IMPERIAL

Department of Mathematics

Deep Reinforcement Learning for Ad Personalization

Martin Batěk

CID: 00951537

Supervised by Mikko Pakkanen

2 September 2024

Submitted in partial fulfilment of the requirements for the
MSc in Machine Learning and Data Science of
Imperial College London

The work contained in this thesis is my own work unless otherwise stated.

Signed: Martin Batěk Date: 17 July 2024

Abstract

ABSTRACT GOES HERE

Acknowledgements

ANY ACKNOWLEDGEMENTS GO HERE

Contents

1. Introduction

The global digital advertising market is worth approximately $602 billion today. Due to the increasing rate of of online participation since the COVID-19 pandemic, this number has been rapidly increasing and is expected to reach $871 billion by the end of 2027 (eMarketer, 2023). Many of the of the major Ad platforms such as Google, Facebook and Amazon operate on a cost-per-user-engagement pricing model, which usually means that advertisers get charged for every time a user clicks on an advertisment. This means that there is a significant commercial incentive to design Ad-serving platforms that ensure that the content shown to each user is as relevent as possible, so as to maximize user engagement and platform revenues as much as possible.

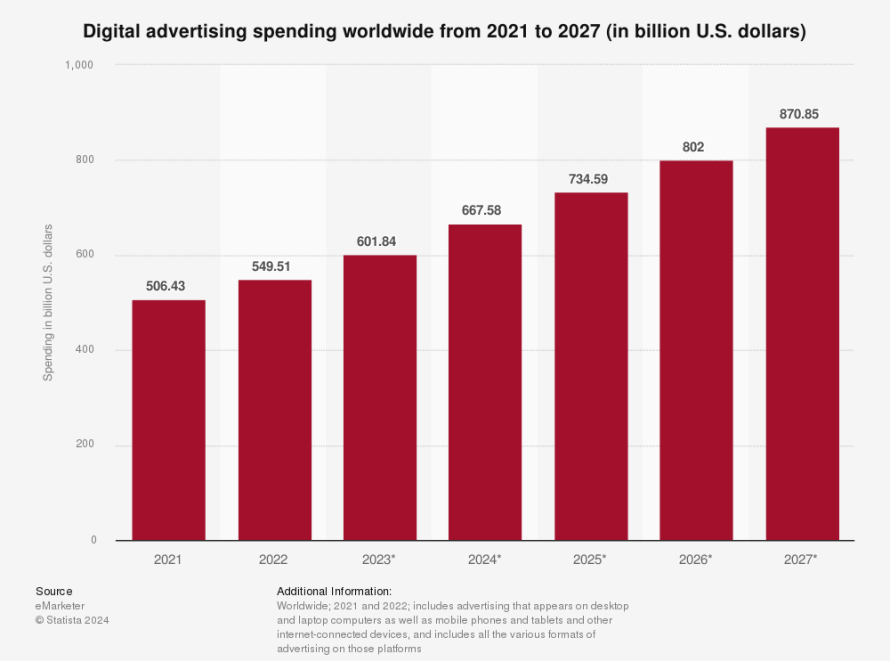

Figure 1.1.: Global Digital Ad Spending 2021-2027. Image taken from eMarketer (2023)

Attaining accurate Click-Through Rate (CTR) prediction is a necessary first step for Ad persionalization, which is why study of CTR prediction methods have been an extremely active part of Machine Learning research over the past through years. Initially, shallow prediction methods such as Logistic Regression, Factorization Machines (Rendle, 2010) and Field-Aware Factorization Machines (Juan et al., 2016) have been used for CTR prediction. However, these methods have often been shown to be unable to capture

the higher order feature interactions in the sparse multi-value categorical Ad Marketplace datasets (Zhang et al., 2021). Since then, Deep Learning methods have been shown to show superior predictive ability on these datasets. A number of Deep Learning models have been proposed, each using a different techniques for feature interaction modelling, ranging from Deep Learning extensions of Factorization Machines such as DeepFM (Guo et al., 2017), to novel methods such as AutoInt (Song et al., 2019). By employing a multi-towered neural network architecture, these models are able to capture both low-order and high-order feature interactions in the data, and therefore tend to achieve supperior predictive performance to their shallow counterparts.

However, irrespective of how well these models perform in a static environments, the reality is that user preferences and advertisment characteristics are constantly changing. Like most online reccomender systems, Ad personalization models must be able to adapt to these changes in order to continue to provide accurate predictions over the longer period (Zheng et al., 2018). This problem necessitates the use of Reinforcement Learning for Ad personalization.

Reinforcement Learning is a subdomain of Machine Learning in which the goal is for an agent to learn an optimal policy that maximizes the expected reward in an environment where the state-action-reward progression can be modelled as a Markov Decision Process (Puterman, 2014). Early Reinforcement Learning methods involved deriving a the transition probabilities for the state-action pairs on the basis of interactions with the environment and then using Dynamic Programming methods such as the Upper Confidence Bound RL (UCB-RL) algorithm (Auer et al., 2008) and the the Thompson Sampling algorithm for Reinforcement Learning (Pike-Burke, 2024a). However, in cases where the state-action space is too sparce to be reasonably enumerated, it is often more practical to user a function approximator to directly estimate the expacted cumalative reward for each action in each state. This method of Reinfocement Learning is commonly referred to as Q-learning (Watkins, 1989), and has the advantage of being *model-free*, meaning that it does not require the agent to have a model of the environment thereby making it more scalable to large and sparce datasets. In (Hornik et al., 1989), (Cybenko, 1989) and (Hornik et al., 1990) Deep Neural Networks with activation functions are shown to be universal function approximators which naturally lead to the incorporation of DNN's in Q-Learning. This has lead to the development of the Deep Q-Learning Agent, which has been shown to be able to learn optimal policies in a number of different domains, such as the Atari 2600 game environment Mnih et al. (2015). Beyond this, Deep Reinforcement Learning has shown promising results in a number of different aplications, including robot control and computer vision (Wang et al., 2024). In the context of Ad personalization, DRL has also be applied to online reccomender systems such as News article reccomendation (Zheng et al., 2018) and video reccomendation on Youtube (Chen et al., 2019). In both papers, the authors show that the DRL agent is able to learn an optimal content reccomendation policy on the basis of user engagement data. This reveals that there is potential for applying these methods to the problem of Ad personlization, thereby creating a truely adaptive marketing platform.

Research Question and Contributions

In this report, I aim to construct a Ad serving system that is truely adaptive and personalized to the changing user preferences and advertisment charateristics. In order to achieve this goal, I will first need to find a suitable Deep Learning Model arcitecture for CTR prediction, and then incorporating this model as the Q-function approximator in a Deep Q-Learning algorithm. The key contributions that I make in this report are as follows:

- I evaluate the performance of five popular Deep Learning models for CTR prediction on three well-known benchmark datasets, Criteo (Tien et al., 2014), KDD12 (Aden, 2012) and Avazu (Wang and Cukierski, 2014).
- I construct a novel Deep Reinforcement Learning Frame for Ad personalization, and as a proof-of-concept and evaluate its performance using the KDD12 dataset.

Structure of the Report

In chapter 2, I begin by providing a background introducing the problem of Click-Through Rate prediction in the context of Ad personalization, and explore the unique challenges posed by the typically sparse multi-value categorical datasets that are common in the Ad marketplace. I then proceed to review the literature on Deep Learning models for CTR prediction, highlighting the different techniques that each framework uses to capture the key feature interactions in the data. I also review the literature on Deep Reinforcement Learning, specifically the DRN algorithm introduced by Zheng et al. (2018), which can be analogously applied to the Ad personalization context. In chapter 3, I evaluate the performance of different Deep Learning models for CTR prediction on three well-known benchmark datasets, Criteo (Tien et al., 2014), KDD12 (Aden, 2012) and Avazu (Wang and Cukierski, 2014). In chapter 4, I construct a Deep Reinforcement Learning model for Ad personalization and evaluate its performance on the same benchmark datasets. Finally, in chapter 5, I discuss the results of the experiments and provide some concluding remarks.

2. Background

2.1. Deep CTR Prediction

2.1.1. Problem Formulation and Ad Marketplace Data

In their respective surveys on the use of Deep Learning methods for CTR prediction, Gu (2021) and Zhang et al. (2021) outline the problem of CTR prediction as one that essentially boils down to a binary (click/no-click) classification problem utilizing user/ad-view event level online session records. The goal of CTR prediction is to train a function f that takes in a set of ad marketplace features $\mathbf{x} \in \mathbb{R}^n$, and maps these to a probability that the user will click on the ad in that given context. In other words, $f_\Theta : \mathbb{R}^n \to \mathbb{R}$ such that:

$$\mathbb{P}(\text{click}|\mathbf{x}) = \mathbb{P}(y = 1|\mathbf{x}) = \sigma(f_\Theta(\mathbf{x})) \tag{2.1}$$

where y is the binary click label, Θ represents the parameter vector for f and $\sigma(x) = (1 + e^{-x})^{-1}$ is the sigmoid function. To ease the notation for the rest of the report, we will use the shorthand $p(y) = \mathbb{P}(y = 1|\mathbf{x})$ formulations in the following sections.

An instance of the ad marketplace features $\mathbf{x}$ is typically recorded at a user/ad impression event level and typically consists of

- **User Features:** Features that describe the user, such as User ID, demographic information, metrics related to the user's past interactions with the platform, etc.
- **Ad Features:** Features that describe the ad, such as Ad ID, Advertiser ID and Ad Category.
- **Contextual Features:** Features that describe the context in which the ad is being shown, such as the time of day, the position of the ad on the page and the site on which the ad is being shown.

Figure 2.1 shows a snapshot of the KDD12 dataset, which is a typical example of the type of data that is used for CTR prediction.

A defining characteristic for this type of data is that many of the features are multi-value categories with a high degree of of cardinality (He and Chua, 2017). In order to user categorical data in a classifier model, it is common practice to embed the these categorical features as multidimensional vectors. While the dimensionality of these embeddings can vary amongt the different sparse categorical features, for the sake of simplicity of notation, below we assume that all sparse categorical feature embeddings have the same dimensionality, D. Let $\mathbf{x_i}^{OH}$ be the one-hot encoded vector representation of the

Click	Impression	DisplayURL	AdID	AdvertiserID	Depth	Position	QueryID	KeywordID	TitleID	DescriptionID	UserID
0	1	4.29812E+18	7686695	385	3	3	1601	5521	7709	576	490234
0	1	4.86057E+18	21560664	37484	2	2	2255103	317	48989	44771	490234
0	1	9.70432E+18	21748480	36759	3	3	4532751	60721	685038	29681	490234
0	1	1.36776E+19	3517124	23778	3	1	1601	2155	1207	1422	490234
0	1	3.28476E+18	20758093	34535	1	1	4532751	77819	266618	222223	490234
0	1	1.01964E+19	21375650	36832	2	1	4688625	202465	457316	429545	490234
0	1	4.20308E+18	4427028	28647	3	1	4532751	720719	3402221	2663964	490234
0	1	4.20308E+18	4428493	28647	2	2	13171922	1493	11658	5668	490234
0	1	5.85475E+17	20945590	35083	2	1	35143	28111	151695	128782	490234
0	1	9.68455E+18	21406020	36943	2	2	4688625	202465	1172072	973354	490234
	Target		Ad Features			User Features			Contextual Features		

Figure 2.1.: Snapshot of the KDD12 dataset Aden (2012)

categiorical feature x_i. Then the *embedded* feature vector $\mathbf{e}_i$ for categorical feature x_i is given by:

$$\begin{aligned}\mathbf{e}_i &= \mathbf{B}_i \mathbf{x}_i^{OH} \\ &= \left[\mathbf{b}_1^i, \ldots, \mathbf{b}_{m_i}^i\right] \mathbf{x}_i^{OH} \\ &= \begin{bmatrix} b_{1,1}^i & \cdots & b_{1,C_i}^i \\ \vdots & \ddots & \\ b_{D,1}^i & \cdots & b_{D,C_i}^i \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \end{bmatrix}\end{aligned} \tag{2.2}$$

where $\mathbf{B}_i$ is the embedding matrix for feature x_i, whose dimensions are determined by the chosen embedding dimension D and the cardinality of the feature, C_i. Assuming that value of x_i is equal to the k-th value in the one-hot encoding mapping, and that therefore the k-th value in $\mathbf{x}_i^{OH}$ is equal to one, Equation 2.2 then simplifies to

$$\mathbf{e}_i = [e_{i1}, \ldots, e_{i,D}]^\intercal = \left[b_{1k}^i, b_{2k}^i, \ldots, b_{Dk}^i\right]^\intercal = \mathbf{b}_k^i \tag{2.3}$$

which is the k-th column of the embedding matrix $\mathbf{B}_i$, otherwise referred to as the k-th embedding vector (Hancock and Khoshgoftaar, 2020). The processed data $\tilde{\mathbf{x}}$ that then gets fed into the model is then composed of a concatenation of all sparse feature embeddigs $\mathbf{e}_i$ and standardized dense numerical feature values $z_i = (x_i - \bar{x_i})/\sqrt{Var(x_i)}$:

$$\begin{aligned}\tilde{\mathbf{x}} &= [\mathbf{e}_1^\intercal, \ldots, \mathbf{e}_s^\intercal, z_{s+1}, \ldots, z_{s+d}] \\ &= [e_{1,1}, \ldots, e_{1,D}, e_{2,1}, \ldots, e_{s,D}, z_{s+1}, \ldots, z_{s+d}] \\ &= [\tilde{x}_1, \ldots, \tilde{x}_{\tilde{n}}]\end{aligned} \tag{2.4}$$

where s and d are the number of *sparse* categorical features and *dense* numerical features respectively in $\mathbf{x}$, and $\tilde{n} = D \cdot s + d$ is the resulting dimensionality of $\tilde{\mathbf{x}}$. Again, to ease the notation in the remainder of the report, we will assume that the prepocessing steps described above are applied to the data, and all formulaic expressions of the models

in the following section with be expressed in terms of $\tilde{\mathbf{x}} = \{\tilde{x}_j\}_{j=1}^{\tilde{n}}$.

The problem posed by high cardinality is that when m_i is large, the high sparsity of the one-hot encoded vector x_i^{OH} can make in extremely difficult for a model to learn the key *implicit* features and patterns present in the data (Gu, 2021). This is indeed the key challenge in building an accurate CTR prediction model, and is a key motivating factor as to why Deep Neural networks have out performed the classical shallow counterparts. This transition will be examined in more detail in the following section.

2.1.2. Shallow CTR Models

Logistic Regression

The earliest examples of CTR classification models incorporated classical "shallow" (single layer) statistical regression methods. The most basic example of this was the **Logistic Regression** model, as implemented by Richardson et al. (2007) on advistisment data from the Microsoft Search engine. The LR model is composed by modelling the *log-odds* (also referred to as the *logit*) of a positive binary label as a linear combination of all of the respective feature values:

$$f_\Theta^{LR}(\tilde{\mathbf{x}}) = \theta_0 + \sum_{j=1}^{\tilde{n}} \theta_j \tilde{x}_j \tag{2.5}$$

where $f_\Theta^{LR} : \mathbb{R}^n \to \mathbb{R}$ represents the Logistic regression model parametized by $\Theta = (\theta_0, \ldots, \theta_{\tilde{n}})$. The benefits of the LR model are that due to its simplicity and low number of parameters, it is relatively easy to train in computational terms and also relatively easy to deploy (Zhang et al., 2021). Howevever, the formulation in equation 2.5 reveals that the LR model does not explicitly account for *feature interactions*. As outlined in section 2.1.1, the categorical features tend to have a high cardinality, resulting in highly sparse feature embeddings. Many of the of the important patterns for CTR prediction are therefore likely to be expressed in terms of *combinations of features* rather then the individual feature values themselves. For example, a user's tendancy to click on a given advertisment is likely to by influenced by the *combination* of the category of good or service the given advertisment is trying to sell (e.g. premium fashion retail, travel, electronics et. cetera) and the demographic/socio-economic category that the given user falls into (e.g. university student, young professional, retiree). These features combinations (and the corresponding combination of respective field values in the preprocessed feature vector $\tilde{\mathbf{x}}$) are commonly referred to in the literature as *cross-features* (Zhang and Zhang, 2023) or more commonly *feature interactions* (Cheng et al., 2016; Song et al., 2019; Xiao et al., 2017).

Whilst it is possible to incorporate feature interactions in an LR model through feature engineering, this quickly becomes infeasible for large sparse datasets. A number of techniques have been developed to automate the necessary feature engineering steps for this, either by implicitly assigning a weight to all second order feature interactions (Chang et al., 2010) or by utilizing Gradient Boosted Decision Trees to pick out the key

interactions (Cheng et al., 2014). Unfortunately, the prior still tends to exhibit poor performance with sparse data, whereas the fact that the Gradient Boosting algorithm for the latter is difficult to paralellize makes this solution difficult to scale in many applications in practice (Zhang et al., 2021).

Factorization Machines

Factorization Machines first proposed by Rendle (2010) can be thought of as an extension of the Logistic Regression framework in equation 2.5 with additional terms that explicitly account for the interactions between different features. Its relative simplicity and computational scalability has made it a widely popular framework for CTR modelling (Gu, 2021). A 2-way (maximum feature interaction degree of 2) Factorization Machine model is formulated as:

$$f_{\Theta}^{FM^2}(\tilde{\mathbf{x}}) = \theta_0 + \sum_{j=1}^{\tilde{n}} \theta_j \tilde{x}_j + \sum_{j=1}^{\tilde{n}} \sum_{k=j+1}^{\tilde{n}} \langle \mathbf{v}_j, \mathbf{v}_k \rangle \tilde{x}_j \tilde{x}_k \tag{2.6}$$

where $\langle \cdot, \cdot \rangle$ represents the inner product between two vectors, the final interaction term above is parametized by $\mathbf{V} \in \mathbb{R}^{\tilde{n} \times F}$. Each row $\mathbf{v}_j$ of $\mathbf{V}$ represents the j-th feature in $\tilde{\mathbf{x}}$ in terms of F latent factors. The factorization matrix $\mathbf{V}$ is typically fitted by optimizing the binary cross-entropy loss function by means of Stochastic Gradient Descent. It is intutive to see that $\mathbf{V}$ will be fitted such that if the interaction between feature j and k have a positive impact on $p(y)$, then j-th and k-th rows of $\mathbf{V}$ will have positive inner products (and vice versa) (Zhang et al., 2021).

Rendle (2010) shows that although direct evaluation of equation 2.6 would appear to have a complexity of $O(F\tilde{n}^2)$, the 2-way FM model in fact scales linearly in $\tilde{n}$ and F:

Lemma 2.1.1. The model equation of a 2-way factorization machine (eq. 2.6) can be computed in linear time $O(F\tilde{n})$.

Proof.

Due to the factorization of the pairwise interactions, there is no model parameter that directly depends on two features (j, k). This means that pairwise interactions can be reformulated as such

$$
\begin{aligned}
&\sum_{j=1}^{\tilde{n}} \sum_{k=j+1}^{\tilde{n}} \langle \mathbf{v}_j, \mathbf{v}_k \rangle \tilde{x}_j \tilde{x}_k \\
&= \sum_{j=1}^{\tilde{n}} \sum_{k=j+1}^{\tilde{n}} \sum_{f=1}^{F} v_{j,f} v_{k,f} \tilde{x}_j \tilde{x}_k \\
&= \frac{1}{2} \left(\sum_{j=1}^{\tilde{n}} \sum_{k=1}^{\tilde{n}} \sum_{f=1}^{F} v_{j,f} v_{k,f} \tilde{x}_j \tilde{x}_k - \sum_{j=1}^{\tilde{n}} \sum_{f=1}^{F} v_{j,f} v_{j,f} \tilde{x}_j \tilde{x}_j \right) \\
&= \frac{1}{2} \sum_{f=1}^{F} \left(\sum_{j=1}^{\tilde{n}} v_{j,f} \tilde{x}_j \sum_{k=1}^{\tilde{n}} v_{k,f} \tilde{x}_k - \sum_{j=1}^{\tilde{n}} v_{j,f}^2 \tilde{x}_j^2 \right) \\
&= \frac{1}{2} \sum_{f=1}^{F} \left(\left(\sum_{j=1}^{\tilde{n}} v_{j,f} \tilde{x}_j \right)^2 - \sum_{j=1}^{\tilde{n}} v_{j,f}^2 \tilde{x}_j^2 \right)
\end{aligned}
$$

The complexity of the final line above is $O(F\tilde{n})$, and hence the FM formulation as per equation 2.6 scales linearly in F and $\tilde{n}$. □

This quality greatly simplifies the computational complexity of scaling the FM model to larger datasets with a more sparse categorical features. Morever, the Factorization Machine framework can be generalized to degree R (i.e. up to any limit of feature interation order) as follows:

$$
f_{\Theta}^{FM^R} \quad = \quad \theta_0 \; + \; \sum_{j=1}^{\tilde{n}} \theta_j \tilde{x}_j \; + \; \sum_{r=1}^{R} \sum_{j_1=1}^{\tilde{n}} \cdots \sum_{j_r=j_{r-1}+1}^{\tilde{n}} \left(\prod_{k=1}^{r} \tilde{x}_{j_k} \right) \left(\sum_{f=1}^{F_r} \prod_{k=1}^{r} v_{j_k,f}^{(r)} \right) \tag{2.7}
$$

The FM framework therefore provides an intuitive and computationally scalable method to account for key feature interactions without the need of extensive feature engineering. Extensions and improvements to FM have been proposed, most notably in the form of the Field-Aware Factorization Machine (FFM) framework by Juan et al. (2016), which only acounts for intereactions between features of different fields (in otherwords, it ignores the interaction between $\tilde{x}_j$ and $\tilde{x}_k$ if both are componets of embedding vector $mathbfe_i$ for some categorical feature x_i) as well as Gradient Boosted Factorization Machines (Cheng et al., 2014), which again aims to augment the FM framework by means of the Gradient Boosting algorithm.

2.1.3. Introducing MLP's in CTR prediction

Despite the advantages of the FM framework, a setback of the formulation in equation 2.7 is that the framework grows highly compex and overparametized for higher values of R. As a consequence, only the 2-way FM framework as per equation 2.6 tends

to be implemented in practice, meaning that the FM model alone is practically insufficient for capturing feature interactions of order >> 2 (Guo et al., 2017). Deep Neural Networks present a powerful alternative for addressing this shortcoming. Neural networks benefit from being universal function approximators (Cybenko, 1989) and from the fact that neural network batch training is paralellizable by means of GPU accelerated computation. This has lead to the successfull application of Deep Learning algorithms across multiple fields such as Natural Language Processing and Image classification (He et al., 2016; Krizhevsky et al., 2017; LeCun et al., 1998). These factors and successes showed that DNN's have the potential to extract informative feature representations from highly sparse and abstract data, and as a consequence, the application of DNN's in CTR prediction started recieving attention in the mid-2010's.

The **Multilayer Perceptron** (MLP) is the most elementary type of Deep Neural Network (Webster, 2024). In general, a MLP with L hidden layers is formulated as such:

$$\mathbf{h}^{(0)} := \tilde{\mathbf{x}} \tag{2.8}$$

$$\mathbf{h}^{(l)} = \phi_l \left(\mathbf{W}^{(l-1)} + \mathbf{b}^{(l-1)} \right), l = 1, \ldots, L \tag{2.9}$$

$$\hat{y} = \phi_{out} \left(\mathbf{w}^{(L)} \mathbf{h}^{(L)} + b^{(L)} \right) \tag{2.10}$$

where $\mathbf{W}^{(k)} \in \mathbb{R}^{n_{l+1} \times n_l}$, $\mathbf{b}^{(k)} \in \mathbb{R}^{n_{l+1}}$, $\mathbf{h}^{(l)} \in \mathbb{R}^{n_l}$, $n_0 = \tilde{n}$, n_l is the number of hidden units in layer l and ϕ_l is the activation function for layer l, When rearranged in the form of equation 2.1, the above becomes:

$$f_{\Theta}^{MLP}(\tilde{\mathbf{x}}) = \psi_{out} \left(\psi_L \left(\cdots \psi_1 \left(\tilde{\mathbf{x}} \right) \cdots \right) \right) \tag{2.11}$$

where each function ψ_l represents the affine transformation and element-wise activation operation for layer l. MLPs can be thought of as an acyclic graph, as displayed in Figure 2.2. The data $\tilde{\mathbf{x}}$ first gets fed through the *input layer*, then gets processed by multiple *hidden layers* that include a series of affine transfromations followed by activation functions, before the final result is produced by the *output layer*.

Figure 2.2 demonstrates the potential that DNN's have for modelling higher order feature interactions. By training the network by means of Stochastic Gradient Descent, it should be possible to calculate the appropriate weight ($\mathbf{W}_l$) and bias ($\mathbf{b}_l$) parameters in order capture the relevant high-order feature patterns in the data. As such, many CTR modelling frameworks have been developed that use Deep Learning techniques to build upon and imporved the previously discussed classical methods by incorporating Deep Neural Networks such as the ML in the model architecture (Zhang et al., 2021).

2.1.4. Single vs Dual Tower Architectures

Before moving on to DNN enhanced CTR models in section 2.1.5 it is worth briefly discussing the difference between **Single-Tower** models and **Dual-Tower** architecture models. Single Towel models place all layers successively in the architecture, and can

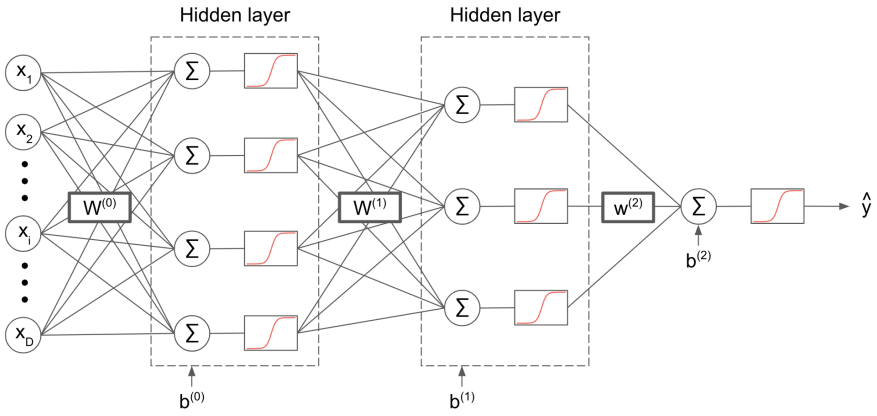

Figure 2.2.: Multilayer Perceptron with two hidden layer. Taken from (Webster, 2024)

generally be formulated as in equation 2.11. Since all feature inputs are passed through the same set of successive affine transformations and activations, Single Tower models are usually able to capture higher order feature interactions, but the signal from the low-order interactions tend to be lost (Zhang et al., 2021). The FNN (Zhang et al., 2016), FGCNN (Liu et al., 2019) and PNN (Qu et al., 2016) models covered in the subsections 2.1.5 and 2.1.6 are examples of Single Tower models.

In order to avoid diluting the signal from the lower order feature interactions, many architectures adopt a Dual Tower architecture, as shown on the right-hand side of Figure 2.3. A separate **Feature Interaction Layer** is placed paralelly to the DNN, and the final output is composed of a weighted sum of the feature interaction and DNN outputs. With this architecture, the feature interaction layer is usually dedicated to capturing the important lower order feature interaction signals, whereas the DNN acts as a *residual network* for capturing any meaningful signals that may be missing. As a result, Dual Tower networks tend to benefit form better training stability and better performance (Zhang et al., 2021).

2.1.5. DNN Enhanced CTR models

Factorization-machine Supported Neural Networks

One of the earliest example of the use of Deep Neural Networks being used to enhance existing CTR modeling methods is the **Factorization-machine Supported Neural Network** (FNN) (Zhang et al., 2016). The FNN model is a Single Tower model that works by pretraining a 2-way Factorization Machine model as in equation 2.6 on the concatenated one-hot encoded categorical feature vectors, and then using the feature interaction vectors and weights as the embedding matrix.

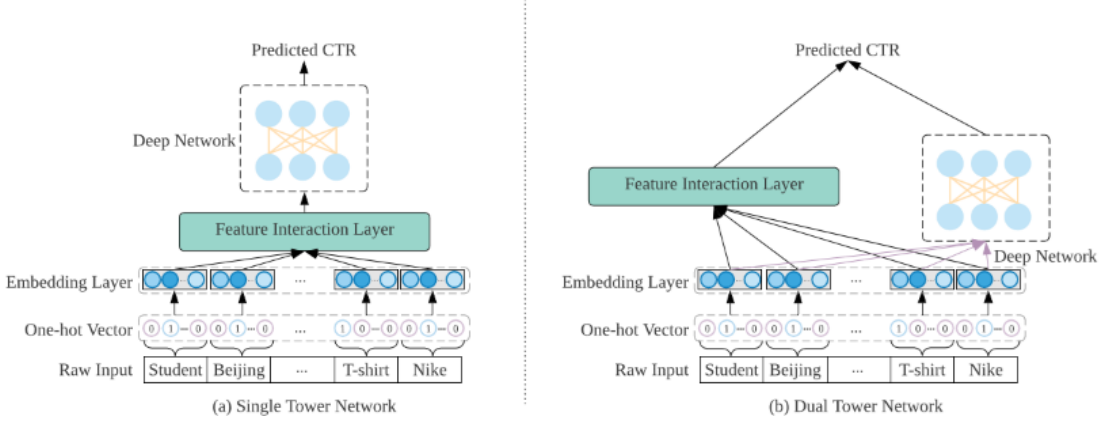

Figure 2.3.: Single vs Dual Architecture Networks. Source: (Zhang et al., 2021)

Below we start with the concatenated one-hot encoded categorical feature vector:

$$\dot{\mathbf{x}} = \left[\mathbf{x}_1^{OH}, \ldots, \mathbf{x}_n^{OH}\right]$$

$$f_\Theta^{FM^2} = \theta_0 + \sum_{j=1}^{\dot{n}} \theta_j \dot{x}_j + \sum_{j=1}^{\dot{n}} \sum_{k=j+1}^{\dot{n}} \langle \mathbf{v}_j, \mathbf{v}_k \rangle \dot{x}_j \dot{x}_k$$

Using the weights and biases from $\Theta = (\theta_0, \theta_1, \ldots, \theta_{\dot{n}}, \mathbf{v}_1, \ldots, \mathbf{v}_{\dot{n}})$, the preprocessed MLP input is calculated as follows:

$$\tilde{\mathbf{x}} = [\theta_0, \mathbf{e}_1^\intercal, \mathbf{e}_2^\intercal, \ldots, \mathbf{e}_s^\intercal]$$

where each embedding vector $\mathbf{e}_i$ is defined as the concatination of the weight (θ_i) and FM interaction vector ($\mathbf{v}_i$) corresponding to feature i in $\dot{\mathbf{x}}$:

$$\mathbf{e}_i = [\theta_i, \mathbf{v}_i^\intercal]$$

The above can alternatively also be recovered from equation 2.2 by setting the embedding matrix $\mathbf{B}_i$ to a $(F+1) \times C_i$ matrix with the following values

$$\mathbf{B}_i = \begin{bmatrix} b^i_{1,1} & \cdots & b^i_{1,C_i} \\ \vdots & \ddots & \\ b^i_{F+1,1} & \cdots & b^i_{F+1,C_i} \end{bmatrix} = \begin{bmatrix} \theta_i & \cdots & \theta_i \\ v^1_1 & \cdots & v^1_{C_i} \\ \vdots & \ddots & \\ v^F_1 & \cdots & v^F_{C_i} \end{bmatrix}$$

Figure 2.4 portrays the structure of this model as it was presented in the original paper by Zhang et al. (2016).

By incorporating the FM model feature interaction vectors in the embedding layer before the DNN, the FNN model is able to leverage the FM model's strength in interaction identification. The FNN model then leverages a MLP network to capture the higher

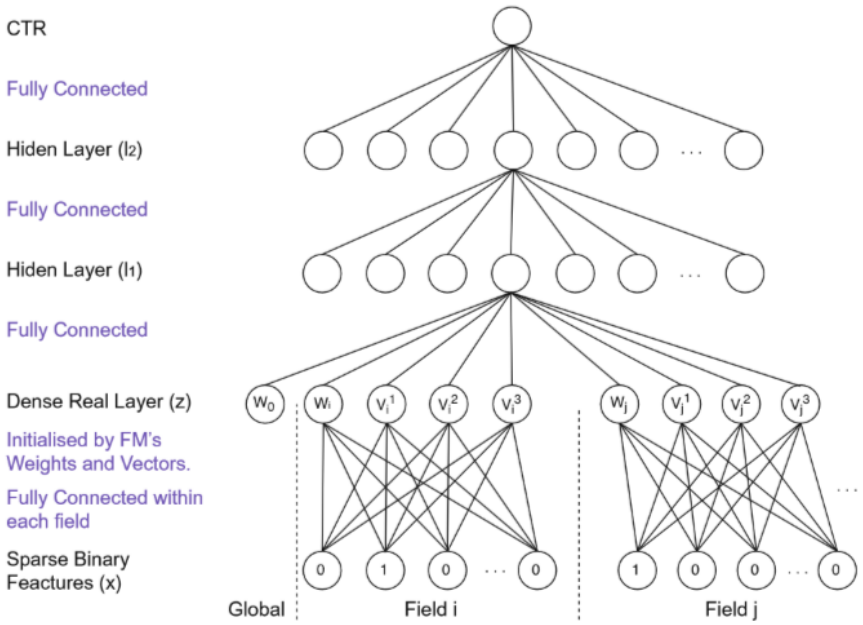

Figure 2.4.: FNN model architecture. Source: (Zhang et al., 2016)

order interaction signals much more efficiently than would have been feasibly possible when relying ownly on FM. Because of this, comprehensive experiments carried out by Zhang et al. (2016) confirmed that FNN has superior CTR estimation performance than both LR and FM.

However Guo et al. (2017) and Zhang et al. (2021) find that due to its Single Tower architecture, lower-order feature interaction signals tend to be lost in the network. Guo et al. (2017) further found that the FM pretraining step described above represents a significant overhead in terms of training efficiency. In the next subsections, we will see how the Wide & Deep and DeepFM models aim to solve for these issues.

Wide and Deep

The **Wide and Deep** (W&D) model was developed by Cheng et al. (2016).

$$f_{\Theta}^{W\&D} = \theta_0 + \sum_{k=1}^{\hat{n}} \theta_k \hat{\mathbf{x}}_k + f_{\Phi}^{MLP}(\tilde{\mathbf{x}}) \tag{2.12}$$

Figure 2.5 reveals that the W&D model is composed with a Dual-Tower Architecture, with a Deep Component and a Wide Component (shown on the left and right hand sides of Figure 2.5 respectively). The Deep Component is composed of a MLP with multiple hidden layers, each with the Rectified Linear Unit (ReLU) activation function. The Wide Component is formulated by the first two terms in equation 2.12, and is composed of a simple linear transformation of the input features. The key aspect of the

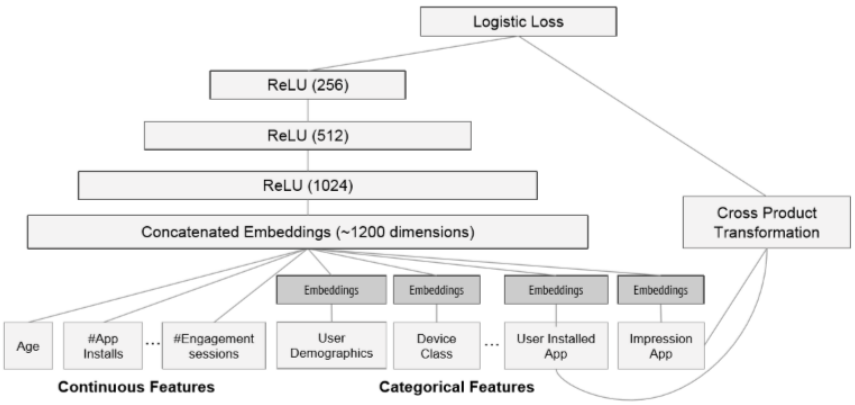

Figure 2.5.: Wide and Deep Model, as illustrated in (Shen, 2017)

Wide Component is the fact that the linear transformation is not simply applied to the preprocessed features $\tilde{x}$, but instead to these concatinated with a set of cross-product transformed features. In other words:

$$\hat{\mathbf{x}} = [\tilde{\mathbf{x}}, \upsilon_1(\tilde{\mathbf{x}}), \ldots, \upsilon_P(\tilde{\mathbf{x}})] \tag{2.13}$$

Where $\upsilon_k(\tilde{\mathbf{x}}) = \prod_{j=1}^{\tilde{n}} \tilde{x}_j^{c_{kj}}$ and $c_{kj} \in \{0, 1\}$.

Consequentially of equation 2.13, the Wide Component *memorizes* the key feature interations that are defined by the specific *cross-product transformations* ($\upsilon_k(\tilde{\mathbf{x}})$) (Cheng et al., 2016). Meanwhile, the Deep Component captures any residual signals that may not have been explicitly included in the manually defined cross-product transformations (Zhang et al., 2021). In this sense, the W&D model overcomes the shortcomings of the FNN model, by having dedicated pathways in the archtecture for higher and lower order feature interactions. However, the downside of W&D is the fact that the feature interactions in the Wide component need to be manually incorporated by defining the cross-product transformation functions $\{\upsilon_k(x)\}_{k=1}^{P}$. This means that there is a significant feature engineering component that would be necessary to use this model effectively.

DeepFM

The **DeepFM** model was developed by Guo et al. (2017) in order to address the shortcomings of the FNN and W&D models mentioned in this section, as well as those of the PNN model which will be covered in section 2.1.6. Similarly to the W&D model, the DeepFM network two components arranged as a Dual Tower architecture, as visualized in Figure 2.6.

The *FM component* effectively replaces the Wide component in the W&D model. It consists of a 2-way Factorization Machine layer that models pairwise feature interactions between the different fields of $\tilde{\mathbf{x}}$ as inner products of the respective feature latent vectors $\mathbf{v}_j$ (Guo et al., 2017). Due to the linear scalability of the FM discussed in section 2.1.2,

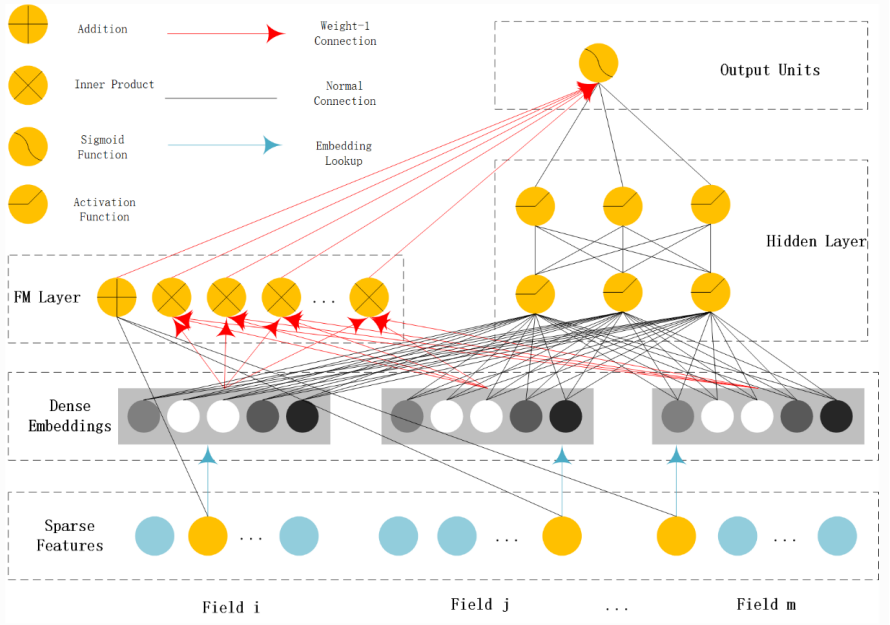

Figure 2.6.: DeepFM network architecture. Source: (Shen, 2017)

the FM component can effectively capture important order-2 feature interactions automatically, without the need to manually define cross-product functions as in the case of the W&D model.

Meanwhile, the *Deep component* fulfills a similar purpose as in the case of the W&D model. The deep component is a MLP network that takes the feature embedding vector $\tilde{\mathbf{x}}$ as inputs, and learns the higher-order feature interactions that cannot by captured by the 2-way FM component. Like with the W&D model, the Deep component acts as a residual network for capturing signals that we ommitted by the FM component. The DeepFM model can therefore be formulated as in equation 2.14.

$$f_{\Theta}^{DeepFM} = f_{\Phi}^{FM^2}(\tilde{\mathbf{x}}) + f_{\Omega}^{MLP}(\tilde{\mathbf{x}}) \tag{2.14}$$

A notable difference in the Deep components between the W&D and DeepFM models are in the construction of the embedding layers that preprocess the input to the MLP. In DeepFM, the latent feature vectors ($\mathbf{V}$) are trainable network weights that are derived during SGD optimization in the FM component. For every successive step in the model training, the learned latent feature vectors are used in the embedding layer that preprocesses that raw input features before the MLP of the Deep component (see Figure 2.7). This is similar to how the feature embeddings were derived in the FNN model (Zhang et al., 2016), except for fact the FM layer is included in the overall learning architecture of the model. This elimitates the need to pretrain the FM model, thereby allowing for the FM latent feature vectors to by learned concurrently during the overall DeepFM model training procedure (Guo et al., 2017).

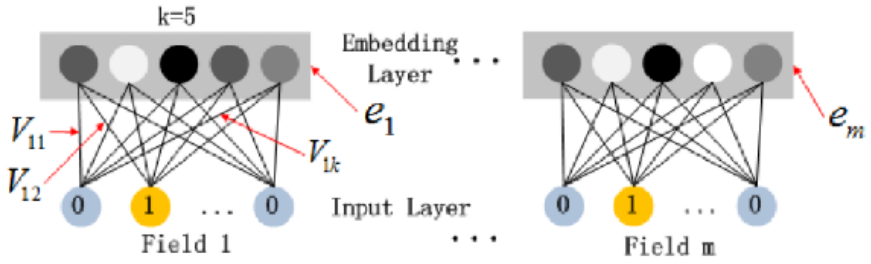

Figure 2.7.: The embedding layer of the Deep Component in the DeepFM model. Source: (Guo et al., 2017)

The key benefits of the DeepFM model are threefold. Firstly, we have already mentioned above that the fact that the FM model is incorporated directly into the model architecture eliminates the need to pretrain the FM latent feature vectors, thereby eliminating the computational overhead that is neccessary for this in the case of FNN (Zhang et al., 2016). Secondly, the Dual Tower architecture allows the model to simultaneously learn both low as well as high order feature interactions. Thirdly, the previously discussed computational efficiency of the FM model means that DeepFM is relatively scalable in terms of the number of features and the size of the latent embedding space, especially in comparison to the Product based Neural Network (PNN) models (Qu et al., 2016), which will be covered in section 2.1.6.

2.1.6. Feature Interaction Operator Models

MLPs have been proven to be universal function approximators, meaning that any function can be can be sufficiently approximated with a larger enough MLP (Cybenko, 1989; Hornik et al., 1989, 1990). However, Shalev-Shwartz et al. (2017) found that for complex problems where the true target function is actually a larger set of uncorrelated solution functions, Deep Neural Networks suffer from an *insensitive gradient issue* during gradient descent optimization. Shalev-Shwartz et al. (2017) show that when the target function is a set of uncorrelated functions, the variance of the gradient with respect to the target decreases linearly with respect to the number of functions that make up the target. The decrease in this variance has the effect of decreasing the correlation between the gratient and the target, causing the optimization of the DNN to fail. Qu et al. (2018) argues that since the target function for the CTR classification task typically consists of a set of uncorrilated if-then classifiers on the basis sparse categorical features, the insensitive gradient issue is likely to be prevalent in cases where MLPs are relied upon directly to detect the key feature interactions in sparse CTR classification data.

The above justifies the design of specific layers and architectures that explicitly detect important feature interactions in the data. In this section, we discuss **Feature Interaction Operators**, which are deep learning layers that were developed specifically to assist the DNN in its capacity to learn higher feature interactions (Zhang et al., 2021). The three different type of Feature Interaction Operators that are discussed in this section

are Product Operators, Convolutional Operators and Attention Operators.

Product Operators models

Product Operator networks are neural networks that include layers with inner or outer product operations in order to explicitly model feature interactions (Zhang et al., 2021). The **Product-bassed Neural Network** (PNN) introduced the concept of product operator models as it includes a product layer between the embedding layer and the MLP in order to model second order feature interactions in the data. All of this is arranged as a Single Tower architecture, as shown in Figure 2.8.

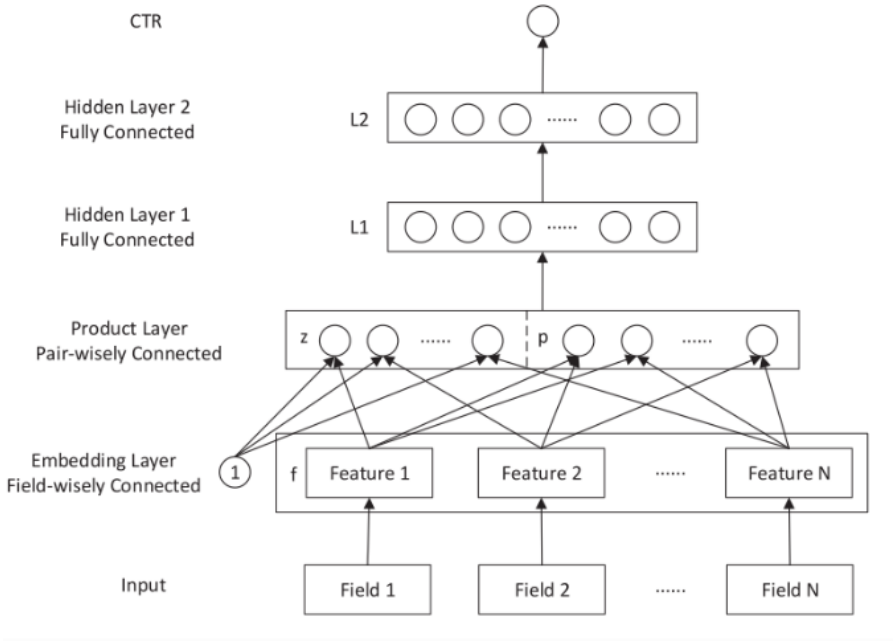

Figure 2.8.: Product-based Neural Network. Source: (Qu et al., 2016).

The key defining component of the PNN is the *Product Layer*. Each embedding field $\mathbf{e}_i$ in the preceding embedding layer is pair-wisely connected to each of the other fields and a "1" constant signal. The output of the product signal can then be broken out into two parts:

- Virtue to the constant "1" signal, the first part is simply a vector $\mathbf{z}$ that consists of a concatenation of all of the field embeddings. In other words $\mathbf{z} = \tilde{\mathbf{x}}$.
- A second-order interaction vector, $\mathbf{p} = \{p_{i,j}\}$ where $i, j = 1, \ldots, n$, where each element $p_{i,j} = g(\mathbf{e}_i, \mathbf{e}_j)$ defines a pairwise field interaction.

In their initial paper, Qu et al. (2016) proposed two variants of the PNN model, the Inner Product Based Neural Network and the Outer Product Based Neural Network,

differentiated by whether g is the Inner or Outer product operation respectively. The $\mathbf{z}$ and $\mathbf{p}$ vectors are then both projected to $\mathbb{R}^{D^1}$ space (the input dimension of the MLP network) by meant of trainable weight matrices, W_z and W_p. The input to the MLP network is then the sum of the two resulting vectors and a bias vector $\mathbf{b}_1$. The formulation for the PNN network is summarized in equation 2.15.

$$f_{\Theta}^{PNN}(\tilde{\mathbf{x}}) = f_{\Phi}^{MLP}(W_z\tilde{\mathbf{x}} + W_p\{g(\mathbf{e}_i, \mathbf{e}_j)\}_{i,j=1}^{n} + \mathbf{b}_1) \tag{2.15}$$

The inclusion of the product layer in the PNN model automatically incorporates second order field-wise interactions as inputs to the MLP by means of inner and outer products, thereby partially alleviating the previously mentioned insensitive gradient issue. Qu et al. (2016) found that as a result of this, the PNN models outperformed LR, FM, FNN and the CCPM models in terms of Log Loss and AUC. However, a major disadvantage of the product layer operations is the computational time complexity, which increases quadratically with the number of fields and the embedding dimension. In order to alleviate this, Qu et al. (2016) implement simplified versions of the inner and outer product computations (in which some neurons are eliminated in the inner product, and the result for the outer product is compressed for all fields at once), but even then the PNN models are still tend to be less computational than its peers. Furthermore, since the PNN model leverages a Single Tower architecture it suffers from the same issue as the FNN model wherein the lower order interactions are ignored.

Convolutional Operators

Convolutional Operator Models use the convolution operation to extract key local-global features from the categorical field embeddings. The earliest and most well known example of a Convolutional Operator model is the **Convolutional Click Prediction Model** (CCPM) developed by Liu et al. (2015). The overall architecture of this model was shown in the original paper as in Figure 2.9.

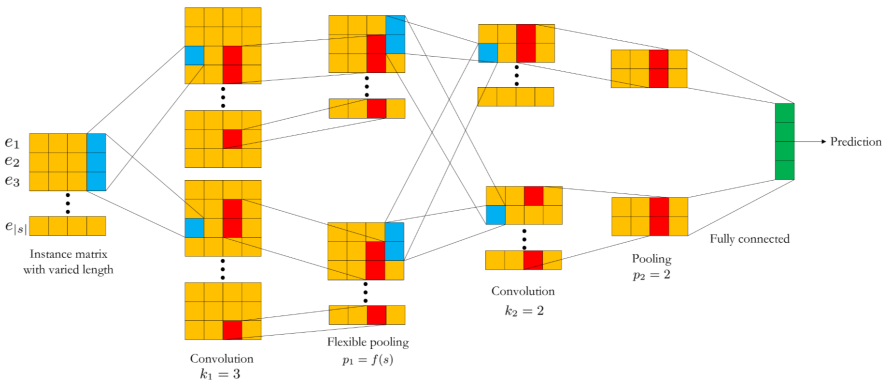

Figure 2.9.: Convolution Click Prediction Model architecture. Source: (Liu et al., 2015).

The input to the CCPM model consists of a $s \times D$ dimensional matrix of stacked

categorical feature empeddigs, as shown on the left-hand side of Figure 2.9. Per the standard architecture discussed in (Liu et al., 2015), this matrix is then passed through a series of Convolutional and Pooling layers. The number of maximum features that the intermediate pooling layers take is flexible to account for the flexible input matrix length. The final prediction is calculated by passing the final pooling result through a MLP, shown on the right-hand side of Figure 2.9 in green. This makes the CCPM model a Single Tower architecture model that can be roughly summarized as per equation 2.16.

$$f_{\Theta}^{CCPM}(\tilde{\mathbf{x}}) = f_{\Phi}^{MLP}(f_{\Omega}^{Conv}(\tilde{\mathbf{x}})) \tag{2.16}$$

where f_{Ω}^{Conv} represents the series of convolutions and pooling layers described above.

Liu et al. (2015) show that the CCPM model outperforms the LR, FM and RNN models in terms of Log Loss/Binary Cross-Entropy. However, a common critisizm of the CCPM model is that due to the equivariance property of the Convolution operations, the degree to which it is able to capture important feature interactions in the data is highly dependant on how the features are ordered in the input matrix (Gu, 2021; Qu et al., 2018; Zhang et al., 2021). Convolutions by nature extract feature maps in the local neighbourhood of each variable, but fail to do so globally. The **Feature Generation by Convolutional Neural Network** model proposed by Liu et al. (2019). The architecture for the FGCNN model is shown in Figure 2.10

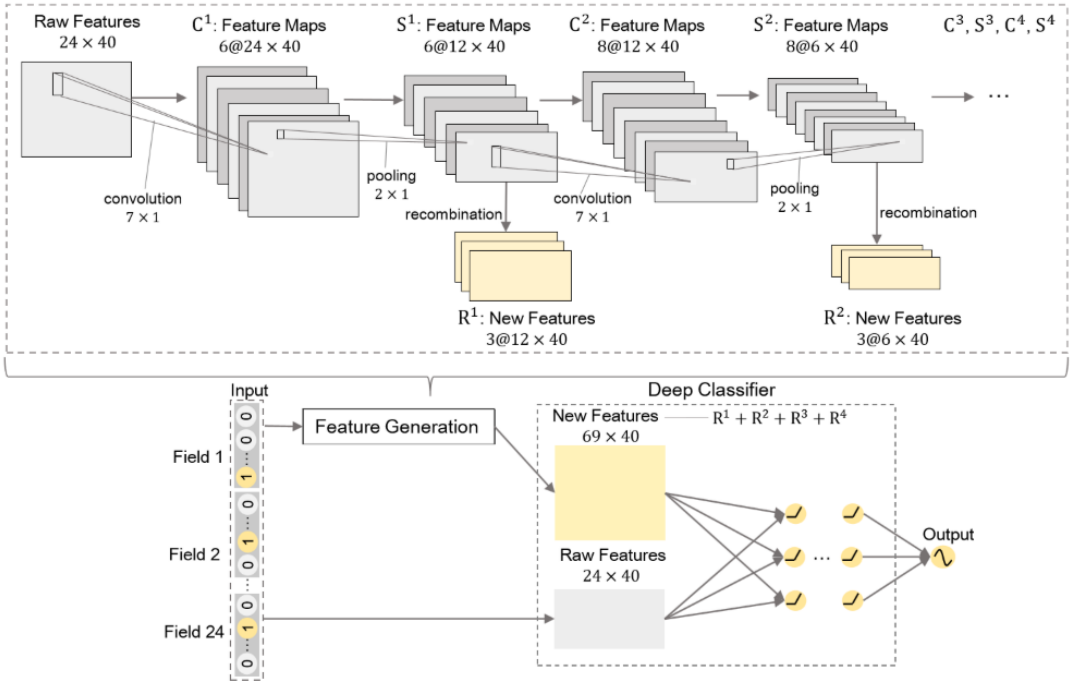

Figure 2.10.: FGCNN model architecture with Feature Generation Component. Source: (Liu et al., 2019)

The main body of the FGCNN model consists Deep Classifier that is essentially an Inner Product Neural Network, which was discussed above (Qu et al., 2018). The inputs to the Deep Classifier consist of the input feature embeddings ($\tilde{\mathbf{x}}$), concatenated with a set of new features that are created in the Feature Generation component of the model. This Feature Generation component represents the primary innovation of the FGCNN model, and is visualized at the top of Figure 2.10. As with CCPM, the Feature Generation component is composed of a searies of two dimensional convolutional and max pooling layers, and takes the input feature embedding matrix as an input. However, in order to solve for the input order dependancy issue prevailent with CCPM, the resulting feature maps are first passed through a fully connected *recombination layer* that models non-adjacent interactions.

Attention Operators

Attention Operator Models aim to utilize the attention mechanism for identifying the key feature interactions in the data. The **Autotomatic Feature Interaction Learning** (AutoInt) model proposed by Song et al. (2019) makes use of a multi-head self attention network to model the important feature interactions in the data. The initial paper separates the model into three parts: an embedding layer, an interaction layer and an output layer. The embedding layer aims to project each sparse multi-value categorical a and dense numerical feature into a lower dimensional space, as per the below:

$$\mathbf{e_i} = \frac{1}{q}\mathbf{V_i x_i}$$

where $\mathbf{V_i}$ is the embedding matrix for the i-th field, x_i is a multi-hot vector, and q is the number of non-zero values in x_i. The interaction layer employs the multi-head mechanism to determine which higher order feature interaction are meaningful in the data. This not only improves the efficiency of model traning, but it also improves the model's explainability. Lastly, the output layer is a fully connected layer that takes in the concatinated output of the interaction layer, and applies the sigmoid activation function to produce the final prediction. The architecture of the AutoInt model is shown in Figure2.11.

2.2. Deep Reinforcement Learning

The second part of the background chapter is dedicated to Deep Reinforcement Learning. We first proceed by explaining the fondational concepts, in which we will establish definitions for Markov Decision Processes, Reinforcement Learning and Dynamic Programming. We then move on to explain Q-Learning, a specific class of Reinforcement Learning algorithms, as well as how Deep Learning models are being applied in the case of Deep Q-Learning. Finally, we introduce the Deep Reinforcement Learning News reccomendation (DRN) algorithm (Zheng et al., 2018), a Q-learning algorithm which we have repurposed for ad recommendation.

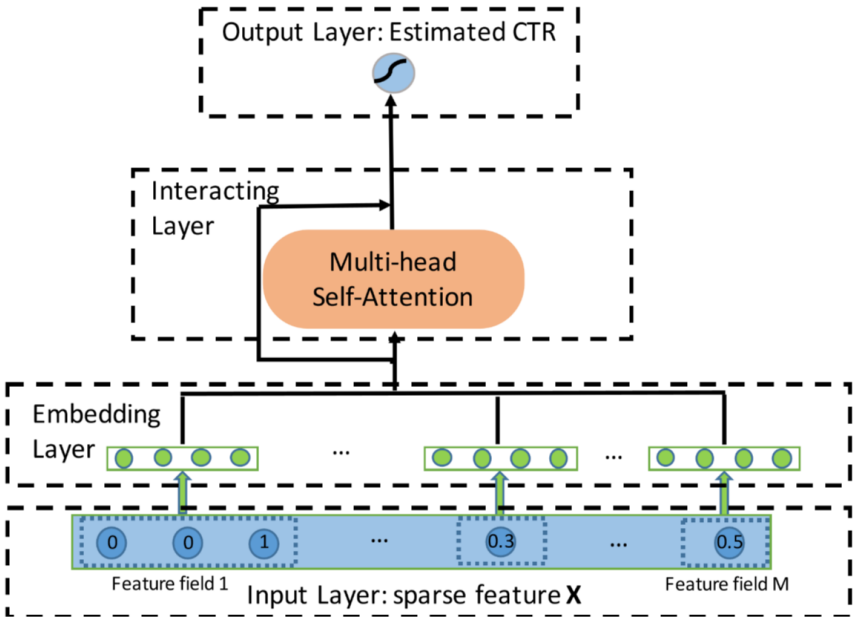

Figure 2.11.: The Automatic Feature Interaction Learning model architecture. Source: (Song et al., 2019)

2.2.1. Reinforcement Learning Basics

Markov Decision Process and Bellman Optimality Equations

In the case of many online systems and applications where there is a series of interactions between users and the system, it is often desirable to find the optimal set of content to display to the users in order to maximize their engagement as time goes on. This problem can be framed as a **Markov Decision Process**. Definition 2.2.1 was taken from (Pike-Burke, 2024b):

Definition 2.2.1. An episodic **Markov decision process** (MDP) is defined by tuple $\mathcal{M}(\mathcal{S}, \mathcal{A}, H, \nu, \{P_h\}_{h=1}^H, \{r_h\}_{h=1}^H)$ where:

- $\mathcal{S}$ is the state space of finite cardinality.
- $\mathcal{A}$ is the finite set of actions.
- $H \in \mathbb{N}$ is the horizon of the problem.
- ν is the initial state distribution.
- $\{P_h\}_{h=1}^H$ is the collection of transition functions where $P_h : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ where $\Delta(\mathcal{S})$ is the set of probability distributions over $\mathcal{S}$. When action $a \in \mathcal{A}$ is taken from state $s \in \mathcal{S}$ at stage h, $P_h(s'|s, a)$ gives the probability of transitioning to state $s' \in \mathcal{S}$ for all $s' \in \mathcal{S}$.

- $\{r_h\}_{h=1}^H$ is the collection of reward functions, $r_h : \mathcal{S} \times \mathcal{A} \to [0, 1]$ where $r_h(s, a)$ gives the reward from taking action $a \in \mathcal{A}$ from state $s \in \mathcal{S}$ at stage $h \in \{1, \ldots, H\}$.

To relate the above to the Ad marketplace context, we can adopt the language introduced in section 2.1.1. Namely, the state space $\mathcal{S}$ is comprised of the set of all available **User** and **Contextual** features in the ad marketplace data, the action space $\mathcal{A}$ is made up of the set of available advertisments with the associated **advertisment** features and the reward would be the binary click label for each instance.

The aim of **Reinforcement Learning** is to interact with the MDP process environment in such a way that allows the agent to learn the *optimal policy* - i.e. the state-stage $\to$ action mapping that maximizes the *value* over the longer term. We refine some of these terms with more definitions from (Pike-Burke, 2024b) below:

Definition 2.2.2. A **policy** $\pi = \{\pi_h\}_{h=1}^H$ is a sequence of mappings $\pi_h : \mathcal{S} \to \mathcal{A}$ for any $s \in \mathcal{S}, a \in \mathcal{A}$.

Definition 2.2.3. the **value** of a policy π from state $s \in \mathcal{S}$ in stage $h \in \{1, \ldots, H\}$ is given by:

$$V_h^\pi = \mathbb{E}\left[\sum_{l=h}^{H} \gamma^{l-h} r_l(s_l, a_l) \;\middle|\; s_h = s, a_l = \pi(s_l), s_{l+1} \sim P_l(\cdot|s_l, a_l)\right] \tag{2.17}$$

Where γ represents a chosen *discount factor* for potential rewards received in the future. The *optimal policy* π^* is then the one with the highest value, i.e.:

$$\pi_h^*(s) = \arg\max_{\pi \in \Pi} V_h^\pi(s)$$

for any $s \in \mathcal{S}$ and any $h \in \{1, \ldots, H\}$. In order to optimize for the policy that maximizes the expected value, we would need to consider the expected value of taking a specific action a at specific stage h and state s, for some given policy π. This is given by the **Q-function**, defined below in definition 2.2.4.

Definition 2.2.4. The **Q-function** associated with taking action $a \in \mathcal{A}$ from state $s \in \mathcal{S}$ at stage $h \in \{1, \ldots, H\}$ under some given policy π is given by

$$Q_h^\pi(s, a) = \mathbb{E}\left[\sum_{l=h}^{H} \gamma^{l-h} r_l(s_l, a_l) \;\middle|\; s_h = s, a_h = a, a_l = \pi(s_l), s_{l+1} \sim P_l(\cdot|s_l, a_l)\right] \tag{2.18}$$

The relationship between the value function V and the state action value function Q is summarized in the **Bellman equations**. For any policy π and for all h, s, a:

$$\begin{aligned} V_h^\pi(s) &= Q_h^\pi(s, \pi_h(s)) \\ Q_h^\pi(s, a) &= r_h(s, a) + \gamma \sum_{s' \in \mathcal{S}} P_h(s'|s, a) V_{h+1}^\pi(s') \\ V_{H+1}^\pi(s) &= 0 \end{aligned}$$

The above leads to the key equations that underpin Reinforcement Learning - the **Bellman Optimality equations**.

Proposition 2.2.5. If V^* satisfies the Bellman Optimality Equations, then for all h, a, s:

$$V_h^*(s) = \max_{a \in \mathcal{A}} Q_h^*(s,a)$$
$$Q_h^*(s,a) = r_h(s,a) + \gamma \sum_{s' \in \mathcal{S}} P_h(s'|s,a) V_{h+1}^*(s')$$
$$V_{H+1}^*(s) = 0$$

The above implies that the optimal policy π^* is given by:

$$\pi_h^*(x) = \arg\max_{a \in \mathcal{A}} Q_h^*(s,a)$$

In other words, the optimal policy is found by maximizing the Q-function at each stage and state.

Dynamic Programming, UCBRL and Thompson Sampling RL

When the transition probabilities $P_h(s'|s,a)$ and reward function $r_h(s,a)$ are known, it is then possible to find the optimal policy by first calculating the value of the Q function at stage of the episode (which will simply be $r_H(s,a)$), and then working back to find $Q_h^*(s,a)$ for each stage. This is known as the *Dynamic Programming* or the *Backward Recursion* algorithm. The steps involved are described in algorithm 1.

Algorithm 1 Dynammic Programming algorithm

Require: $P_h(s'|s,a)$ and $r_h(s,a)$ are known
1: Set $V_{H+1}^*(s) = 0$ for all $s \in \mathcal{S}$
2: **for** $h = H, \ldots, 1$ **do**
3: Calculate $Q_h^*(s,a) = r_h(s,a) + \gamma \sum_{s' \in \mathcal{S}} P_h(s'|s,a) V_{h+1}^*(s')$
4: Set $\pi_h^*(s) = \arg\max_{a \in \mathcal{A}} Q_h^*(s,a)$
5: Define $V_h^*(s) = \max_{a \in \mathcal{A}} Q_h^*(s,a) = Q_h^*(s, \pi_h^*(s))$
6: **end for**

Of course in practice, we more often than not do not know the transition probabilities and reward function of the environment. A possible method for proceeding in this case in by executing a Reinforcement Learning algorithm that chooses actions in such a way that simultaneously allows for the collection of sufficient datapoints for estimating the reward function and transition probabilities, while also minimizing the long term cumulative *regret* as much as possible.

Definition 2.2.6. Let K be the total number of episodes, π^* be the optimal policy and π_t be the policy chosen for episode t. The the cumulative regret over K episodes is given

by

$$\mathcal{R}_T = \sum_{t=1}^{K} \mathbb{E}\left[V_1^{\pi^*}(s_{1,t}) - V_1^{\pi_t}(s_{1,t})\right] \tag{2.19}$$

Two algorithms for doing this are the Upper Confidence Bound for Reinforcement Learning (UCBRL) (Auer et al., 2008) and the Thompson Sampling algorthm for Reinforcement Learning (Pike-Burke, 2024a). UCBRL works by calculating the empirical mean transition probabilities as:

$$\hat{P}_{h,t}(s'|s,a) = \frac{N_{h,t}(s,a,s')}{\max\left(N_{h,t}(s,a),1\right)}$$

Where $N_{h,t}(s,a,s')$ represents the number of times until episode t that taking action a from state s resulted in a transition to state s', and $N_{h,t}(s,a)$ is the number of times upt to episode t that action a was taken from state s. The result from Weissman et al. (2003) is then used to establish confidence bounds on the transition probabilities, which is then used to calculate the Q-value of each action-state pair to some minimum degree of confidence.

Meanwhile, the Thompson RL algorithm (Pike-Burke, 2024a) proceeds by taking a Bayesian approach and sampling a transition distribution from a Dirichelet distribution, which is the conjugate prior distribution for a categorical distribution. It then establishes a policy calculating the Q-function values with this transition distribution, and acts accordingly for the episode. Finally, it then revises the Dirichelet distribution by posterior update before the next episode.

Both of the methods above are sufficient in cases where the state and action spaces are finite and relatively small. However, a major drawback with both of these algorithms is that they both require the state transtition model to be approximated and stored. This poses a serious practical issue in the case of Ad personalization, since as we have covered in section2.1.1, Ad marketplace data tends to be extremely sparse once encoded. This means that in order to fully calculate the transition probabilities, one would need to account for a vast number to state-action-transition tuples, which is likely to be computationally infeasible. In the next section, we explore Q-learning, a subdomain of Reinforcement Learning that aims to learn the Q-function directly, and is therefore *model free*.

2.2.2. Q-Learning and Deep Q-Learning

Q-learning was pioneered by Watkins (1989) as a model-free, and therefore computationally efficient method for solving RL problems that have a sparse state and action space. The basic Q-learning algorithm works by maintaining an estimate of the Q-function for every state-action pair, and selecting a policy that ise greedy with respect to this estimate (Pike-Burke, 2024b). The Q-function estimate $\hat{Q}(s,a)$ is usually initialized with its value set to $\hat{Q}_h(s,a) = H$ for all $s \in \mathcal{S}, a \in \mathcal{A}$. In each stage, the algorithm proceeds by choosing the action that maximizes the $\hat{Q}_h(s,a)$, and observing the reward received $r_h(s,a)$ and the resulting state s'. Before the next stage, the $\hat{Q}_h(s,a)$ estimate

is then updated to reflect the actual results observed. The steps for the basic Q-learning algorithm are shown in Algorithm 2

Algorithm 2 Basic Q-Learning Algorithm. Source: (Pike-Burke, 2024b)

Initialization: $\hat{Q}_{h,0}(s,a) = H$ for all $s \in \mathcal{S}, a \in \mathcal{A}, h \in \{1,\dots,H\}$
for episode $t = 1,\dots,K$ **do**
  Observe $s_{1,t}$
  for Stage $h = 1,\dots,H$ **do**
    Select action $a_{h,t} = \arg\max_{a\in\mathcal{A}} \hat{Q}_{h,t}(s_{h,t},a)$ and update $N_{h,t}(s_{h,t},a_{h,t})$
    Observe $s_{h+1,t}$ and $r_{h,t}(s_{h,t},a_{h,t})$
    for All (s,a,s') values **do**
      if $(s,a,s') = (s_{h,t},a_{h,t},s_{h+1,t})$ **then**
        Update $\hat{Q}_{h,t+1}(s,a) = (1-\alpha_{h,t})\hat{Q}_{h,t}(s,a) + \alpha_{h,t}(r_{h,t}(s,a) + \gamma \arg\max_{a'\in\mathcal{A}} \hat{Q}_{h,t}(s',a'))$
      else
        Set $\hat{Q}_{h,t+1}(s,a) = \hat{Q}_{h,t}(s,a)$
      end if
    end for
  end for
end for

The advantage of the Q-learning algorithm is that the estimates for unobserved values remains the same, there is no additional computation that needs to take place. We only need to store $H \times |\mathcal{S}| \times |\mathcal{A}|$ unique estimates for $\hat{Q}(s,a)$, as opposed to $H \times |\mathcal{S}| \times |\mathcal{A}| \times |\mathcal{S}|$ transition probabilities, which significantly lowers the memory requirement (Pike-Burke, 2024b). Note that in Algorithm 2 the $\alpha_{h,t}$ parameter represents an *update step size* parameter, which usually depends inversely on $N_{h,t}(s_{h,t},a_{h,t})$.

While the basic Deep Q-Learning algorithm significantly decreases the computational requirement by removing the need to store the transition probability model, keeping track of Q-function values may still be prohibitive in the case where the state and action spaces are probibitively sparse, as is the case in the Ad personalization domain. All of the aforementioned algorithms are designed for discrete state and action spaces with relatively small cardinalities, and the performance of these algorithms deteriorates for increased number of state-action combinations that need to be accounted for (Pike-Burke, 2024b). For sparse enviroments, it is therefore desirable to find a suitable *function approximator* $f_\Theta : \mathbb{R}^n \to \mathbb{R}$ for the Q-function that can estimate the value of the of a state-action pair on the basis of a set of input action-state features $\mathbf{x} \in \mathbb{R}^n$ and a set of learned parameters Θ. This means that rather than having to store $\hat{Q}(s,a)$ values for every state-action combination, we will only have to store the function parameters Θ, and assume that $\hat{Q}(s,a) = f_\Theta(\mathbf{x})$, thereby changing the memory requirement from $H \times |\mathcal{S}| \times |\mathcal{A}|$ to simply $|\Theta|$.

Jin et al. (2020) provide a method for deriving the Q-function approximator f_Θ by means of linear approximation. Let ϕ represent a feature L2 normalization such that

$\forall \mathbf{x} \in \mathbf{X}, ||\phi(\mathbf{x})||_2 \leq 1$. Jin et al. (2020) showed that for any policy π, there exists a set of vectors w_h^π such that

$$Q_h^\pi(s,a) = \phi(\mathbf{x})^\intercal w_h^\pi \tag{2.20}$$

Equation 2.20 then essentially reduces the problem of finding the finding the optimal policy approximator $f_\Theta \approx Q_h^*(s,a)$ problem to one that can be solved using gradient descent with respective to the function parameters $\Theta = w_h$.

Although linear approximation may be sufficient for simple tasks, it is still however likely that in a lot of practical applications, assuming that the realtionship between the state-action features and the expected rewards is linear may be too simplistic. The linear Q-function approximator in the form of equation 2.20 may not be expressive enough for the problem at hand (Pike-Burke, 2024b). For these cases, Mnih et al. (2015) proposed the Deep Q-Learning algorithm in which which the Q-function is approximated using a Deep Neural Network called a *Deep Q-Network*. The full Deep Q-Learning algorithm is shown in Algorithm 3.

Algorithm 3 Deep Q-Learning with Experience Replay. Source: (Mnih et al., 2015)

Initialize: Replay memory $\mathbf{D}$ to capacity $\mathbf{N}$.
Initialize: Q-function approximator f_θ with random weights Θ.
Initialize: Set target action-value function $\hat{f}_{\hat{\Theta}}$ with $\hat{\Theta} = \Theta$
for Episode $t = 1, \ldots, K$ **do**
    Initialize state sequence s_1 and preprocess the sequence $\phi_1 = \phi(s_1)$
    for Stage $h = 1, \ldots, H$ **do**
        with probability ϵ select a random action a_h, otherwise select action $a_h = \arg\max_{a \in \mathcal{A}} f_\Theta(\phi_h, a)$
        Execute action a_h and observe reward $r_h(s_h, a_h)$ and the next state s_{h+1}
        Proprocess the features of the next state $\phi_{h+1} = \phi(s_{h+1})$
        Store the transition $(\phi_h, a_h, r_h, \phi_{h+1})$ in $\mathbf{D}$
        Sample a random set of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$
        Set $y_j = \begin{cases} r_j, & \text{if } j = H \\ r_j + \gamma \max_a \hat{f}_{\hat{\Theta}}(\phi_{j+1}, a), & \text{otherwise} \end{cases}$
        Perform gradient descent step on $L(\Theta) = (y_j - f_\Theta(\phi_j, a_j))^2$ with respect to Θ
        Every C steps reset the target action-value function $\hat{f} = f$ by setting weights $\hat{\Theta} = \Theta$
    end for
end for

In the original paper, the Deep Q-learning Network (DQN) algorithm was initially proposed to find the optimal playing policies for 49 of the classic Atari 2600 games, where a given state would be represented by the game's pixel values, the set of actions were possible joystick movements and button presses and the rewards were the number of points accumalated throughout the game. Mnih et al. (2015) due to the highly sequential nature of the state-action-reward data, using newly observed data to directly

update the Q-function DNN approximator would result in significant instabilities due to autocorrelation of inputs (Mnih et al., 2015). It is for this reason that the following modifications made in algorithm 3 over and above basic Q-Learning:

- In order to minimize the risk posed by the sequential dependancies in the newly observed data, Deep Q-Learing model is trained by means of *experience replay.* Every new observation is stored in memory **D**. In order to fit the model, a sample of *experience observations* $(\phi_j, a_j, r_j, \phi_{j+1})$ is sampled from **D** uniformly at random, and is then used to train the model by means of gradient descent.
- Using the same model for selection and calculating the gradient descent targets (y_j) tends to lead to further model instability as the y_j values are overestimated (Mnih et al., 2015). To prevent this, a separate *target model* $\hat{f}$ is used to calculate the y_j values.

2.2.3. DRN: Deep Reinforcement Learning for News Recommendation

DRN (Zheng et al., 2018) is a MDP framework that leverages a Deep Neural Network to approximate the expected total user response for each recommendation at each state. The two major advantages of DRN are firstly that it is composed on the basis of a continuous state and action representation, meaning that it can be scaled to large and sparse datasets, and secondly that the proposed reward function consists of both the immediate reward (user click) as well as the future expected reward (long term user engagement), thereby allowing for better recommendations over a user's lifetime.

3. Deep CTR model Evaluation

3.1. Models and Model selection Methodology

As explained above, I will explore a number of deep learning models. I selected five popular models on the basis of the following criteria

- Competitive predition accuracy in the KDD12, Criteo and Avazu datasets as published on Papers with Code.
- Ideally, I was looking for a representitive set of models for each model type as discussed in (Zhang et. al. 2021). Therefore I was looking for models that employed Product Interaction Opetators, Attention Operators and Factorization Machines as a basis.
- The code for the model has to be accessible and intuitive to use.

On the basis of the above critea, I have chosen the following models to explore:

- Factorization Supported Neural Networks
- Product Based Neural Networks
- Wide and Deep
- DeepFM
- Automatic Feature Interaction (AutoInt)

3.2. Experiment Setup

3.2.1. Datasets and Preprocessing

3.2.2. Evaluation Metrics

3.2.3. Hyperparameter Selection

3.3. Deep CTR Model Results

4. Deep Reinforcement Learning for Ad Personalization

4.1. DeepCTR-RL Framework

4.1.1. Model Framework

4.1.2. Feature types

4.1.3. Double Deep Q-Learning Network

4.1.4. Exploration

4.1.5. Experience Replay

4.2. Experiment Setup

4.2.1. Dataset and Preprocessing

4.2.2. Evaluation Metrics

4.2.3. Hyperparameter Selection

4.3. Deep CTR-RL Results

5. Discussion

Discussion goes here.

6. Conclusion

Conclusion goes here.

A. Appendix

A.1. Abbreviations and Acronyms

Term	Definition	Reference
LR	Logistic Regression	
FM	Factorization Machine	
FFM	Field-Aware Factorization Machine	
DNN	Deep Neural Network	
MLP	Multilayer Perceptron	2.1.3

A.2. Notation

Symbol	Definition	Reference
$\mathbf{x}$	Feature vector, before pre-processing	
n	the number of features in $\mathbf{x}$	
x_i	The i-th feature in $\mathbf{x}$	
$\mathbf{x_i}^{OH}$	One-hot encoded vector representation of categorical feature i	
$\mathbf{e}_i$	Embedded vector representation of categorical feature i	
z_i	Mean and variance standardized value for feature i from $\mathbf{x}$	
$\tilde{\mathbf{x}}$	$\mathbf{x}$ after catigorical embedding and numerical standardization.	
f	Pre-sigmoid classification function	
Θ	Parameter vector for f	

Bibliography

Yi Wang Aden. Kdd cup 2012, track 2, 2012. URL https://kaggle.com/competitions/kddcup2012-track2.

Peter Auer, Thomas Jaksch, and Ronald Ortner. Near-optimal regret bounds for reinforcement learning. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2008.

Yin-Wen Chang, Cho-Jui Hsieh, Kai-Wei Chang, Michael Ringgaard, and Chih-Jen Lin. Training and testing low-degree polynomial data mappings via linear svm. *Journal of Machine Learning Research*, 11(4), 2010.

Minmin Chen, Alex Beutel, Paul Covington, Sagar Jain, Francois Belletti, and Ed H. Chi. Top-k off-policy correction for a reinforce recommender system. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, page 456–464, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359405. doi: 10.1145/3289600.3290999. URL https://doi.org/10.1145/3289600.3290999.

Chen Cheng, Fen Xia, Tong Zhang, Irwin King, and Michael R. Lyu. Gradient boosting factorization machines. In *Proceedings of the 8th ACM Conference on Recommender systems*, page 265–272, 2014.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, page 7–10, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347952. doi: 10.1145/2988450.2988454. URL https://doi.org/10.1145/2988450.2988454.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals, and systems*, 2(4):303–314, 1989. doi: 10.1007/BF02551274.

eMarketer. Digital advertising spending worldwide from 2021 to 2027 (in billion u.s. dollars). Technical report, Statista Inc., 2023. URL https://www-statista-com.iclibezp1.cc.ic.ac.uk/statistics/237974/online-advertising-spe

Liqiong Gu. Ad click-through rate prediction: A survey. In Christian S. Jensen, Ee-Peng Lim, De-Nian Yang, Chia-Hui Chang, Jianliang Xu, Wen-Chih Peng, Jen-Wei Huang, and Chih-Ya Shen, editors, *Database Systems for Advanced Applications. DASFAA*

2021 International Workshops, pages 140–153, Cham, 2021. Springer International Publishing. ISBN 978-3-030-73216-5.

Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: A factorization-machine based neural network for ctr prediction. *CoRR*, abs/1703.04247, 2017. URL `http://arxiv.org/abs/1703.04247`. 1703.04247.

John T. Hancock and Taghi M. Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 7(1):28, 2020. doi: 10.1186/s40537-020-00305-w. URL `https://doi.org/10.1186/s40537-020-00305-w`. ID: Hancock2020.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, page 770–778, 2016.

Xiangnan He and Tat-Seng Chua. Neural factorization machines for sparse predictive analytics, -08-16 2017.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. doi: 10.1016/0893-6080(89)90020-8. URL `https://www.sciencedirect.com/science/article/pii/0893608089900208`. ID: 271125.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5):551–560, 1990. doi: 10.1016/0893-6080(90)90005-6.

Chi Jin, Zhuoran Yang, Zhaoran Wang, and Michael I. Jordan. Provably efficient reinforcement learning with linear function approximation. In Jacob Abernethy and Shivani Agarwal, editors, *Proceedings of Thirty Third Conference on Learning Theory*, volume 125 of *Proceedings of Machine Learning Research*, page 2137–2143. PMLR, 09–12 Jul 2020. URL `https://proceedings.mlr.press/v125/jin20a.html`.

Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. Field-aware factorization machines for ctr prediction. In *10th ACM Conference on Recommender Systems*, RecSys '16, page 43–50, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340359. doi: 10.1145/2959100.2959134. URL `https://doi.org/10.1145/2959100.2959134`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Liu, Yu, Wu, and Wang. A convolutional click prediction model, -10-17 2015.

Bin Liu, Ruiming Tang, Yingzhi Chen, Jinkai Yu, Huifeng Guo, and Yuzhou Zhang. Feature generation by convolutional neural network for click-through rate prediction. In *The World Wide Web Conference*, page 1119–1129, 2019.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 26 2015. doi: 10.1038/nature14236. LR: 20220408; JID: 0410462; CIN: Nature. 2015 Feb 26;518(7540):486-7. doi: 10.1038/518486a. PMID: 25719660; 2014/07/10 00:00 [received]; 2015/01/16 00:00 [accepted]; 2015/02/27 06:00 [entrez]; 2015/02/27 06:00 [pubmed]; 2015/04/16 06:00 [medline]; AID: nature14236 [pii]; ppublish.

Ciara Pike-Burke. Optimism/thompson sampling, 2024a.

Ciara Pike-Burke. Learning agents mlds course, 2024b.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Yanru Qu, Han Chai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. Product-based neural networks for user response prediction. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1149–1154. IEEE, 2016. ISBN 2374-8486. doi: 10.1109/ICDM.2016.0151. ID: 1.

Yanru Qu, Bohui Fang, Weinan Zhang, Ruiming Tang, Minzhe Niu, Huifeng Guo, Yong Yu, and Xiuqiang He. Product-based neural networks for user response prediction over multi-field categorical data. *ACM Trans.Inf.Syst.*, 37(1), oct 2018. doi: 10.1145/3233770. URL `https://doi.org/10.1145/3233770`.

Steffen Rendle. Factorization machines. In *2010 IEEE International Conference on Data Mining*, pages 995–1000, 2010. ISBN 1550-4786. doi: 10.1109/ICDM.2010.127. ID: 1.

Matthew Richardson, Ewa Dominowska, and Robert Ragno. Predicting clicks: Estimating the click-through rate for new ads. In *International Conference on World Wide Web*, WWW '07, page 521–530, New York, NY, USA, 2007. Association for Computing Machinery. doi: 10.1145/1242572.1242643. URL `https://doi.org/10.1145/1242572.1242643`.

Shai Shalev-Shwartz, Ohad Shamir, and Shaked Shammah. Failures of gradient-based deep learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, page 3067–3075. PMLR, aug 2017. URL `https://proceedings.mlr.press/v70/shalev-shwartz17a.html`.

Weichen Shen. Deepctr: Easy-to-use,modular and extendible package of deep-learning based ctr models, 2017. URL `https://github.com/shenweichen/deepctr`.

Song, Shi, Xiao, Duan, Xu, Zhang, and Tang. Autoint, -11-03 2019.

Jean-Baptiste Tien, joycenv, and Olivier Chapelle. Display advertising challenge, 2014. URL `https://kaggle.com/competitions/criteo-display-ad-challenge`.

Steve Wang and Will Cukierski. Click-through rate prediction, 2014. URL `https://kaggle.com/competitions/avazu-ctr-prediction`.

Xu Wang, Sen Wang, Xingxing Liang, Dawei Zhao, Jincai Huang, Xin Xu, Bin Dai, and Qiguang Miao. Deep reinforcement learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 35(4):5064–5078, 2024. doi: 10.1109/TNNLS.2022.3207346.

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, 1989.

Kevin Webster. Week 2: Multilayer perceptron, 2024.

Tsachy Weissman, Erik Ordentlich, Gadiel Seroussi, Sergio Verdu, and Marcelo J. Weinberger. Inequalities for the l1 deviation of the empirical distribution. *Hewlett-Packard Labs, Tech.Rep*, page 125, 2003.

Jun Xiao, Hao Ye, Xiangnan He, Hanwang Zhang, Fei Wu, and Tat-Seng Chua. Attentional factorization machines: Learning the weight of feature interactions via attention networks *, 2017. URL `https://arxiv.org/abs/1708.04617`.

Pengtao Zhang and Junlin Zhang. Memonet: Memorizing all cross features' representations efficiently via multi-hash codebook network for ctr prediction, -10-21 2023.

Weinan Zhang, Tianming Du, and Jun Wang. Deep learning over multi-field categorical data: A case study on user response prediction, 2016. URL `https://arxiv.org/abs/1601.02376`. 1601.02376.

Weinan Zhang, Jiarui Qin, Wei Guo, Ruiming Tang, and Xiuqiang He. Deep learning for click-through rate estimation, 21 Apr 2021. URL `https://arxiv.org/abs/2104.10584`.

Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. Drn: A deep reinforcement learning framework for news recommendation. In *2018 World Wide Web Conference*, pages 167–176, Lyon, France, 2018. International World Wide Web Conferences Steering Committee. doi: 10.1145/3178876.3185994. URL `https://doi.org/10.1145/3178876.3185994`.