Feature Engineering for Click Prediction

Fernando Ivans Rita Lourenço

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Examination Committee

Chairperson: Prof. Miguel Correia

Supervisor: Prof. Diogo R. Ferreira

Member of the Committee: Paulo Carreira

October 2020

Resumo

No mundo de anúncios online a tarefa de Previsão de clicks envolve, analisar diferentes atributos de modo a concluir que anúncios apresentam maior probabilidade de serem clicados. Os atributos nem sempre partilham do mesmo tipo, estes dividem-se essencialmente em dois, numéricos e categóricos. Em geral, estes atributos, devem ser previamente processados de modo a serem usados como entrada para o nosso modelo de previsão. Métodos estes, como por exemplo: ”one hot encoding ” ou ”label encoding” para atributos categóricos; normalização caso os mesmos sejam numéricos. Adicionalmente, estes devem ser combinados de modo a gerar novos atributos com intuito de melhorar a precisão do nosso algoritmo, processo este denominado extração de atributos (Feature Engineering). Nos últimos anos, muitos modelos foram desenvolvidos com a habilidae de efectuar este processo de forma automatizada, isto é, gerando combinações de características. Neste trabalho, iremos aplicar e estudar diferentes modelos em conjuntos de dados reais. Sendo que, ao avaliar a performance destes algoritmos conseguiremos inferir sobre que operações são responsáveis pelo do seu sucesso.

Palavras-chave: Aprendizado-Máquina, Aprendizagem profunda, Extração de Atributos (Feature Engineering)

Abstract

Click prediction in online advertising involves analyzing different features in order to estimate the probability that a certain advertisement will be clicked. These features can be of different types, but essentially they can be divided into numerical and categorical. In general, these features must be preprocessed in order to serve as input in a prediction model as one hot vectors or label encoded in case of being categorical, normalized in case of being numerical dense features. In addition these features must be combined in order to generate new features that can improve the accuracy of the predictions, process called Feature engineering. In recent years, several models have been developed to perform these kind of feature engineering automatically by generating combinations of raw features. In this work, we apply and compare several different models for this purpose, and evaluate them on a real-world dataset. Based on these results, we identify the most promising techniques for ad click prediction and explain why they are more successful.

Keywords: Click-Through Rate, Conversion Rate, Feature Engineering, Deep Learning

Contents

Resumo . . . iii
Abstract . . . v
List of Tables . . . ix
List of Figures . . . xi
Nomenclature . . . 1
Glossary . . . 1

1 Introduction **1**
1.1 Digital advertising models . . . 2
1.2 Deep learning concepts . . . 3
1.3 Objectives . . . 3
1.4 Outline . . . 4

2 Related Work **5**
2.1 Convolutional Network . . . 5
2.2 Deep and Cross Network . . . 6
2.3 Factorization based . . . 8
2.3.1 Deep Factorization Machine . . . 8
2.3.2 Deep Learning over Multi-Categorical Data . . . 9
2.3.3 Sparse Factorization Machines . . . 12
2.4 Atention Based Networks . . . 12
2.4.1 Deep Interest Network . . . 12
2.4.2 AutoInt . . . 13

3 Proposed Approach **15**
3.1 Data sets . . . 15
3.2 Models . . . 16
3.3 Evaluation . . . 17
3.4 Discussion . . . 18

4 Implementation **19**
4.1 Deep Factorization Machine . . . 19

4.2 Deep & Cross Network 21
4.3 AutoInt 22
4.4 Discussion 23

5 Case Studies **25**
5.1 Criteo 25
5.2 Avazu 26
5.3 Discussion 27

6 Conclusions **29**
6.1 Contributions 29
6.2 Future Work 30

Bibliography **31**

A Mathematical Definitions **33**
A.1 Softmax 33
A.2 bag of words 33
A.3 Confusion Matrix 33

List of Tables

2.1 Best logloss from different models from [3] 7
2.2 Best log-loss achieved with various memory budgets from [3] 7
2.3 Performance evaluation using AUC and log-loss from [5] 9
2.4 DeepFM models improvement over Wide and Deep Components, from [5] 9
2.5 Logloss results for the proposed solutions compared to FM and LR, from [9] 11

List of Figures

2.1 CCPM structure from [2] 6
2.2 The Deep & Cross Network from [3] 7
2.3 The structure of the DeepFM framework in the left, DNN component structure in the up rightmost side, PNN structure at the down rightmost side, from [5] 8
2.4 The FNN model structure, from [9] 11
2.5 The FNN model structure, from [9] 11
2.6 Feature representation and statistics, from [15] 13
2.7 AutoInt model structure 13
2.8 Interaction layer structure 14

3.1 Evaluation pipeline 15
3.2 Confusion matrix 17

5.1 Example of function Pandas DataFrame Describe output 26
5.2 Criteo loss and validation loss, respectively - plotted for selected models 26
5.3 Avazu loss and validation loss, respectively - plotted for selected models 27

A.1 Confusion matrix 34

Chapter 1

Introduction

In many world applications such as recommender systems and digital advertising, click-through rate (CTR) prediction plays an important role. In recommender systems for e-commerce web sites such as Amazon, where the goal of using CTR is to increase both revenue and user satisfaction hence fidelity to the service, by selling more items and improve user experience in the platform, through the study of users interest and suggest products that relate to it, boosting the probability that a suggested item is bought, maximizing CTR to predict the probability of it being clicked hence belonging to the users interest. Digital advertising is a market which has been growing at an exponential rate. Users nowadays spend more time in online platforms. This creates a great opportunity for publishers to increase their revenue: as advertisers pay to publish their content in order to achieve higher visibility subscriptions or even downloads. There are many models from which an advertiser can pay to publish its content - it can be based either on visualization, click or conversion. The first and second concepts are intuitive to understand. The advertiser pays for the number of times an ad was seen or by the number of times it was clicked. The last concept is about what happens after a click, meaning the relevant information to consider a positive conversion, this is if an ad was clicked and also a posterior action followed this action, such as: a subscription, download or any other action which benefits the advertiser. If the action is followed through then this symbolizes a positive conversion. This prediction is done by using machine learning models feeding them with data, corresponding to both the publisher website and the user browsing it. Although, there are many implementations to these prediction models. First it is important to extract appropriate features from the raw data. This process called feature engineering enhances our prediction accuracy. In machine learning there are many types of models: the ones having a shallow architecture requires that this process is done manually by experts - experienced data scientists with domain knowledge of the problem at hand. Their work is to find a proper representation and all possible combinations of the provided data in order to create good features. It consumes a lot of time and for bigger data sets, its almost impossible to extract all possible combinations. However, with the introduction to deep learning neural networks: it became possible to automate this task. By exploring these models representative power we are able to generate and test these new features on flight (during training). In this work, we want to conclude the best techniques and approaches by applying different

models to multiple data sets - concluding which algorithms have the best results and why.

1.1 Digital advertising models

In the digital advertising world, publishers and advertisers have many pricing models to negotiate from. These models (Cost per Click - CPC, Cost per Mille - CPM , Cost per action - CPA, etc.) describe what an advertiser is actually paying for, so that the right model can be chosen according to their needs. Click prediction plays a very important role in this type of business because it gives us an approximate expectation about which ads can lead to a higher income. In this work we focus on two paying models, CPC and CPA. In the first model, advertisers pay for each time an ad was clicked, and in the second model for the number of positive conversions. [1]

Mathematically speaking, these revenue models can be described as:

$$\text{CPC} = (\text{cost to the advertiser}) \div (\text{number of clicks received}) \tag{1.1}$$

$$\text{CPA} = (\text{cost to the advertiser}) \div (\text{number of positive conversions}) \tag{1.2}$$

These models use CTR and conversion Rate (CVR) which can be described mathematically as:

- CTR: A ratio that indicates how often a user who sees an ad ends up clicking it. In other words, it is the number of clicks that an ad receives divided by the number of times the ad was shown.

$$\text{CTR} = \frac{\text{clicks}}{\text{impressions}} \times 100 \tag{1.3}$$

  e.g. an ad clicked 15 times with 100 impressions would get a CTR of 15%.

- CVR: The number of positive conversions per ad click divided by the total number of clicks, represented as a percentage. This means how often an ad that was clicked resulted in a positive conversion: user sign-up, item bought, app downloaded, etc.

$$\text{CVR} = \frac{\text{positive conversions}}{\text{clicks}} \times 100 \tag{1.4}$$

  e.g. an item ad that gets 50 sales and was clicked 1000 times would get a CVR of 50 / 1000 = 5%.

This type of work, prediction of CTR and CVR, is compensated in many data competitions like the ones present in kaggle [2], companies provide real data so that engineers and data scientists can develop and test their models, competing for the best log loss and come up with the best answers to questions such as which ads will increase our revenue?, which ads need to change? so that they can adapt their advertising strategies to come up with the best results for their customers at a good competitive price.

[1] In advertising context, a positive conversion is a situation that profits the company which the ad was made for, such as: subscription, download or watching a video linked to the ad.

[2] `https://www.kaggle.com`

1.2 Deep learning concepts

In this section we will introduce some basic concepts. Deep learning is a specific sub field of machine learning: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations[1]. When referring to the deep part in this field, we mean that instead of only one layer of representation (linear models), these models have successive layers stacked one after the other learning more complex projections as it deepens, each layer creates a representation based on the previous layer. Machine learning problems can be seen as a function $f(x)$ which transforms x into a certain y. The difference between machine learning (ML) - shallow and deep learning (DL) architectures is the way we compute $f(x)$, in ML can be defined as $f(x) = y$, however in DL we define as $f(x) = fL(...(f2(f1(x1))))$, where L is the number of layers. Lets take an image classification example:

1. Entry x - an image 28x28 transformed into a 748x1 vector.
2. 10 Neurons representing the probability of y [0 9]
3. Weights/parameters w and b - the weight and bias help the network tune the learning process
4. Classification [3] - for example the softmax normalization function so that our final result can be condense to a value [0 1]

ML shallow architectures, each neuron j will process an image sample, for simplicity we will consider 4 features, in this case neuron j would produce: $w_t^j x + bj = [(w_j, 1x_1) + (w_j, 2x_2) + (w_j, 3x_3) + (w_j, 4x_4)] + b_j$, as we only have this layer it's outcome is the score for this prediction. The previous result for each neuron is fed to our classification function, which gives us the probability result of neuron j belongs to class $c : P(y = c|x; w_c; b_c) = softmaxj(x^t w_j + b_j)$ [4]. However, in practice we do not feed one sample at a time, we have to feed a set of samples which is called - batch. This means that instead of just one x we will have a matrix X hence another one for W and a vector for b, with these we now train the network to adjust the parameter W and b, during a process called training. In this stage we apply a loss function that evaluates the results quality, measuring the distance between prediction and the real value. When transforming this into a DL network, we add intermediate layers between the input and the output[5](lets consider only 2 hidden layers for simplicity), transforming our previous function into: $y = f(x) = f_3(f_2(f_1(x1; W1; b_1); W_2; b_2); W_3; b_3)$, where each $x_n = f_n 1(x_n 1)$. Unlike the example where the classification function was only softmax when we have intermediate/hidden layers we have to apply more suitable functions to their output as they will serve as input to the next level.

1.3 Objectives

The main focus of this work is to explore how can we improve model performance in CTR or/and CTR predictions. One way to do this is through feature engineering - the process of using your own knowl-

[3] This type of functions are denominated as activation functions - appendix A
[4] softmax formula in the appendice
[5] these layers are called hidden layers

edge about the data and about the machine learning algorithm at hand to make the algorithm work better by applying hard-coded (non-learned) transformations to the data before it goes into the model [1]. This excerpt gives us some insight about this step, which is finding good representations for the data (especially categorical as it is the most frequent in CTR/CVR problems) and also generate new features through combination of the existing ones. This process is very time consuming and the effort grows exponentially for bigger data sets. To tackle this problem we will study deep learning models that automate this process. In summary, ou goal is to use deep learning models to automatically generate this feature combinations and enhance our prediction for CTR/CVR.

1.4 Outline

This thesis is organized as follows: Chapter 2 will analyse the structure of existing solutions which implement automated feature engineering; Section 3 will explain the work done and the steps taken in order to achieve our conclusions; Section 4 explains in detail how these models are implemented and some clarification about the python libraries used in this process; Section 5 contains the results achieved by these models and a small debate on their performance individually; Section 6 concludes this thesis and discusses some approaches that could enhance this solutions

Chapter 2

Related Work

The main purpose of this chapter is to describe the existing solutions supporting automated feature engineering, and good manual feature engineering practices to inspire us to a solution in applied to click prediction, either for CTR or CVR. Many interesting works were analysed but only a few were deemed as being most relevant for this purpose. The next sections will briefly present each of the selected works, while trying to answer three important questions, namely: what is the problem being addressed, what approach did the authors propose to address that problem, and what results they were able to achieve. Finally, there are some other related works which represent other good ideas that could be applied.

2.1 Convolutional Network

The Convolutional Click Prediction Model [2] (CCPM) can be used for click prediction on single ad impression or sequential ad impression, which is a series of single impressions. This last type, when collected for each individual user, describes the specific ad-clicking behavior of each user. The main goal is to help online advertisers increase their revenue in commercial Web publishing business. The proposed CCPM model is based on a convolutional neural network, and has the ability to extract local and global key features from the input instances, through wide convolutional layers alternated with dynamic pooling layers. This layer combination can obtain significant feature combinations at each step i denoted as the i^{th} order feature map. We can see in Fig 2.1 the model structure.

The first layer of the model is an embedding layer, which feeds the first convolutional layer, giving as output an instance matrix that has the following structure: each entry is an embedding value for a respective element in an instance, elements have varied length, hence the resulting matrix has also varied length; the columns in the instance matrix represent a distinct attribute of an element; and the values in the embeddings are estimated during training in order to generate a more suitable representation.

The convolutional layer applies a filter weight to the previous layer in a one-dimensional row-wise mode so that every row in the resulting matrix is a one-dimensional convolution. These weights are optimized at each round and give the model the advantage of detecting features and recognizing specific ranges of neighborhood in the input instances.

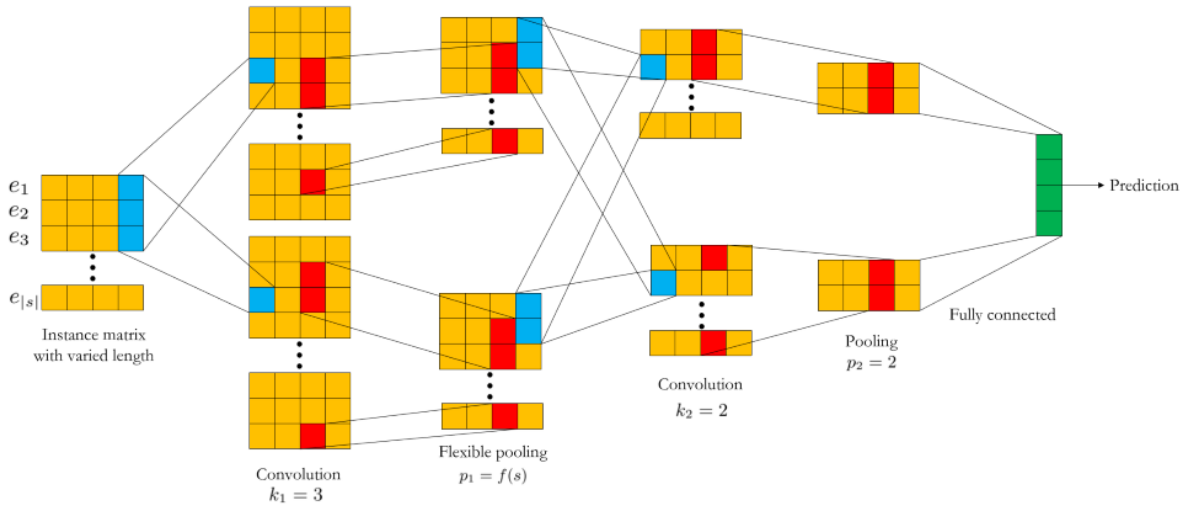

Figure 2.1: CCPM structure from [2]

The dynamic pooling layer has some interesting things to notice too: it has a shrinking variable p that controls the number of selected biggest values from the previous layer output. This represents the filtering of the result into the p biggest values in the previous layer output. Each stage is a combination of a convolutional and a dynamic pooling layer. Before the next stage, an activation function is applied to obtain threshold values from the last layer output. At the end of this process there is a fixed size pooling layer followed by a fully connected layer and the prediction is done through softmax.

The model was tested for both types of inputs: single ad impression and sequential ad impression. The authors used two public real-world datasets chose[1] and Avazu[2]. The second dataset has 17 data fields per click data element. The model was compared to three state-of-the-art methods – logistic regression (LR), factorization machine (FM), and a recurrent neural network (RNN). The last one was not used on the Avazu dataset, as it does not contain sequential data. Using the log-loss as measure, CCPM outperformed the other models on both datasets, concluding that this model is effective. The effectiveness of this model relies on the good tuning of both the dimension and the filter width hyper-parameters, responsible for recognizing specific neighborhoods. As the size of the filter width and layer go deeper, key features are extracted and noise is eliminated.

2.2 Deep and Cross Network

In the context of ad click prediction, ref. [3] identifies frequently predictive features and at the same time explores unseen or rare cross features. The Deep and Cross network (DCN) has a very interesting structure because it is the combination of two separate approaches: a Cross Network, where feature crossing is done explicitly at each layer, and a Deep Network, which is a fully-connected feed-forward neural network. Both approaches work in parallel and the outputs are concatenated so that the networks are trained jointly. More detail is shown in Fig. 2.2.

[1] `http://2015.recsyschallenge.com/`
[2] `https://www.kaggle.com/c/avazu-ctr-prediction`

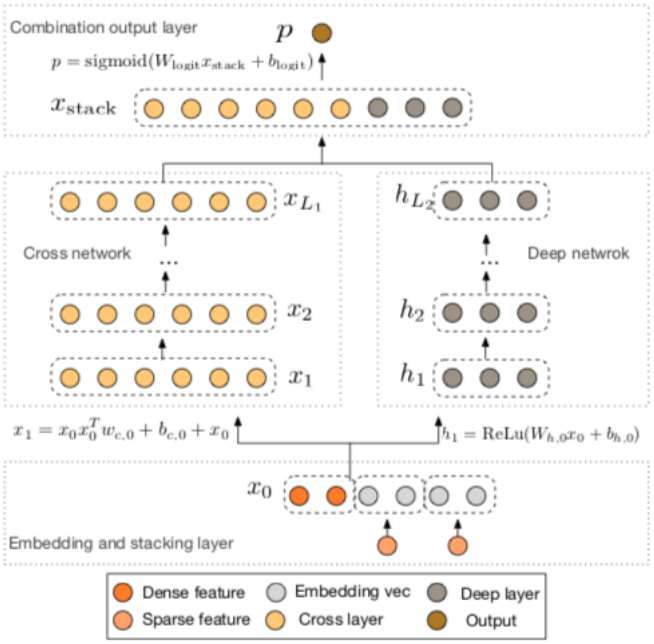

Figure 2.2: The Deep & Cross Network from [3]

Model	DCN	DC	DNN	FM	LR
Logloss	**0.4419**	0.4425	0.4428	0.4464	0.4474

Table 2.1: Best logloss from different models from [3]

The structure starts with an embedding and stacking layer which feeds both networks with an input vector. This input vector is computed by stacking the normalized dense features and the embedding vectors from the categorical features, which are usually encoded as one-hot vectors, generating a high-dimensional feature space for large vocabularies. These binary vectors are embedded into dense vectors of real values in order to reduce dimensionality.

This work was tested on the Criteo Display ADS data,[3] which contains 13 integer features and 28 categorical features. The model obtained a lower log-loss compared to Deep Crossing (DC) [4], a deep neural network (DNN), a factorization machine (FM), and logistic regression (LR). The results are shown in Table 2.1. The authors also made a more detailed comparison with DNN, that is basically a DCN without the cross network. In this case they varied the number of parameters for both models, and in every case DCN performed better.

[3] `https://www.kaggle.com/c/criteo-display-ad-challenge`

#Params	5×10^4	1×10^5	4×10^5	1.1×10^6	2.5×10^6
DNN	0.4480	0.4471	0.4439	0.4433	0.4431
DCN	**0.4465**	**0.4453**	**0.4432**	**0.4426**	**0.4423**

Table 2.2: Best log-loss achieved with various memory budgets from [3]

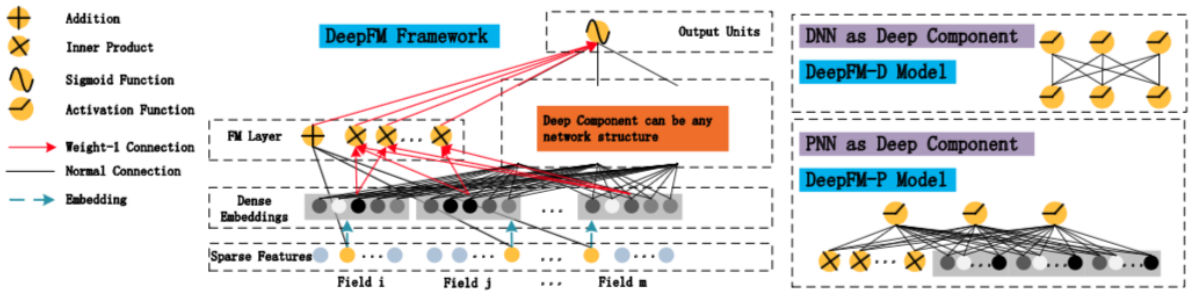

Figure 2.3: The structure of the DeepFM framework in the left, DNN component structure in the up rightmost side, PNN structure at the down rightmost side, from [5]

2.3 Factorization based

2.3.1 Deep Factorization Machine

DeepFM [5] is an end-to-end deep learning framework that enables to learn both low- and high-order feature interactions. This paper explores the click prediction paradigm for recommender systems to estimate the probability of a user clicking on a recommended item, like online advertising where advertisement owners want to improve their revenue based on click through rate (CTR). The solution architecture is a factorization machine (FM) component introduced by [6] to learn feature interactions up to order 2, and a deep component which is a feed-forward neural network whose objective is to learn high-order feature interaction.

This model has 2 variants, DeepFM-D and DeepFM-P, which differ on the fact that the first uses a fully-connected deep neural network, and the second uses a Product-based Neural Network (PNN) [7], as shown in Fig. 2.3. The last one introduces a product layer between the embedding and the first hidden layer, and this approach has 3 variants the IPNN that uses the inner product, the OPNN that uses outer product and the PNN that uses both.

This work was tested on two data sets, Criteo[4] and Company, a private dataset. The first has 45 million user click records with 13 numerical features and 26 categorical features. These have been split into train and test sets in two different ways: randomly and sequentially. The resulting data sets were designated as Criteo-Random and Criteo-Sequential, respectively. The Company dataset is a commercial industrial set, where the author collected data for 8 consecutive days from game center of an App Store, acquiring over 1 Billion records with the following features: app (e.g., id, category, etc.), user (e.g., user's downloaded apps, etc.), and context (e.g., operation time, etc.). To evaluate the model, two metrics were used: AUC (area under the ROC curve) and log-loss. The results are presented in Table 2.3.

To analyze the results from the previous table, we can see 12 models, which the proposed approaches outperformed. These models were divided into 4 categories:

- Wide models: Logistic Regression (LR) and Factorization Machines (FM) [6, 8].

[4] https://www.kaggle.com/c/criteo-display-ad-challenge/data

	AUC			LogLoss		
	Company∗	Criteo-Random	Criteo-Sequential	Company∗	Criteo-Random	Criteo-Sequential
LR	0.8641	0.7804	0.7777	0.02648	0.46782	0.4794
FM	0.8679	0.7894	0.7843	0.02632	0.46059	0.4739
DNN	0.8650	0.7860	0.7953	0.02643	0.4697	0.4580
FNN	0.8684	0.7959	0.8038	0.02628	0.46350	0.4507
IPNN	0.8662	0.7971	0.7995	0.02639	0.45347	0.4543
OPNN	0.8657	0.7981	0.8002	0.02640	0.45293	0.4536
PNN∗	0.8663	0.7983	0.8005	0.02638	0.453305	0.4533
LR & DNN	0.8671	0.7858	0.7973	0.02635	0.46596	0.4565
FM & DNN	0.8658	0.7980	0.7985	0.02639	0.45343	0.4551
DeepFM-D	0.8715	0.8016	**0.8048**	0.02619	0.44985	**0.4497**
DeepFM-IP	**0.8720**	**0.8019**	0.8019	**0.02616**	**0.4496**	0.4525
DeepFM-OP	0.8713	0.8008	0.8020	0.02619	0.4510	0.4524
DeepFM-∗P	0.8716	0.7995	0.8015	0.02619	0.4515	0.4530

Table 2.3: Performance evaluation using AUC and log-loss from [5]

		Wide				Deep			
		DeepFM-D	DeepFM-IP	DeepFM-OP	DeepFM-∗P	DeepFM-D	DeepFM-IP	DeepFM-OP	DeepFM-∗P
Company∗	AUC	0.4%	0.47%	0.39%	0.43%	0.75%	0.67%	0.65%	0.61%
	LogLoss	0.49%	0.61%	0.49%	0.49%	0.91%	0.87%	0.80%	0.72%
Criteo-Random	AUC	1.54%	1.58%	1.44%	1.28%	1.98%	0.60%	0.34%	0.15%
	LogLoss	2.33%	2.39%	2.08%	1.97%	4.22%	0.85%	0.43%	0.40%
Criteo-Sequential	AUC	2.61%	2.24%	2.26%	2.19%	1.19%	0.30%	0.22%	0.12%
	LogLoss	5.1%	4.52%	4.54%	4.41%	1.81%	0.40%	0.26%	0.07%

Table 2.4: DeepFM models improvement over Wide and Deep Components, from [5]

- Deep models: Deep Neural Network (DNN), Fuzzy Neural Network (FNN), Product-based Neural Network (PNN) [7] with 3 variants IPNN, OPNN, PNN*.
- Wide & Deep models: LR & DNN and FM & DNN.
- DeepFM models: DeepFM-IP, DeepFM-OP and DeepFM-*P.

Table 2.4 presents the improvement of the DeepFM models over their Wide and Deep components respectively, concluding that DeepFM-D beats the other approaches by a significant percentage.

2.3.2 Deep Learning over Multi-Categorical Data

The problem at hand is user response prediction, which plays a critical role in many Web applications for recommender systems. The solution for this type of problem must ensure that we can extract a prediction allowing us to conclude if a potential ad can be considered “relevant” to a user. By calculating the click-through rate as aforementioned, this paper [9] takes CTR estimation challenge over a large multi-field categorical feature spaces, and introduces two types of deep learning models using supervised- and unsupervised-learning embedding methods. The first is called Factorization Machine supported Neural Network (FNN) inspired by FMs [6, 8, 10]. The second is a Sampling-based Neural Network (SNN) powered by either a sampling-based restricted Boltzmann machine (SNN-RBM) [11] or a sampling-based denoising auto-encoder (SNN-DAE) [12] using a proposed negative sampling method. These layers have the property of efficiently reducing feature spaces from sparse to dense continuous, hence being used as the bottom layer to feed multi-layer neural nets with full connections which can later explore non-trivial data patterns.

In Fig. 2.4 and Fig. 2.5 we explore the solution architecture for each model (FNN and SNN, respectively) in more detail.

We start by describing the network structure from the bottom where we find the FM layer. It should be noted that it receives as input sparse binary features, because categorical fields are one-hot encoded field-wise, meaning that, an example where we have a field city, each instance is represented as a binary vector with only one positive unit, that represents the city that instance belongs to. As we can expect, the length of this vector is the same as our field size and for higher dimensions the input becomes more sparse and difficult to handle. The feature interaction is computed as the inner product of their vectors across fields, resulting on a dense vector initialized when training the FM using a sigmoid function, solving the sparsity problem. This allows the model to learn a good structural data representation in the latent space allowing the learning from the fully connected layers to be more efficient. Each feature is assigned with a bias weight, the initial weights from the FM layer are trained using stochastic gradient descent (SGD) and the hidden layers from the FNN are updated using restricted Boltzmann machine (RBM) pre-training which preserves the information detail, as only the weights for positive input units are calculated, reducing the computational complexity. In order to tune the weights, the authors use chaining rule back propagation.

The SNN model differs from the FNN model in the bottom layer. The data is pre-trained using a negative sampling method, which instead of considering the whole feature set for each feature field, selects m negative units and for example city=Lisbon when $m = 1$ and random when $m = 0$. The unsampled parameters represented as black units in Fig. 2.5 are ignored when pre-training the data instance. Using this sampled units we can train using two variants, a sampling-based denoising auto-encoder denoted as SNN-DAE and a sampling-based restricted Boltzmann machine denoted as SNN-RBM. The first uses stochastic gradient descent (SGD) with an unsupervised approach allowing a big decrease on the data dimension and achieving a high recovery performance. The second uses contrastive divergence. These methods are used to initialize the weights in the first layer, which is fully connected with a sigmoid activation function resulting on a real-value dense vector to be fed to the further layers. This approach dramatically reduces computational complexity.

The proposed models where evaluated on iPinYou dataset,[5] which is a public real-world display ad dataset, containing very few positive labels compared to the data instance size. The features are all categorical, hence were one-hot encoded beforehand. To compare the performance the authors used the area under the roc curve (AUC) measure against a Logistic Regression (LR) and a Factorisation Machine (FM), which are a linear and a non-linear model respectively, widely used for CTR prediction [13]. The results, as we can observe in Table 2.5, shows that the proposed models outperform the compared ones, concluding that the fact that they are able to catch underlying patterns does enhance a model performance.

[5] `http://data.computational-advertising.org/`

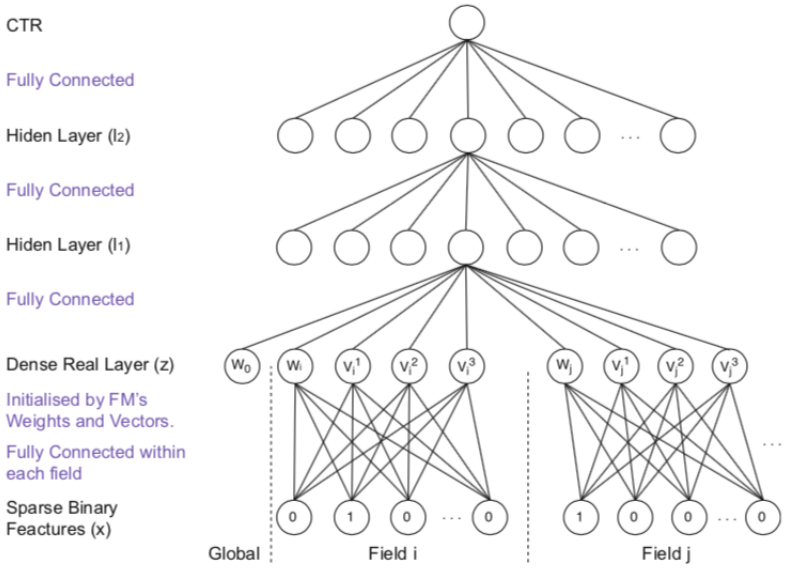

Figure 2.4: The FNN model structure, from [9]

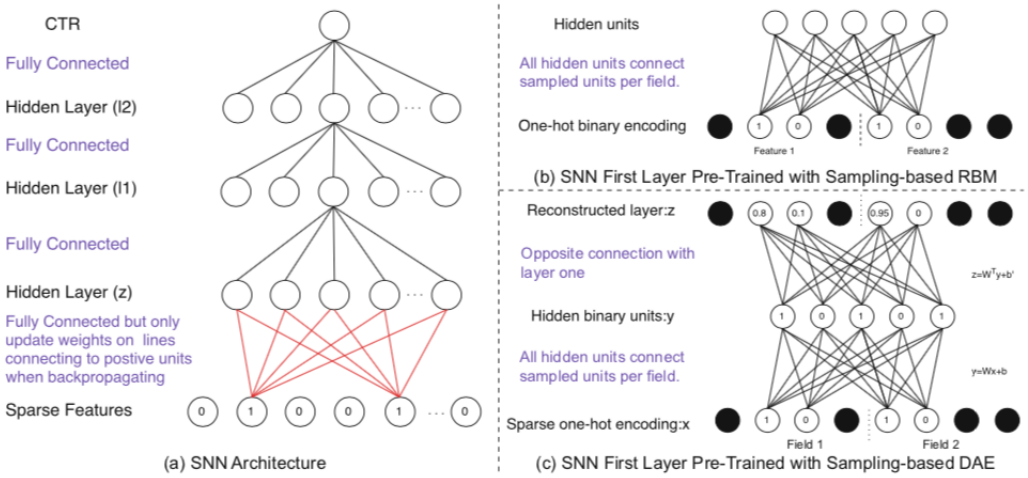

Figure 2.5: The FNN model structure, from [9]

	LR	FM	FNN	SNN-DAE	SNN-RBM
1458	70.42 %	70.21 %	**70.52 %**	70.46 %	70.49 %
2259	69.66 %	69.73 %	**69.74 %**	68.08 %	68.34 %
2261	62.03 %	60.97 %	62.99 %	**63.72 %**	**63.72 %**
2997	60.77 %	60.87 %	61.41 %	**61.58 %**	61.45 %
3386	80.30 %	79.05 %	**80.56 %**	79.62 %	80.07 %
all	68.81 %	68.18 %	**70.70 %**	69.15 %	69.15 %

Table 2.5: Logloss results for the proposed solutions compared to FM and LR, from [9]

2.3.3 Sparse Factorization Machines

Sparse Factorization Machines [14] are a variant of Factorization Machines [6, 8] using the Laplace [6] instead of the traditional Gaussian [7] distribution. As already mentioned in other approaches, FM models are quite useful in the task of revealing proper combinations of data, but the data sparsity becomes a problem, hence the change in distribution proposed by these authors. Laplace can better fit the sparse data with higher ratio of zero elements.

2.4 Atention Based Networks

2.4.1 Deep Interest Network

The Deep Interest Network (DIN) [15] proposes a solution exploring historical behavior data where user interests are represented using an interest distribution and has an attention-like network structure. This type of data is described in two aspects:

- Diversity: User interest might be connected to more than one item. This can be analyzed through user characteristics such as age, gender, etc. (e.g. a teenager might be interested in t-shirts, games and tech at the same time);
- Local Activation: As user interests are diverse, not all historical behavior is relevant to the analysis (e.g. a user might click on some item due to a previously complementary item purchased and not on some other items in previous records).

Before we step into the model structure, let us analyze the way data is represented for this solution. In Fig. 2.6, we can notice that this model is provided with one-hot vectors and multi-hot vectors. The major difference between these two vector structures is that one-hot vectors only have one positive instance, while multi-hot vectors can have more than one positive instance (where a positive instance is represented as 1 and negative instances are represented as 0). The second thing we should notice is that the dimensionality for these features varies from 2 to 10^9, which allows us to conclude that the data is very sparse. We can also conclude that the number of positive instances is very low, except for the multi-hot vectors, meaning that this structure refers to more than one label. Last but not least, we may observe that features are separated into four groups.

The DIN model uses embedding vectors for user, behavior, ad, and a weight denoted as attention score. The weight defines the contribution of a behavior to the overall user interest on a certain ad, being the user embedding vector a sum of the respective attention-behavior operation for each ad. This model is based on Multilayer Perceptrons (MLPs), which are able to reduce a lot of the feature engineering compared to logistic regression model (LR). MLPs in this context are used to fit the aforementioned embedding vector followed by a pooling layer to get a fixed size embedding vector through sum or average operations. By pooling, this process drops information.

6
7

Feature Category	Feature Name	Dimemsion	Type	#Nonzero Ids/Sample
User Profile Features	gender	2	one-hot	1
	age_level	~ 10	one-hot	1
	...	...	...	...
User Behavior Features	visited good_ids	$\sim 10^9$	multi-hot	$\sim 10^3$
	visited shop_ids	$\sim 10^7$	multi-hot	$\sim 10^3$
	visited cate_ids	$\sim 10^4$	multi-hot	$\sim 10^2$
	...	...	...	...
Ad Features	good_id	$\sim 10^7$	one-hot	1
	shop_id	$\sim 10^5$	one-hot	1
	cate_id	$\sim 10^4$	one-hot	1
	...	...	...	...
Scene Features	pid	~ 10	one-hot	1
	time	~ 10	one-hot	1
	...	...	...	...

Figure 2.6: Feature representation and statistics, from [15]

2.4.2 AutoInt

The AutoInt model proposed in [16], uses a multi-head attention mechanism [17] combined with residual connections. This work aims to find low- and high-dimensional representations of the sparse raw features and their meaningful combinations. This study focus on Click through rate. The problem setup starts with a set of fields $x \in R^n$[8] being the concatenation of a user u and some item v features.

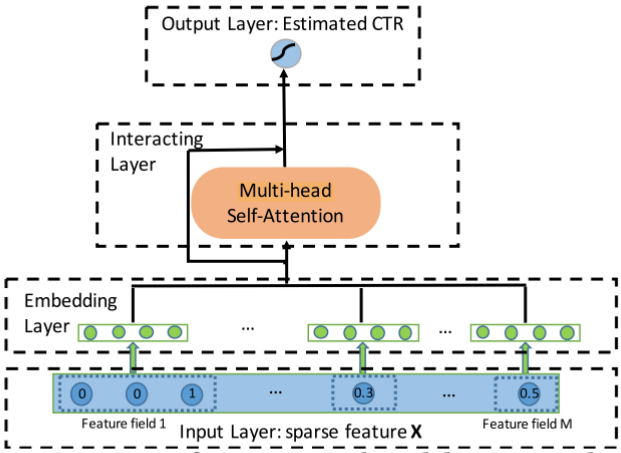

Figure 1: Overview of our proposed model AutoInt. The details of embedding layer and interacting layer are illustrated in Figure 2 and Figure 3 respectively.

Figure 2.7: AutoInt model structure

These can be, as aforementioned both numerical or the most common case categorical which are encoded as one-hot vectors[9], the size of our set is n. Following the collection of such fields, the goal becomes finding p-order combinatorial features. Given a function g our goal is to generate $< x_i1, x_i2, ..., x_ip >$, where p is the number of features involved in this computation, which can be: multiplication; outer product; inner product; etc. But to achieve this our data must be represented in a low-dimensional space all features into the same space and generate of higher order combinations.

AutoInt learns this representations hence generating high order combinatorial features, as we in-

[8] missing real symbol
[9] acrescentar ao appendice – this type of approach is very memory consuming

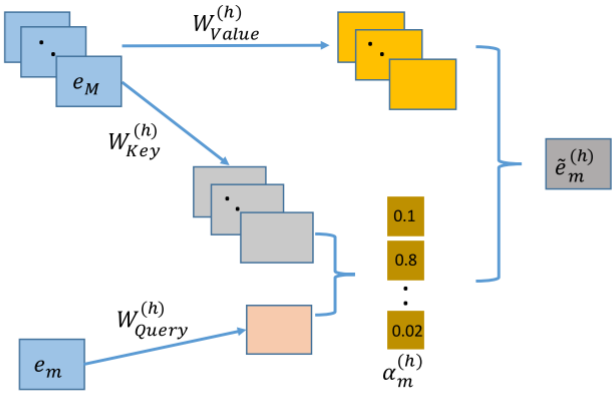

Figure 3: The architecture of interacting layer. Combinatorial features are conditioned on attention weights, i.e., $\alpha^{(h)}_m$.

Figure 2.8: Interaction layer structure

crease the number of heads we are able achieve higher orders. Composed by four layers where the first is input layer, having the concatenation of our features, followed by an embedding layer transforming them into a low dimensional vector using an embedding matrix for each. The last two layers are the interaction layer, where the function g is applied, and the output layer, which concatenates the last layer output, applies non linear transformations and calculates the logloss. The interaction layer, is composed by two core technologies, attention mechanismsc̃ite and residual connections[].

We can see in fig. 2.8 the combination of feature e_m and e_M, being the latest a set where our pivot feature will interact. Also important to note the $W^(h)$ value which is a network weight passed onto the next head appended to the result of this combination. The first observation refers to the attention mechanism as we saw we focus on one feature while computing every combination to the rest of features in our set. The second observation is the residual value that allows the network to 'remember some considerations' made about these combinations. Deeper into this layer we will be pivoting features which already are combinations and combine them with the rest of them increasing from 2nd to 3-order and so on, but also, recalling which ones where better and associating a relevance to it. This combination allows the model to learn the interactions and uncover hidden patterns within data as it is difficult to come up with all this computation for a big feature dictionary. But it also has its downsides, as we add heads to our model it becomes computationally more expensive.

Chapter 3

Proposed Approach

In this chapter we will describe our approach to make this study. As aforementioned our goal is to compare different models that execute feature engineering automatically. Our approach began by creating a processing pipeline 2.1 First we have to select a data set, it can be either for click-through rate or conversion rate. We have selected two data sets by coincidence both CTR. The second step is to select a model to test, in the 4 we show both the theory behind the selected models and their possible implementation. Last step is to chose a proper evaluation metric, as we can see 2.1 there are many functions that can be selected which will be explain further in this chapter.

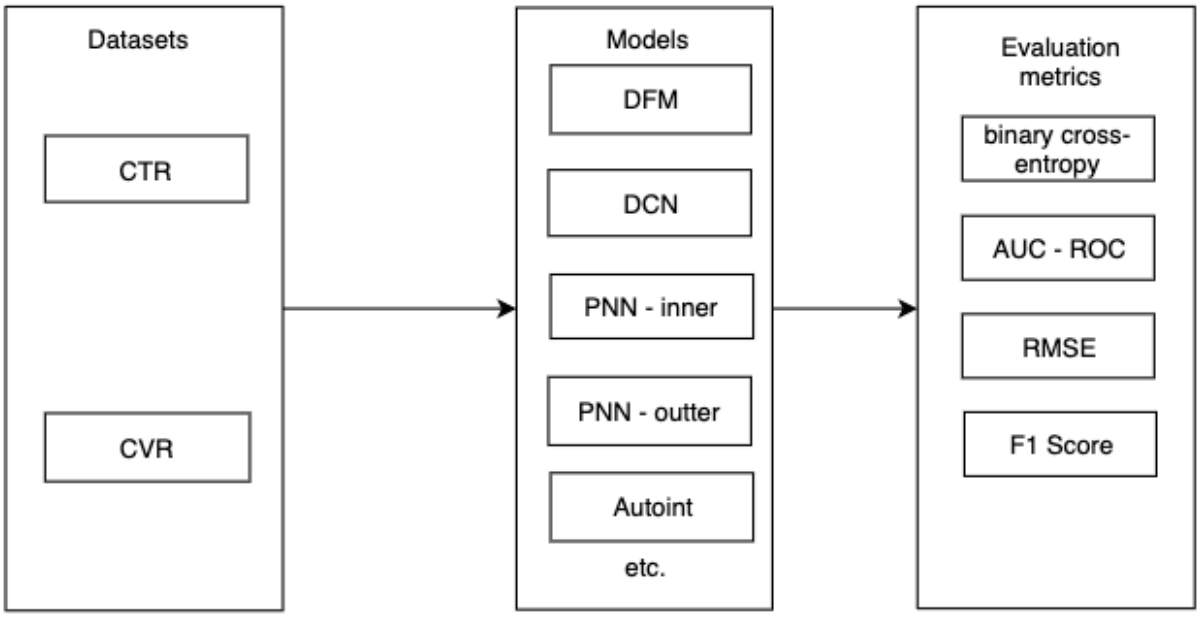

Figure 3.1: Evaluation pipeline

3.1 Data sets

In this section we shall address what to expect in CTR/CVR data sets. Data comes with different types, however in these specific cases the most common are numerical and categorical, we can also find time

series. The numerical features, are the ones having a meaningful ranking among them [18]. Taking as example grades, we know that if student A had a better grade than student B, then $A > B$. Hence, these values domain is ordered, and this order has a meaning. Categorical data on the other hand, does not have an order. Lets consider another example, colors [Red, Blue, Green], a naive approach to encode this data would be assigning a number to each value, e.g red - 1, blue - 2 and green - 3. However, we cannot state that red blue green, so this approach would induce our model to error. This type of data is called qualitative as it does not have a numerical meaning. One way to deal with this data them for our model to interpret is to transform into a on-hot vector, where only one positive instance would exist at each record, for blue for example we would have blue - [0, 1, 0]. Despite being commonly used we have to keep in mind that for bigger vocabularies this increases sparsity, becoming expensive to both memory and processing.

There are many other approaches to this problem such as label encoder, rank encoding and others. Beyond this, we also have the other issues to address before the data is ready to be fed into our model. Many data sets can have: noisy data; missing values (NaNs); biased data; big vocabulary ranges; string encoded or even list encoded features. There are many approaches to solve these problems, but the biggest concern when tackling it is, whether the solution we come across does not have a negative impact to our feature distribution and its co-relation with other features. Imagine a scenario where we have many missing values, we cannot just delete the rows where their occur as we may lose important information, and if we delete the column itself we will get to the same problem. Instead we should study the distribution in order to see which value can be assign to the missing values. Python pandas1 is an open source library where many functions such as pandas.DataFrame.info() and pandas.DataFrame.describe(). These functions give us insight about the existence of: null values (NaNs), most common percentiles for numeric data, vocabulary size and most frequent values for categorical data; in chapter 1 we will see an example of these functions outputs.

3.2 Models

Deep learning (DL) models created an opportunity to automate feature engineering. Our goal is to generate combinatorial features, in order to enhance our model accuracy. These can be one of two types, low-order or high-order; The first ones are linear combinations between two features, the latter ones combine three or more. Given an input vector $x \in R_n$, where n is the number of existing features. The definition of a p-order combination is $\langle x_{i1}, x_{i2}, ..., x_{ip} \rangle = g(x_{i1}, ..., x_{ip})$, where p is the number of features interacting and $g(\cdot)$ is a combinatorial function [16]. In general DL models apply these type of functions through a layer enabling this relation to be learned, for example the cross layer in [3].

Our research lead to believe that some mathematical operations, such as: inner product, outer product, multiplication, factorization, etc; have properties that explore the relation between features. These are applied in many model layers, see chapter 2. We will test and study some of these models, in order to conclude which ones perform better and how these layers influence the prediction. To do this we selected the following models:

- Autoint
- deep & cross network
- deep factorization machine
- product-based neural network - inner and outer product version

These models apply the aforementioned operations combined with deep layers, in order to explore both low- and high-order interactions. In the next chapter we will make the bridge between a possible implementation and the math behind these models.

3.3 Evaluation

	Actual Positive	Actual Negative
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

Figure 3.2: Confusion matrix

In this section we will define and explain some metrics to evaluate our model performance. These are also called loss functions, as they calculate how far the model is from true predictions, both positive and negative. First concept we have to understand what is a confusion matrix, fig. 3.2. In case of predictions if either we predict that an instance belongs to a class or not, from here we can extract the accuracy=(TP+TN)/(TP+FP+FN+TN),precision=TP/TP+FP and recall=TP/TP+FN. The green areas represent the values we got right while the red area the ones we did not. The metrics are:

- F1 score, is a metric commonly used for classification problems defined as:

  $2 \cdot \frac{precision \cdot recall}{precision + recall}$

- Root-squaredmeanerror(RMSE), it is popular within regression problems defined as:

  $\sqrt{\frac{\sum_{i=1}^{N}(predicted_i - actual_i)}{N}}$

- Area Under the receiver operating characteristics curve (AUC - ROC), first we need to understand the concept of sensitivity = TP / (TP + FN) and specificity = FN / (FP + TN). This metric is the plot between sensitive and (1-specificity).

- binary cross-entropy, is the negative average of the log of corrected predicted probability for each instance, meaning it takes a certainty of the classification into account. Defined as:

  $L(y_{real}, y_{predicted}) = -\frac{1}{N}\sum_{i=0}^{N} y_{real} * log(y_{pred}) + (1 - y_{real}) * log(1 - y_pred)$

3.4 Discussion

Dealing with these type of problems, CTR/CVR. One must have into account two things: These type of problems are classification problems, where $y \in \{0, 1\}$; The data found to solve these tasks comes with many problems to solve. A study of each data set has to be done in order to understand what techniques to clean the data and prepare it to be useful as input for a model. Models need hyper-parameter tuning, e.g: setting the number of hidden layers, how many cross layers Deep and cross network should use, how many attention head our model needs. However, we address this matter superficially, because our goal is to understand how feature engineering is done and how does affect these models performance. We also have to select a proper activation function for both the output layer and the layers in between. In sum, there are many things to address when deploying a deep learning model. This type of work consumes a lot of time and computational resources. In this work we only cover the basics, as it is not our main goal, but is necessary for the reader understanding.

Chapter 4

Implementation

In this chapter, we will briefly explain the logic behind our selected models and make the bridge between the math behind each model logic and their corresponding implementation using python data science libraries. These models can be implemented using keras custom layer. By defining the build method we can create the weights and bias needed, which are learnable parameters that allow our model to keep track of how much importance the result components of our layer should have. The logic to how the features can be combined with the corresponding weights is defined in the call method, where the math that make our models unique. Exploring different algorithms to combine the features and create low and high-order combinatorial features to improve the model performance. In each of the following sections, we will explain the key points to implement some of our models special behavior

4.1 Deep Factorization Machine

Deep factorization machine [19], combines two architectures the factorization and deep component. The first, measure feature interactions through the inner product of their latent vectors having the following output: $yFM = \langle w, x \rangle + \sum_{i=1}^{d} \sum_{j=i+1}^{d} \langle V_i, V_j x_i \cdot x_j$. Where the first part of the sum reflects order-1 feature importance and the second part the pairwise order-2 feature interaction. The second component is a feed-forward neural network that learns high-order features and as the following output: $a^{(l+1)} = \sigma(W^l a^l + b^l)$. Where l represents the $l - th$ layer and σ is the activation function, the output is a dense real value later fed to a sigmoid function.

The previous formulas can be implemented in keras custom layers as:

```
square_of_sum = tf.square(reduce_sum(concated_embeds_value,
                axis=1, keep_dims=True))
sum_of_square = reduce_sum(concated_embeds_value * concated_embeds_value,
                axis=1, keep_dims=True)
cross_term = square_of_sum - sum_of_square
cross_term = 0.5 * reduce_sum(cross_term, axis=2, keep_dims=False)
```

This code is to be implemented in the call of our custom layer as we can see, our cross term is the square of the sum minus the sum of the square, solving notable case gives us the multiplication between our features. The next section code corresponds to the deep part:

```
self.kernels = [self.add_weight(name='kernel'+str(i),
        shape=(hidden_units[i],
        hidden_units[i + 1]),
        initializer=glorot_normal(
        seed=self.seed),
        regularizer=l2(self.l2_reg),
        trainable=True) for i in range(len(self.hidden_units))]

self.bias = [self.add_weight(name='bias'+str(i),
        shape=(self.hidden_units[i],),
        initializer=Zeros(),
        trainable=True) for i in range(len(self.hidden_units))]

if self.use_bn:
        self.bn_layers = [tf.keras.layers.BatchNormalization()
                for _ in range(len(self.hidden_units))]
```

This should be implemented in the build function and the variables self.kernel and self.bias are the weights and bias, correspondingly.

```
self.kernels = [self.add_weight(name='kernel' + str(i),
        shape=(idden_units[i], hidden_units[i + 1]),
        initializer=glorot_normal(seed=self.seed),
        regularizer=l2(self.l2_reg),
        trainable=True) for i in range(len(self.hidden_units))]

self.bias = [self.add_weight(name='bias' + str(i),
        shape=(self.hidden_units[i],),
        initializer=Zeros(),
        trainable=True) for i in range(len(self.hidden_units))]

if self.use_bn:
        self.bn_layers = [tf.keras.layers.BatchNormalization()
                for _ in range(len(self.hidden_units))]

        self.dropout_layers = [tf.keras.layers.Dropout(self.dropout_rate,
                seed=self.seed+i)
```

```
            for i in range(len(self.hidden_units))]

        self.activation_layers = [activation_layer(self.activation)
            for _ in range(len(self.hidden_units))]

super(DNN, self).build(input_shape)
```

Lastly the call method should be defined as above, where the main operation is the multiplication of the weight and the corresponding input.

4.2 Deep & Cross Network

Deep and cross network [3], as also uses a deep part which implementation would be similar to the one in the previous section. So we will only focus on the cross layer, that as the following formula: $x_{l+1} = x_0 x_l^T w_l + b_l + x_l$. Where l denotes the layer, w_l and b_l the weight and bias, correspondingly. This operation between vector creates polynomial with a degree equal to the layer depth, allowing the model to extract high-order feature interactions from this. A possible implementation for this layer is as follows:

```
self.kernels = [self.add_weight(name='kernel' + str(i),
 shape=(dim, %1),
 initializer=glorot_normal(seed=self.seed),
 regularizer=l2(self.l2_reg), trainable=True)
  for i in range(self.layer_num)]

 self.bias = [self.add_weight(name='bias' + str(i),
  shape=(dim, 1),
  initializer=Zeros(),
  trainable=True) for i in range(self.layer_num)]
```

Where we define the needed weight and bias respectively, should be implemented in the build function.

```
x_0 = tf.expand_dims(inputs, axis=2)
x_l = x_0
for i in range(self.layer_num):
        xl_w = tf.tensordot(x_l,
        self.kernels[i],
        axes=(1, 0))

        dot_ = tf.matmul(x_0, xl_w)
        x_l = dot_ + self.bias[i] + x_l
        x_l = tf.squeeze(x_l, axis=2)
```

This is the logic of our layer, should be implemented in the call function. As we can see we successively multiply the original input and as the layer grows deep, we attain a $l-th$ degree polynomial as previously mentioned.

4.3 AutoInt

This last model uses an interesting approach which can be enhanced by changing the operation related to feature interaction. In this work it is used the inner product but can be defined as a neural network. The correlation between features is define as: $\alpha_{m,k}^{(h)} = \frac{\exp^{\psi^{(h)}}(e_m,e_k)}{\sum_{i=01}^{n}\exp^{\psi^{(h)}}(e_m,e_k)}$, where ψ is our similarity function between feature m and k under the attention head h define as $\psi^h(e_m,e_k) = \langle W_{Query}^h e_m, W_{Key}^h e_k\rangle$, W_{Query} and W_{Key} are transforming matrices from the original embedding space to a new space. The representation of feature m is updated in this subspace guided by the coefficient α as follows $\hat{e}_{m,k}^{(h)} = \sum_{k=1}^{M}\alpha_{m,k}^{(h)}(W_{value}^{(h)}e_k)$, and in the last step a residual connection is added to preserve both combinatorial and raw features where $e_m^{Res} = ReLU(\tilde{e}_m + W_{Res}e_m)$. This can be implemented as follows:

```
self.W_Query = self.add_weight(name='query',
shape=[embedding_size, self.att_embedding_size * self.head_num],
dtype=tf.float32,
initializer=tf.keras.initializers.TruncatedNormal(seed=self.seed))

self.W_key = self.add_weight(name='key',
shape=[embedding_size, self.att_embedding_size * self.head_num],
dtype=tf.float32,
initializer=tf.keras.initializers.TruncatedNormal(seed=self.seed+1))

self.W_Value = self.add_weight(name='value',
shape=[embedding_size, self.att_embedding_size * self.head_num],
dtype=tf.float32,
initializer=tf.keras.initializers.TruncatedNormal(seed=self.seed+2))

self.W_Res = self.add_weight(name='res',
shape=[embedding_size, self.att_embedding_size * self.head_num],
dtype=tf.float32,
initializer=tf.keras.initializers.TruncatedNormal(seed=self.seed))
```

These are the weights needed for this model, and the logic can be defined as:

```
querys = tf.tensordot(inputs,
        self.W_Query,
```

```
            axes=(-1, 0))

        keys = tf.tensordot(inputs, self.W_key, axes=(-1, 0))
        values = tf.tensordot(inputs, self.W_Value, axes=(-1, 0))

        querys = tf.stack(tf.split(querys, self.head_num, axis=2))
        keys = tf.stack(tf.split(keys, self.head_num, axis=2))
        values = tf.stack(tf.split(values, self.head_num, axis=2))

        inner_product = tf.matmul(querys, keys, transpose_b=True)
        self.normalized_att_scores = softmax(inner_product)

        result = tf.matmul(self.normalized_att_scores, values)
        result = tf.concat(tf.split(result, self.head_num, ), axis=-1)
        result = tf.squeeze(result, axis=0)


        result += tf.tensordot(inputs, self.W_Res, axes=(-1, 0))
        result = tf.nn.relu(result)
```

This layer should be called recursively, to simulate multiple heads. As we can see the aforementioned weight matrices are multiplied using the inner product in order to explore the feature interactions.

A full implementation of these codes can be found in DeepCTR[1]

4.4 Discussion

In the previous sections we shared the most relevant parts of the logic of some models. These models can be implemented in python using Tensorflow and keras libraries. The Tensorflow library runs the operations on a GPU, which is many times faster than a CPU, hence provides results in less time. This helps a lot as this models even using a GPU, can take as long as one week to train 1000 epochs depending on the data set size. Keras is an open source library for python that enables the creation and deployment of prediction models. Contains many known algorithms already implemented, there a many ways to construct a model and create a proper architecture. This library gives data scientists the tools to deploy their own layers and combined them with the existing ones, e.g the cross network combined with the deep neural network. The latter is already implemented in keras as it has deeply used in many research areas.

[1] https://github.com/shenweichen/DeepCTR

Chapter 5

Case Studies

This chapter presents our case studies, we search for CTR and CVR data sets. However we selected two data sets that correspond to CTR problems. There were many other data sets found, but too large to handle with the time in hands. For these data sets we had models run for The computations in the selected models despite running on GPU, which is faster than CPU, still take a lot of time. First step is to study the data set using the python pandas library, then we deal with the data problems found to feed the models. Different datasets require different approaches to deal with these problems. In the next sections we will present our results in these selected data sets in order to take more general conclusion.

5.1 Criteo

Display Advertising Challenge from CriteoLabs[1], this is a CTR problem found on kaggle competitions. This data set contains a portion of 7 traffic days in criteo web site and 45840617 records: 13 integer, 26 categorical columns, hashed for anonymization purposes and the label feature which we want to predict. As the csv file from this data set has no header first we create the name for each feature, then we import the data set using read csv function, specifying the separator and the column names aforementioned. Afterwards, we analyze the content by using two functions: info, with the parameter null counts equals True; describe. From the first one we can conclude the existence of NaN values in columns, which were replaced by the standard deviation for the numerical features and a preset value for the categorical features. The second one gives us the following result: count unique values, mean, standard deviation, min, the three percentiles and max; for numerical features and: count number of samples, unique, top, frequency of the top; for the categorical features. see 5.3

Studying the results of this function for both data types, we decided to normalize the integer features using MinMaxScaler[2], function belonging to sklearn. As for the categorical features due to the large vocabularies we tried one-hot encoding which did not fit into memory while computing, so our approach was the following:

[1] https://www.kaggle.com/c/criteo-display-ad-challenge
[2] learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

	I1
count	2.504706e+07
mean	3.502413e+00
std	9.429076e+00
min	0.000000e+00
25%	0.000000e+00
50%	1.000000e+00
75%	3.000000e+00
max	5.775000e+03

	C1
count	45840617
unique	1460
top	05db9164
freq	22950860

Figure 5.1: Example of function Pandas DataFrame Describe output

```
for col in categorical_columns:
        concat = pandas.concat([df_train[col], df_test[col]])
        value_counts = concat.value_count(dropna=False).to_dict()
        df_train[col] = df_train[col].map(value_counts)
        df_test[col] = df_test[feature_columns].astype(float)
```

This code, concatenates values that have the same number of occurrences. Hence, creating a sort of a ranking among the categories. From this process we attained features that can be fed to our models in order to test them. We plotted the results for each model comparing both loss and validation loss, see fig.

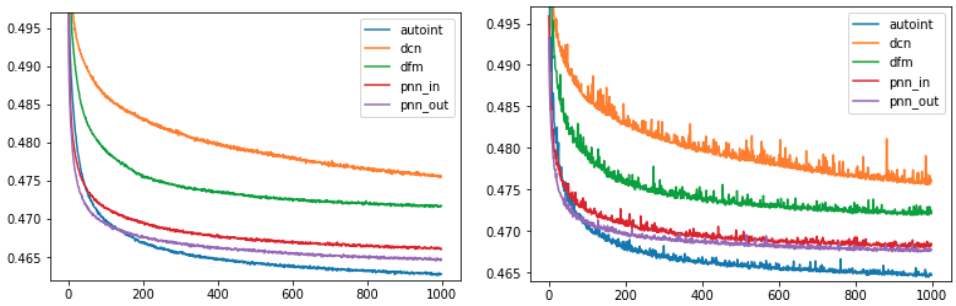

Figure 5.2: Criteo loss and validation loss, respectively - plotted for selected models

We concluded that for this data set the best model performing during the training phase was AutoInt.

5.2 Avazu

Click-through rate prediction for online advertising sponsored by Avazu. This data set contains 11 days worth of avazu data summing up to 40428968 records. Most of the variables in this data set are categorical, but some of them are encoded as numbers already. Our approach to deal with this data set was pretty similar to the previous one, we applied MinMaxScaler to normalize numerical features, and used our aforementioned rank algorithm to deal with the big vocabularies of categorical features. The difference to between the latter data set this one is the nonexistence of NaNs, and this data set has a time series. The latter type even tho mentioned before in this paper is not the focus so we handled this

feature as being categorical. We now present our results for training on our selected models.

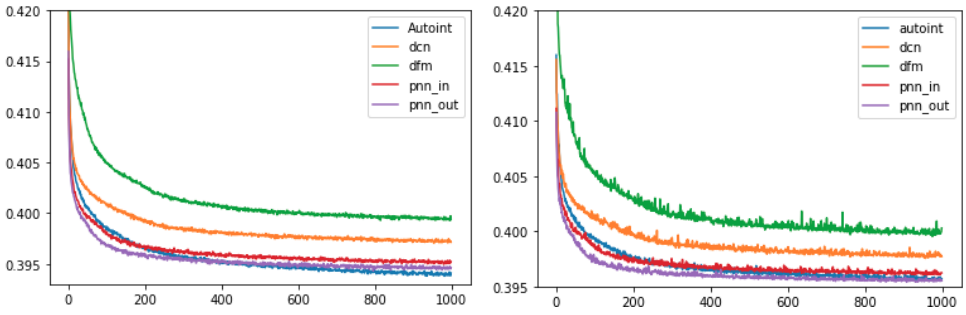

Figure 5.3: Avazu loss and validation loss, respectively - plotted for selected models

5.3 Discussion

In this figure we can conclude that both outer product network outperformed AutoInt, but we can also note that it slightly overfitting. So despite having the best performance we cannot tell with certainty that this is our best model. We acknowledge that from all the models we studied, these two had the best performance and it is interesting to note that: The approach using External product projects the incoming vector to a perpendicular space and nulls out the ones that are very close, this can be translated as the relation between the two vector being used on this operation, it is a very promising approach but is very expensive compute. On the other hand AutoInt uses attention mechanism which in sum is a computation of the relation of features under an attention head with a given operation, we used the inner product as it is cheaper to compute and gave a result very near the PNN outer product. In sum, with a better configuration under these two algorithms we could have an interesting match in terms of results, and we can also enhance (having better computational conditions) the AutoInt network by applying the outer product in the attention head and augmenting the number of attention heads. So our results for this experience even tho it allowed concluding that the outer product is stronger than inner product it was inconclusive on which model is better because for a default configuration we could not avoid the slight overfitting hence cannot conclude if for ”perfect” configurations on these algorithms which one would outperform the other.

Chapter 6

Conclusions

In conclusion to this work, the techniques which performed better for our case studies were the outer product in PNN and the inner product in the Attention network for AutoInt. Although in training these models perform better, we were not able to extract the full power from the other models. Because, training is a process that requires a lot of resources related to time and memory; We were able to train our models through 1000 epochs achieving a stable loss and validation loss values. But with some hyper-parameter tuning, and other data strategy would have given us best results for these two models. The results we achieved were not as strong as the papers introducing these algorithms. Our main hypothesis resides in the fact that we need to apply a good embedding to the data before we apply it to the model. The mathematical operations Outer and Inner product explore feature relations and are very useful to this task, but they must be combined with proper weights that learn these interactions. A model performance depends highly on the creation of these new features both low- and high-order. The best results in this area reside in weather a good representation can be found for our features. These must be in the same representation dominion so semantics is very important for this task in order to represent data in a way that their combination is meaningful. Many encoding strategies exist for different type of data, but the challenge is to find a good representation that allows these data to relate and improve the models performance throughout the different layers.

6.1 Contributions

The major achievements of the present work were the study of the difference of performance for both inner and outer product across different models. In this work we applied a study to real data sets, studying different approaches that automate feature engineering. There are many models created recently that deal with this task in an automated matter, but they require a lot of tuning. Despite this challenge, these approaches reach high accuracy. We concluded that operations such as inner and outer product have a high impact in uncovering low- and high-order feature interactions. In previous models, shallow networks this work would be done manually and it is a very time consuming task, requiring a lot of domain expertise. As these techniques were explored we came to the conclusion that a good set-up combining the

aforementioned operations and weights to keep up with the relevant combinations. Despite being very resource consuming, computation time and memory usage. It is a very promising solution to discharge some of this workload from data scientists, that with this advantage may focus on enhancing the model through hyper-parameter tuning instead of spending hours trying to combine features manually. As this task is virtually impossible for very big data sets. Plus one would not be able to come across all feature combinations. The combinatorial order achieved by these models is extremely high. In conclusion, this work introduces the reader to a simple approach on which models can help increase the task of feature engineering for CTR prediction, tackling issues like data normalization and encoding.

6.2 Future Work

Through the results achieve throughout this project, we propose to explore the possibility to enhance the AutoInt model by using a outre product network to explore the feature interaction. By changing the inner product to this layer in function ψ. Because this operation explores the orthogonality of vectors multiplication, our research showed us that this property is very powerful when it comes to explore high-order feature interactions.

Bibliography

[1] F. Chollet. *Deep Learning with Python*. Manning Publications, 1st edition, 2017.

[2] Q. Liu, F. Yu, S. Wu, and L. Wang. A convolutional click prediction model. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 1743–1746. ACM, 2015.

[3] R. Wang, B. Fu, G. Fu, and M. Wang. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*, ADKDD'17, pages 12:1–12:7. ACM, 2017.

[4] Y. Shan, T. R. Hoens, J. Jiao, H. Wang, D. Yu, and J. Mao. Deep crossing: Web-scale modeling without manually crafted combinatorial features. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 255–262. ACM, 2016.

[5] H. Guo, R. Tang, Y. Ye, Z. Li, X. He, and Z. Dong. DeepFM: An end-to-end wide & deep learning framework for CTR prediction. *arXiv:1804.04950*, 2018.

[6] S. Rendle. Factorization machines. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 995–1000. IEEE Computer Society, 2010.

[7] Y. Qu, H. Cai, K. Ren, W. Zhang, Y. Yu, Y. Wen, and J. Wang. Product-based neural networks for user response prediction. *arXiv:1611.00144*, 2016.

[8] S. Rendle. Factorization machines with libFM. *ACM Trans. Intell. Syst. Technol.*, 3(3):57:1–57:22, 2012.

[9] W. Zhang, T. Du, and J. Wang. Deep learning over multi-field categorical data: A case study on user response prediction. *arXiv:1601.02376*, 2016.

[10] A. P. Ta. Factorization machines with follow-the-regularized-leader for ctr prediction in display advertising. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2889–2891, 2015.

[11] G. E. Hinton. *A Practical Guide to Training Restricted Boltzmann Machines*, pages 599–619. Springer, 2012.

[12] Y. Bengio, L. Yao, G. Alain, and P. Vincent. Generalized denoising auto-encoders as generative models. In *Advances in Neural Information Processing Systems 26*, pages 899–907. Curran Associates, Inc., 2013.

[13] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: Estimating the click-through rate for new ads. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 521–530. ACM, 2007.

[14] Z. Pan, E. Chen, Q. Liu, T. Xu, H. Ma, and H. Lin. Sparse factorization machines for click-through rate prediction. In *16th IEEE International Conference on Data Mining (ICDM)*, pages 400–409, 2016.

[15] G. Zhou, C. Song, X. Zhu, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai. Deep interest network for click-through rate prediction. *arXiv:1706.06978*, 2017.

[16] W. Song, C. Shi, Z. Xiao, Z. Duan, Y. Xu, M. Zhang, and J. Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. *CoRR*, 2018.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf`.

[18] W. Lu, S. Kawasaki, and J. Sakuma. Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data. *IACR Cryptology ePrint Archive*, 2016:1163, 2016.

[19] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. DeepFM: A factorization-machine based neural network for CTR prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1725–1731, 2017.

Appendix A

Mathematical Definitions

A.1 Softmax

Activation function used in Deep Learning for classification problems, it's used in order to obtain normalized values in the 0-1 range.

$$P(y = c \| \mathbf{x}; \mathbf{w_c}; \mathbf{b_c}) = softmax_c(\mathbf{x^t w_c} + b_c) = \frac{e^{\mathbf{x^t w_c} + b_c}}{\sum_j |e^{\mathbf{x^t w_j} + b_j}|} \tag{A.1}$$

A.2 bag of words

In language processing the bag of words technique is used to make a vector out of a phrase or document, in which the words are represented as the number of times it appears in the phrase or document.

e.g John likes to watch movies, Mary likes to watch movies too - would be encoded as:

{john: 1, likes: 2, to: 2, watch: 2, movies: 2, too: 1}

this simplification allows the algorithm to store a dictionary of the words and their occurrences in text, saving some space and facilitating the computations between phrases or documents. comparisons

A.3 Confusion Matrix

Confusion matrix or Error Matrix, is a table that helps us uncover between how many actual values and predicted values exist for different classes that the model will predict. This concept is very important because shows how reliable a model is, this can be done through various metrics such as accuracy, precision, etc.

	Actual Positive	Actual Negative
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

Figure A.1: Confusion matrix