs which is not the case in this dataset, however they have been shown to work equally well on dense data: the original DCN paper demonstrated that it performs well on a range of sparse and dense datasets (Wang et al., 2017). There are disadvantages associated with these models. Wide and Deep relies on exhaustive searching of cross features for the linear component (Cheng et al., 2016), which in this dataset was not laborious due to few categorical features but could be an inefficiency with other datasets. The CIN of xDeepFM has high time complexity so might not be scalable to applications in industry (Lian et al., 2018). The inner product operation of PNNs also has high computational complexity (Qu et al., 2016).

SAINT was also among the best-performing models, and TabTransformer, while having lower performance than SAINT and the feature interaction-based models, was the highest performing of the remaining models. The reason for the high performance of SAINT is probably twofold. The first is that SAINT embeds all features – continuous and discrete – compared to TabTransformer which only embeds discrete features (Somepalli et al., 2021). Given that the majority of features in the dataset used in this study are continuous, this may have boosted performance of models. The second is that it uses intersample attention which improves performance when smaller sample sizes coexist with larger numbers of features (Somepalli et al., 2021), which is the case in the training datasets in this study. Somepalli et al. (2021) demonstrate that few datapoints receive attention; it is only those that are difficult to classify without comparison to other samples. However, as a classification task becomes more difficult

i.e. the classes are less separable, the intersample attention layer becomes less sparse, which may well be the case in this study as diagnosing Covid-19 infection on vital signs and blood tests is an inherently difficult task. The performance of TabTransformer was worse than SAINT likely because continuous features bypass the self-attention block, such that relationships between discrete and continuous features are lost and do not contribute to prediction (Huang et al., 2020; Somepalli et al., 2021).

Other models performed less well. NODE and Quantum Forest use differentiable trees, losing the automatic feature selection that GBDT benefits from, which may be the reason for lower performance (Popov et al., 2019; Chen, 2020). DNDT was the worst-performing model, and the only model with AUC < 0.7 when trained on balanced data. The reason for this is likely to be the key drawback in the design of DNDT which is that, due to the use of the Kronecker product, it does not scale well to larger numbers of features (Yang et al., 2018). In this study, the issue was addressed using the same method as suggested by Yang et al. (2018), that is training multiple models on subsets of features and then combining their predictions by majority voting. However, predictive performance is likely to have been lowered by relationships between features in different subsets being lost. These models also lose interpretability. An alternative mechanism to scale DNDT to larger numbers of features, which might be explored in future work, is to make use of the sparsity of the final binning which leads to a much smaller number of non-empty leaves than total leaves (Yang et al., 2018).

TabNet performed better in general than differential tree-based models. This may be because it has a robust selection process for salient features (Arik & Pfister, 2019): this is of paramount importance especially on datasets of small size such as the training datasets in this study. The sparse feature selection of TabNet also ensures the correct inductive bias for this particular

Covid-19 prediction task as the majority of features are redundant: only a small subset of features e.g. oxygen saturation, respiratory rate and some haematological investigations within the full blood count are important in distinguishing Covid-19 infection. This characteristic of high redundancy is also the case in many EHR datasets so TabNet has potentially wide applicability. Additionally, in this task, instance-wise feature selection is beneficial as the most critical features can differ for different patients, depending on their age and gender. For example, kidney function tests such as creatinine and eGFR in older patients might not be particularly informative as kidney function naturally declines with age, whereas in a younger patient it is more likely to be a result of Covid-19 infection. However, one explanation for TabNet having only an intermediate level of performance is that the dataset does not have a very large number of features so sparse selection might result in a degree of underfitting. RLN also had an intermediate level of performance. The reason it performs well may be for the same reasons as TabNet, which is that it is adapted to perform on datasets with features that are highly variable in relative importance (Shavitt & Segal, 2018). However, sparse feature selection might hinder learning of highly expressive models, just as is the case with TabNet.

Regularisation cocktails improved performance of deep learning predictive models almost universally. This is due to prevention of overfitting which improves generalisation performance, hence the performance improvements are greater on the 3 external validation datasets. Dropout reduces overfitting and improves generalisation as it prevents co-adaptations of units in neural networks (Srivastava et al., 2014). SE improves generalisation by combining multiple neural networks corresponding to different local minima captured during the course of optimisation (Huang et al., 2017). BN aids generalisation as datapoints are seen in conjunction with others in the batch (Ioffe & Szegedy, 2015). SWA finds wider optimal solutions and converges in the centre of optimal regions so are more robust to differences between the training and test sets,

resulting in better generalisation (Izmailov et al., 2018). Finally, Lookahead improves generalisation by performing weight averaging during training (Zhang et al., 2019).

The power of regularisation cocktails to improve performance found in this study complements existing work in the literature which have studied the potential of combining techniques, albeit on a smaller scale. It was suggested by Ioffe & Szegedy (2015) that because BN and dropout achieves similar goals, BN can reduce the need for and strength of dropout and thus speed up training while maintaining generalisation performance. Consistent with this hypothesis, this study found that BN allowed dropout to be reduced in strength: in a number of models e.g. NODE, Quantum Forest, TabNet, TabTransformer and SAINT, a low dropout rate of 0.1 was used, or 0.05 in the case of the SNN which has normalisation built into its architecture. Lookahead has also been suggested as a complementary technique to SWA by Zhang et al. (2019), who showed that the two techniques in combination led to highest accuracy during training and in the weight averaged final network. This study corroborates that as, for a number of models e.g. TabTransformer, DeepFM, DCN, xDeepFM and SNN, a combination of Lookahead and SWA led to optimal performance.

In this study, the most important regularisation cocktail ingredients were weight decay, dropout, BN and Lookahead. This is similar to the most frequently selected techniques in Kadra et al. (2021) with the exception that SE was also commonly used in that work. As in that work, there was no regularisation method or combination that was consistently optimal in this study, with regularisation cocktails being model specific. However similar to their findings, all families of regularisation techniques were important: all models benefitted from weight decay and implicit regularisation and 78% of models benefitted from model averaging, which demonstrates the utility of simultaneous application of diverse regularisation methods.

A number of these regularisation techniques also have advantages in terms of low computational overhead. SE has the same training time as training a single model (Huang et al., 2017). A key advantage of SWA is that it has almost no additional computational time or memory requirements compared to conventional SGD (Izmailov et al., 2018). Additional time is only needed to update a running average of the weights once per epoch, so computational time is similar to SGD. Memory is only needed to store a running average of weights and it is only necessary to store a single model with the average weights, so memory requirement is also similar to SGD. Lookahead is computationally efficient both in terms of time and memory, due to copying of parameters and simple arithmetic operations which are amortised over all inner loop updates, and only needing to maintain a single additional copy of model parameters (Zhang et al., 2019).

However, these techniques are not without drawbacks. Dropout increases training time. This is because training involves learning different random architectures, hence gradients that are computed are not those of the final architecture. Weight updates are noisy hence training time is increased (Srivastava et al., 2014). SWA uses tail averaging which requires a hyperparameter of when to begin averaging to be selected; this can have significant impact on performance (Zhang et al., 2019). This study used 0.5 * number of epochs i.e. weight averaging in the second half of training, which is the approach recommended by Izmailov et al. (2018). Lookahead, which performs weight averaging throughout training, avoids this problem as it can be used from the beginning of training (Zhang et al., 2019). Because Lookahead computes an exponential moving average rather than an arithmetic average (like SWA), it also helps place focus on recent weights which are more likely to be optimal.

An interesting observation was made that specialised regularisation-based architectures did not benefit from additional regularisation cocktails. One can hypothesise that this is because these are already optimised for generalisation performance, with the use of specific techniques which include normalisation and modified loss functions (Shavitt & Segal, 2018; Lounici et al., 2021, Klambauer et al., 2017).

The majority of models were not robust to training on class-imbalanced data. The reason for this is that the prediction task has increased difficulty when data is imbalanced and there are fewer training opportunities for the minority class. It becomes difficult to escape the obvious local minima which is to make a prediction consistently in favour of the majority class, as in the vast majority of cases, this is the correct prediction and results in lowest loss. This is a significant problem that many have grappled with on healthcare datasets, which are typically imbalanced against the class of interest due to low prevalence of diseases and conditions (Rahman & Davis, 2013). In this study, the issue was dealt with by under-sampling the majority class but in real-world applications this may not be desirable as it reduces the volume of training data making sample sizes insufficient. Additionally, there arises the problem of bias in the sampling process. Ideally, in the healthcare domain, there is a need for matched sampling so that the distribution of patients in the majority class is not skewed by under-sampling. However, this can be extremely challenging to ensure for all relevant features. For this reason, better methods for data augmentation such as synthetic data generation is needed, hence the importance of Part 1 of this study.

The robustness of the feature interaction-based models for imbalanced training data is probably due to their ability to model high-order interactions and generalise to even rare or unseen feature interactions. The reason for robustness of SAINT may be due to intersample attention which

leverages other similar samples in the data to help classification (Somepalli et al., 2021) - in this task, Covid-19 infection is associated with a very specific profile in a few blood tests which is consistently present across multiple samples.

For some models e.g. NODE and TabTransformer, regularisation cocktails appeared to improve robustness to imbalanced training data. However, this effect was not consistent and is likely due to the use of a cyclical learning rate in some of the regularisation cocktail ingredients e.g. SWA (Loshchilov & Hutter, 2017), which can perturb the model to escape the obvious local minima.

All models were relatively robust when tested on datasets with imbalanced and missing data. There are advantages to this as it means that, as long as a training dataset is carefully constructed and a model is trained well, it can be deployed on diverse datasets, which is beneficial to practical real-world applications where a model needs to be used on imperfect data.

Performance comparison with existing work: It is difficult to directly compare performance of the models on the dataset in this study with that reported in their original papers on open source datasets, due to the difference in nature of tasks e.g. binary or multiclass classification and task difficulty. Chen (2020) reported an accuracy in the range of 20 – 40% for Quantum Forest and, although accuracy is not equivalent to AUC, this study achieved significantly higher performance. DNDT accuracy was reported by Yang et al. (2018) to be largely in the range of 70 – 90%, TabNet AUC was largely in the range 0.7 – 0.9 (Arik & Pfister, 2019), TabTransformer AUC was largely in the range 0.7 – 0.9 (Huang et al., 2020) and SAINT AUC was largely in the range 0.85 – 0.95 (Somepalli et al., 2021). In all cases, this study achieved comparable results. In Cheng et al. (2016) and Qu et al. (2016), Wide and Deep and PNN

achieved AUC of around 0.7 and 0.75 – 0.8, respectively; in comparison, this study achieved much higher performance (AUC increment of 0.15 and 0.1, respectively). This study also achieved higher performance on xDeepFM than Lian et al. (2018) who reported AUC of 0.8 – 0.85, and a comparable result on DeepFM which achieved AUC of 0.8 – 0.9 (Guo et al., 2017). However for MLR and SNN, the AUC in this study was lower than in Lounici et al. (2021) and Klambauer et al. (2017) (AUC of 0.9 and 0.8 – 0.85, respectively).

6.3 Limitations

Despite demonstrated performance improvements, the approach in this study is not without limitation. In comparison to shallow methods such as GBDT, deep learning itself has higher computational requirements and requires more extensive tuning of data-dependent hyperparameters. Thus, using deep learning for predictive tasks on tabular data is more time consuming and requires more expertise (Shwartz-Ziv & Armon, 2021). Therefore, in many practical applications, shallow methods such as GBDT might still be preferred. However, as we have enumerated, deep learning has several advantages such as online learning and integration in multi-modal pipelines which are equally important for practical use, arguably outweighing these drawbacks. Regularisation cocktails requires even further hyperparameter optimisation (Kadra et al., 2021). However, some of the regularisation techniques used have been demonstrated in existing work to be robust to hyperparameter changes so do not require significant tuning: for example, Lookahead is robust to changes in the inner loop optimiser, ratio of fast to slow weight updates and the slow weight learning rate (Zhang et al., 2019).

Another limitation of this work is that training datasets for Part 2 of the study did not have missing data. This was due to the constraints of the datasets available. The main Covid-19

pandemic dataset was the Portsmouth dataset (this had the largest number of Covid-19 positive cases) but it had low missing data hence it would not have been informative to train on this. It was possible to train on a different pandemic dataset with more missing data e.g. Bedford dataset, but this had much fewer number of Covid-19 positive cases and would have necessitated changing the training and external validation datasets, which would have prevented the results from being directly comparable. This study also only investigated classification and not regression and did not study datasets with extreme outliers or missing labels, among other special cases.

Chapter 7: Conclusion

This chapter summarises the contributions of the study (Section 7.1) and advances suggestion of future work (Section 7.2).

7.1 Contributions

The study made the following discoveries:

1. TVAE with normalising flows improves the current state-of-the-art performance of deep generative models on tabular data.
2. Deep generative models are robust to real-world datasets with imbalanced and missing data, highlighting their potential for the healthcare sector.
3. Specialised deep learning models developed for predictive tasks on tabular data can equal and surpass the performance of GBDT, enabling the combination of high performance, multi-modal pipelines and online learning, which is key in approximating clinician-like decision-making processes in the healthcare sector.
4. Regularisation cocktails almost universally improve performance of all families of specialised architectures, and advances the state-of-the-art performance of deep learning models for predictive tasks on tabular data.
5. Deep learning predictive models are generally not robust to datasets with imbalanced data, highlighting the need for caution in selection and construction of training datasets in the healthcare domain.
6. Regularisation cocktails can still improve performance in the setting of imbalanced training datasets.

7. The full pipeline of synthetic tabular data generation for imbalanced training data augmentation and missing data imputation, combined with specialised deep learning architectures and regularisation cocktails, may be an effective approach to rapidly screen for Covid-19 infection in hospitals, improving treatment decisions and infection control, without the significant cost of Covid-19 testing.

7.2 Future work

Possible future work to build on this study include evaluating deep learning generative models on privacy metrics. This is important in the healthcare domain, where fully anonymising datasets is often challenging and time consuming. It is possible that CTGAN would result in better performance than TVAEs in this respect (Xu et al., 2019).

The deep learning predictive models studied could also be trained on datasets with significant missing data to evaluate performance. This has only been sporadically studied, predominantly on SAINT and TabTransformer, which were both found to be robust to missing data, SAINT because it can borrow feature values from other similar datapoints in the dataset and TabTransformer due to contextual embeddings which can be flexible in how it draws information from features (Somepalli et al., 2021; Huang et al., 2020). However, robustness of other models to missingness in training data is not well understood. Similarly, models can be tested on training data of more varied quality, for example data with extreme outliers or missing labels. Further regularisation techniques such as data augmentation e.g. Cut-Out, Mix Up, Cut-Mix, FGSM Adversarial Learning, and structural regularisation and linearisation e.g. skip connections, Shake-Shake and Shake-Drop could be included in cocktails (Kadra et al., 2021), for which there was not sufficient time to explore in this study. A final avenue which can be explored is the applicability of these models to semi-supervised learning.

Appendices

A Hyperparameter search

This section gives the hyperparameter search spaces for each model in the study.

Table A.1. CTGAN hyperparameters search space.

Hidden dimension	{120, 250, 500}
Latent dimension	{16, 32, 64, 128}

Table A.2. TVAE hyperparameters search space.

Hidden dimension	{128, 256, 512}
Latent dimension	{16, 32, 64, 128}

Table A.3. Sylvester flow hyperparameters search space.

Num orthogonal vectors M	{2, 4, 8}

Table A.4. NICE flow hyperparameters search space.

Num layers	UniformInt[2,6]
Hidden dimension	{40, 80, 160}

Table A.5. RealNVP flow hyperparameters search space.

Num layers	UniformInt[2,6]
Hidden dimension	{40, 80, 160}

Table A.6. NODE hyperparameters search space.

Num layers	UniformInt[1, 8]
Num total trees	{512, 1024, 2048}
Tree depth	UniformInt[4, 8]
Tree output dimension	UniformInt[1, 5]
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.7. Quantum Forest hyperparameters search space.

Num layers	UniformInt[1, 8]
Num total trees	{512, 1024, 2048}
Tree depth	UniformInt[4, 8]
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.8. DNDT hyperparameters search space.

Num cut points per feature	{1, 2}
Temperature τ	LogUniform[$1e^{-3}$, $1e^{-1}$]
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]

Table A.9. TabNet hyperparameters search space.

N_d	{16, 32, 64, 128, 256}
N_a	{16, 32, 64, 128, 256}
Relaxation factor γ	{1, 1.2, 1.5, 2}
Num steps N_{steps}	UniformInt[3, 10]
Sparsity coefficient λ_{sparse}	LogUniform[$1e^{-6}$, $1e^{-1}$]
Momentum m_B	Uniform[0.7, 0.95]

Virtual batch size B	{32, 64, 128, 256}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256, 512, 1024}

Table A.10. TabTransformer hyperparameters search space.

Embedding dimension	{8, 16, 32, 64, 128, 256}
Num layers N	{1, 2, 3, 6, 12}
Num attention heads	{2, 4, 8}
MLP first layer hidden dimension	{m*l, $m \in \mathbb{Z} \mid 1 \leq m \leq 8$}
MLP second layer hidden dimension	{m*l, $m \in \mathbb{Z} \mid 1 \leq m \leq 3$}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.11. SAINT hyperparameters search space.

Embedding dimension	{8, 16, 32, 64, 128, 256}
Num layers L	{1, 2, 3, 6, 12}
Num attention heads	{2, 4, 8}
MLP first layer hidden dimension	{m*l, $m \in \mathbb{Z} \mid 1 \leq m \leq 8$}
MLP second layer hidden dimension	{m*l, $m \in \mathbb{Z} \mid 1 \leq m \leq 3$}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.12. Wide and Deep hyperparameters search space.

Embedding dimension	{4, 8, 16, 32, 64}
Deep neural network num layers	UniformInt[2, 5]
Deep neural network hidden dimension	{64, 128, 256, 512, 1024}

Deep neural network shape	{rectangular, conical}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.13. DeepFM hyperparameters search space.

Embedding dimension	{4, 8, 16, 32, 64}
Deep neural network num layers	UniformInt[2, 5]
Deep neural network hidden dimension	{100, 200, 400, 800}
Deep neural network shape	{rectangular, conical}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.14. DCN hyperparameters search space.

Embedding dimension	{4, 8, 16, 32, 64}
Deep neural network num layers	UniformInt[2, 5]
Deep neural network hidden dimension	{64, 128, 256, 512, 1024}
Deep neural network shape	{rectangular, conical}
Num cross layers	UniformInt[3, 10]
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.15. xDeepFM hyperparameters search space.

Embedding dimension	{4, 8, 16, 32, 64}
Deep neural network num layers	UniformInt[2, 5]
Deep neural network hidden dimension	{100, 200, 400, 800}
Deep neural network shape	{rectangular, conical}

Num CIN layers	UniformInt[2, 5]
CIN hidden dimension	{100, 200, 400, 800}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.16. PNN hyperparameters search space.

Embedding dimension	{4, 8, 16, 32, 64}
Deep neural network num layers	UniformInt[2, 5]
Deep neural network hidden dimension	{100, 200, 400, 800}
Deep neural network shape	{rectangular, conical}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.17. RLN hyperparameters search space.

Num layers	UniformInt[2, 5]
Regularisation coefficients learning rate ν	LogUniform[$1e^{5}$, $1e^{7}$]
Normalisation factor θ	UniformInt[-6, -14]
Weights learning rate η	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.18. MLR hyperparameters search space.

Num layers	UniformInt[2, 5]
Hidden dimension	{256, 512, 1024, 2048}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.19. SNN hyperparameters search space.

Num layers	{2, 4, 8, 16, 32}
Hidden dimension	{128, 256, 512, 1024}
Network shape	{rectangular, conical}
Learning rate	LogUniform[$1e^{-5}$, $1e^{-1}$]
Batch size	{128, 256}

Table A.20. Weight decay / L2 regularisation hyperparameters search space.

Weight decay / L2 regularisation	LogUniform[$1e^{-5}$, $1e^{-1}$]

Table A.21. Dropout hyperparameters search space.

Dropout rate p	Uniform[0.05, 0.8]

Table A.22. SE hyperparameters search space.

Initial learning rate α_0	{0.1, 0.2}
Num epochs T	100
Num snapshots M	UniformInt[4, 6]

Table A.23. SWA hyperparameters search space.

Initial learning rate α_1	LogUniform[$1e^{-3}$, $1e^{-1}$]
Averaging period c	5
Start averaging epoch	0.5 * num epochs

Table A.24. Lookahead hyperparameters search space.

Fast weights num steps k	{5, 10}
Slow weights step size α	{0.5, 0.8}

B Source code

For code used in this study, see the accompanying file to this dissertation.

Bibliography

1. Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., ... & Zhu, Z. (2016). Deep Speech 2: End-to-end speech recognition in English and Mandarin. In *International Conference on Machine Learning*, PMLR (pp. 173-182).
2. Arık, S. O., & Pfister, T. (2019). TabNet: attentive interpretable tabular learning. *arXiv preprint arXiv:1908.07442*.
3. Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein generative adversarial networks. In *International Conference on Machine Learning*, PMLR (pp. 214-223).
4. Aviñó, L., Ruffini, M., & Gavaldà, R. (2018). Generating synthetic but plausible healthcare record datasets. *arXiv preprint arXiv:1807.01514*.
5. Bae, H., Jung, D., Choi, H. S., & Yoon, S. (2019). AnomiGAN: Generative adversarial networks for anonymizing private medical data. In *Pacific Symposium on Biocomputing 2020* (pp. 563-574).
6. Blondel, M., Fujino, A., Ueda, N., & Ishihata, M. (2016). Higher-order factorization machines. In *Advances in Neural Information Processing Systems* (pp. 3351-3359).
7. Bradley, A. P. (1997). The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, *30*(7), 1145-1159.
8. Bronsert, M., Singh, A. B., Henderson, W. G., Hammermeister, K., Meguid, R. A., & Colborn, K. L. (2020). Identification of postoperative complications using electronic health record data and machine learning. *The American Journal of Surgery*, *220*(1), 114-119.
9. Camino, R. D., Hammerschmidt, C. A., & State, R. (2019). Improving missing data imputation with deep generative models. *arXiv preprint arXiv:1902.10666*.
10. Che, Z., Cheng, Y., Zhai, S., Sun, Z., & Liu, Y. (2017). Boosting deep learning risk prediction with generative adversarial networks for electronic health records. In *2017 IEEE International Conference on Data Mining (ICDM)*, IEEE (pp. 787-792).

11. Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785-794).
12. Chen, Y. (2020). Deep differentiable forest with sparse attention for the tabular data. *arXiv preprint arXiv:2003.00223*.
13. Cheng, H. T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., ... & Shah, H. (2016). Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems* (pp. 7-10).
14. Choi, E., Biswal, S., Malin, B., Duke, J., Stewart, W. F., & Sun, J. (2017). Generating multi-label discrete patient records using generative adversarial networks. In *Machine Learning for Healthcare Conference*, PMLR (pp. 286-305).
15. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.
16. Dinh, L., Krueger, D., & Bengio, Y. (2014). NICE: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*.
17. Dinh, L., Sohl-Dickstein, J., & Bengio, S. (2017). Density estimation using Real NVP. *arXiv preprint arXiv:1605.08803*.
18. Dinnes, J., Deeks, J. J., Berhane, S., Taylor, M., Adriano, A., Davenport, C., ... & Cochrane COVID-19 Diagnostic Test Accuracy Group. (2021). Rapid, point-of-care antigen and molecular-based tests for diagnosis of SARS-CoV-2 infection. *Cochrane Database of Systematic Reviews*, (3).
19. Dong, W., Fong, D. Y. T., Yoon, J. S., Wan, E. Y. F., Bedford, L. E., Tang, E. H. M., & Lam, C. L. K. (2021). Generative adversarial networks for imputing missing data for big data clinical research. *BMC medical research methodology*, *21*(1), 1-10.
20. Feng, J., Yu, Y., & Zhou, Z. H. (2018). Multi-layered gradient boosting decision trees. *arXiv preprint arXiv:1806.00007*.

21. Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 1189-1232.
22. Gershman, S., & Goodman, N. (2014). Amortized inference in probabilistic reasoning. In *Proceedings of the Annual Meeting of the Cognitive Science Society* (Vol. 36, No. 36).
23. Gershman, S., Hoffman, M., & Blei, D. (2012). Nonparametric variational inference. *arXiv preprint arXiv:1206.4665*.
24. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, JMLR Workshop and Conference Proceedings (pp. 249-256).
25. Golas, S. B., Shibahara, T., Agboola, S., Otaki, H., Sato, J., Nakae, T., ... & Jethwani, K. (2018). A machine learning model to predict the risk of 30-day readmissions in patients with heart failure: a retrospective analysis of electronic medical records data. *BMC medical informatics and decision making*, *18*(1), 1-17.
26. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT press.
27. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems*, *27*.
28. Gorishniy, Y., Rubachev, I., Khrulkov, V., & Babenko, A. (2021). Revisiting Deep Learning Models for Tabular Data. *arXiv preprint arXiv:2106.11959*.
29. Gregor, K., Danihelka, I., Graves, A., Rezende, D., & Wierstra, D. (2015). Draw: A recurrent neural network for image generation. In *International Conference on Machine Learning*, PMLR (pp. 1462-1471).
30. Gregor, K., Danihelka, I., Mnih, A., Blundell, C., & Wierstra, D. (2014). Deep autoregressive networks. In *International Conference on Machine Learning*, PMLR (pp. 1242-1250).

31. Guo, H., Tang, R., Ye, Y., Li, Z., & He, X. (2017). DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247*.
32. Hammad Alharbi, H., & Kimura, M. (2020). Missing data imputation using data generated by GAN. In *2020 the 3rd International Conference on Computing and Big Data* (pp. 73-77).
33. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770-778).
34. Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., ... & Zhou, Y. (2017). Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*.
35. Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., ... & Lerchner, A. (2017). beta-VAE: Learning basic visual concepts with a constrained variational framework.
36. Ho, J., Kalchbrenner, N., Weissenborn, D., & Salimans, T. (2019). Axial attention in multidimensional transformers. *arXiv preprint arXiv:1912.12180*.
37. Hoffman, M. D., Blei, D. M., Wang, C., & Paisley, J. (2013). Stochastic variational inference. *Journal of Machine Learning Research*, *14*(5).
38. Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., & Weinberger, K. Q. (2017). Snapshot ensembles: Train 1, get M for free. *arXiv preprint arXiv:1704.00109*.
39. Huang, X., Khetan, A., Cvitkovic, M., & Karnin, Z. (2020). Tabtransformer: Tabular data modeling using contextual embeddings. *arXiv preprint arXiv:2012.06678*.
40. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, PMLR (pp. 448-456).

41. Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.
42. Jaakkola, T. S., & Jordan, M. I. (1998). Improving the mean field approximation via the use of mixture distributions. In *Learning in Graphical Models* (pp. 163-173). Springer, Dordrecht.
43. Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., & Saul, L. K. (1999). An introduction to variational methods for graphical models. *Machine Learning*, *37*(2), 183-233.
44. Kadra, A., Lindauer, M., Hutter, F., & Grabocka, J. (2021). Regularization is all you Need: Simple Neural Nets can Excel on Tabular Data. *arXiv preprint arXiv:2106.11189*.
45. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T. Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, *30*, 3146-3154.
46. Ke, G., Zhang, J., Xu, Z., Bian, J., & Liu, T. Y. (2018). TabNN: A universal neural network solution for tabular data.
47. Keras. (n.d.). *Keras*. Retrieved May 10, 2021, from ***https://keras.io/***.
48. Kingma, D. P., & Welling, M. (2014). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.
49. Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017). Self-normalizing neural networks. In *Proceedings of the 31st international conference on neural information processing systems* (pp. 972-981).
50. Kogan, E., Twyman, K., Heap, J., Milentijevic, D., Lin, J. H., & Alberts, M. (2020). Assessing stroke severity using electronic health record data: a machine learning approach. *BMC medical informatics and decision making*, *20*(1), 1-8.
51. Lai, S., Xu, L., Liu, K., & Zhao, J. (2015). Recurrent convolutional neural networks for text classification. In *29th AAAI conference on Artificial Intelligence*.

52. Lay, N., Harrison, A. P., Schreiber, S., Dawer, G., & Barbu, A. (2018). Random hinge forest for differentiable learning. *arXiv preprint arXiv:1802.03882*.
53. Lian, J., Zhou, X., Zhang, F., Chen, Z., Xie, X., & Sun, G. (2018). xDeepFM: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 1754-1763).
54. Loshchilov, I., & Hutter, F. (2017). SGDR: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.
55. Lounici, K., Meziani, K., & Riu, B. (2021). Muddling Label Regularization: Deep Learning for Tabular Datasets. *arXiv preprint arXiv:2106.04462*.
56. Lundberg, S. M., Erion, G. G., & Lee, S. I. (2018). Consistent individualized feature attribution for tree ensembles. *arXiv preprint arXiv:1802.03888*.
57. Luz, C. F., Vollmer, M., Decruyenaere, J., Nijsten, M. W., Glasner, C., & Sinha, B. (2020). Machine learning in infection management using routine electronic health records: tools, techniques, and reporting of future technologies. *Clinical Microbiology and Infection*, *26*(10), 1291-1299.
58. Mandair, D., Tiwari, P., Simon, S., Colborn, K. L., & Rosenberg, M. A. (2020). Prediction of incident myocardial infarction using machine learning applied to harmonized electronic health record data. *BMC medical informatics and decision making*, *20*(1), 1-10.
59. Martinez, D. A., Levin, S. R., Klein, E. Y., Parikh, C. R., Menez, S., Taylor, R. A., & Hinson, J. S. (2020). Early prediction of acute kidney injury in the emergency department with machine-learning methods applied to electronic health record data. *Annals of emergency medicine*, *76*(4), 501-514.
60. Miller, K., Hettinger, C., Humpherys, J., Jarvis, T., & Kartchner, D. (2017). Forward thinking: Building deep random forests. *arXiv preprint arXiv:1705.07366*.

61. Mnih, A., & Gregor, K. (2014). Neural variational inference and learning in belief networks. In *International Conference on Machine Learning*, PMLR (pp. 1791-1799).
62. Nalisnick, E., Hertel, L., & Smyth, P. (2016). Approximate inference for deep latent gaussian mixtures. In *NIPS Workshop on Bayesian Deep Learning* (Vol. 2, p. 131).
63. Patki, N., Wedge, R., & Veeramachaneni, K. (2016). The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, IEEE (pp. 399-410).
64. Pereira, R. C., Santos, M. S., Rodrigues, P. P., & Abreu, P. H. (2020). Reviewing Autoencoders for Missing Data Imputation: Technical Trends, Applications and Outcomes. *Journal of Artificial Intelligence Research*, *69*, 1255-1285.
65. Petrilli, C. M., Jones, S. A., Yang, J., Rajagopalan, H., O'Donnell, L., Chernyak, Y., ... & Horwitz, L. I. (2020). Factors associated with hospital admission and critical illness among 5279 people with coronavirus disease 2019 in New York City: prospective cohort study. *BMJ*, *369*.
66. Popov, S., Morozov, S., & Babenko, A. (2019). Neural oblivious decision ensembles for deep learning on tabular data. *arXiv preprint arXiv:1909.06312*.
67. Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2017). CatBoost: unbiased boosting with categorical features. *arXiv preprint arXiv:1706.09516*.
68. PyTorch. (n.d.). *PyTorch*. Retrieved May 10, 2021, from ***https://pytorch.org/***.
69. Qu, Y., Cai, H., Ren, K., Zhang, W., Yu, Y., Wen, Y., & Wang, J. (2016). Product-based neural networks for user response prediction. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, IEEE (pp. 1149-1154).
70. Rahman, M. M., & Davis, D. N. (2013). Addressing the class imbalance problem in medical datasets. *International Journal of Machine Learning and Computing*, *3*(2), 224.

71. Rainforth, T., Kosiorek, A., Le, T. A., Maddison, C., Igl, M., Wood, F., & Teh, Y. W. (2018). Tighter variational bounds are not necessarily better. In *International Conference on Machine Learning*, PMLR (pp. 4277-4285).
72. Rendle, S. (2010). Factorization machines. In *2010 IEEE International Conference on Data Mining*, IEEE (pp. 995-1000).
73. Rendle, S. (2012). Factorization machines with libFM. *ACM Transactions on Intelligent Systems and Technology (TIST)*, *3*(3), 1-22.
74. Rezende, D., & Mohamed, S. (2015). Variational inference with normalizing flows. In *International Conference on Machine Learning*, PMLR (pp. 1530-1538).
75. Rezende, D. J., Mohamed, S., & Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, PMLR (pp. 1278-1286).
76. Salimans, T., Kingma, D., & Welling, M. (2015). Markov chain Monte Carlo and variational inference: Bridging the gap. In *International Conference on Machine Learning*, PMLR (pp. 1218-1226).
77. Saxe, A. M., McClelland, J. L., & Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*.
78. SDV. (n.d.). *Single Table Metrics.* Retrieved May 10, 2021, from https://sdv.dev/SDV/user_guides/evaluation/single_table_metrics.html.
79. Shavitt, I., & Segal, E. (2018). Regularization learning networks: deep learning for tabular datasets. *arXiv preprint arXiv:1805.06440*.
80. Shimodaira, H. (2000). Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference*, *90*(2), 227-244.
81. Shwartz-Ziv, R., & Armon, A. (2021). Tabular Data: Deep Learning is Not All You Need. *arXiv preprint arXiv:2106.03253*.

82. Soltan, A. A., Kouchaki, S., Zhu, T., Kiyasseh, D., Taylor, T., Hussain, Z. B., ... & Clifton, D. A. (2021). Rapid triage for COVID-19 using routine clinical data for patients attending hospital: development and prospective validation of an artificial intelligence screening test. The Lancet Digital Health, 3(2), e78-e87.
83. Somepalli, G., Goldblum, M., Schwarzschild, A., Bruss, C. B., & Goldstein, T. (2021). SAINT: Improved Neural Networks for Tabular Data via Row Attention and Contrastive Pre-Training. *arXiv preprint arXiv:2106.01342*.
84. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929-1958.
85. Sun, Y., Cuesta-Infante, A., & Veeramachaneni, K. (2019). Learning vine copula models for synthetic data generation. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 33, No. 01, pp. 5049-5057).
86. Telenti, A., Arvin, A., Corey, L., Corti, D., Diamond, M. S., García-Sastre, A., ... & Virgin, H. W. (2021). After the pandemic: perspectives on the future trajectory of COVID-19. *Nature*, 1-14.
87. TensorFlow. (n.d.). *TensorFlow.* Retrieved May 10, 2021, from https://www.tensorflow.org/.
88. Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, *58*(1), 267-288.
89. Tikhonov, A. N. (1943). On the stability of inverse problems. In *Dokl. Akad. Nauk SSSR* (Vol. 39, pp. 195-198).
90. Titsias, M., & Lázaro-Gredilla, M. (2014). Doubly stochastic variational Bayes for non-conjugate inference. In *International Conference on Machine Learning*, PMLR (pp. 1971-1979).
91. Tran, D., Ranganath, R., & Blei, D. M. (2015). The variational Gaussian process. *arXiv preprint arXiv:1511.06499*.

92. Turner, R.E., & Sahani, M. (2011). Two problems with variational expectation maximisation for time-series models. In Barber, D., Cemgil, A. T., & Chiappa, S. (Eds.). (2011). *Bayesian time series models*. Chapter 5, pp. 109–130. Cambridge University Press.
93. van den Berg, R., Hasenclever, L., Tomczak, J. M., & Welling, M. (2018). Sylvester normalizing flows for variational inference. *arXiv preprint arXiv:1803.05649*.
94. van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., ... & Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.
95. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (pp. 5998-6008).
96. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2017). Graph attention networks. *arXiv preprint arXiv:1710.10903*.
97. Wang, R., Fu, B., Fu, G., & Wang, M. (2017). Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17* (pp. 1-7).
98. Wong, A., Young, A. T., Liang, A. S., Gonzales, R., Douglas, V. C., & Hadley, D. (2018). Development and validation of an electronic health record–based machine learning model to estimate delirium risk in newly hospitalized patients without known cognitive impairment. *JAMA network open*, *1*(4), e181018-e181018.
99. Wong, J., Horwitz, M. M., Zhou, L., & Toh, S. (2018). Using machine learning to identify health outcomes from electronic health record data. *Current epidemiology reports*, *5*(4), 331-342.
100. Wynants, L., Van Calster, B., Collins, G. S., Riley, R. D., Heinze, G., Schuit, E., ... & van Smeden, M. (2020). Prediction models for diagnosis and prognosis of covid-19: systematic review and critical appraisal. *BMJ*, *369*.

101. Xiao, J., Ye, H., He, X., Zhang, H., Wu, F., & Chua, T. S. (2017). Attentional factorization machines: Learning the weight of feature interactions via attention networks. *arXiv preprint arXiv:1708.04617*.

102. Xu, L., Skoularidou, M., Cuesta-Infante, A., & Veeramachaneni, K. (2019). Modeling tabular data using conditional GAN. *arXiv preprint arXiv:1907.00503*.

103. Xu, L., & Veeramachaneni, K. (2018). Synthesizing tabular data using generative adversarial networks. *arXiv preprint arXiv:1811.11264*.

104. Yang, Y., Morillo, I. G., & Hospedales, T. M. (2018). Deep neural decision trees. *arXiv preprint arXiv:1806.06988*.

105. Yoon, J., Drumright, L. N., & van der Schaar, M. (2020). Anonymization through data synthesis using generative adversarial networks (ADS-GAN). *IEEE journal of biomedical and health informatics*, *24*(8), 2378-2388.

106. Yoon, J., Jordon, J., & van der Schaar, M. (2018). GAIN: Missing data imputation using generative adversarial nets. In *International Conference on Machine Learning*, PMLR (pp. 5689-5698).

107. Zhang, J., Cormode, G., Procopiuc, C. M., Srivastava, D., & Xiao, X. (2017). Privbayes: Private data release via bayesian networks. *ACM Transactions on Database Systems (TODS)*, *42*(4), 1-41.

108. Zhang, M. R., Lucas, J., Hinton, G., & Ba, J. (2019). Lookahead optimizer: k steps forward, 1 step back. *arXiv preprint arXiv:1907.08610*.

109. Zheng, T., Xie, W., Xu, L., He, X., Zhang, Y., You, M., ... & Chen, Y. (2017). A machine learning-based framework to identify type 2 diabetes through electronic health records. *International journal of medical informatics*, *97*, 120-127.

110. Zhou, Z. H., & Feng, J. (2017). Deep forest. *arXiv preprint arXiv:1702.08835*.