Paper Summary:

Wide & Deep Learning—Tensorflow

* The paper focuses on training linear models together with a (feed-forward) neural net. It offers better recommendations (think predictions) without compromising speed too much.

Introduction

* Finding the important features to use for linear regression alone can be **cumbersome**
* **Deep neural networks can generalize unseen feature combinations** through low-dimensions, but alone, can over-generalize
* Wide & Deep learning: jointly trained wide linear models and deep neural networks combine memorization with generalization
* The focus of the paper is on recommender systems—when you search or install an app from the App Store, you get recommended apps based on your query
* **Memorization**: learning *the correlation of items or features and using this to inform the present based on historical data*. Recommendations are topical
* **Generalization**: explores new feature combinations that have never or rarely occurred in the past. Recommendations are more diverse
* Currently, and in the past, logistic regression models are used because they are **simple, scalable, and easy to understand.** They are trained based on some **binary classification** (0, 1 for example, did the user install this app? Yes/no 🡪 1/0)
* **Memorization** is done by using the **AND** operation on two objects. (did the user do ABC and XYZ?) for which this results in true (1). ABC and XYZ need not be apps, but can be categorical as well. **This type of feature training does not do well for searches that have not appeared in the training data before.**
* Deep NN (an embedding based model) can learn a low-dimensional dense embedding vector for each query and item feature with less feature engineering, **but this is difficult if the underlying query-item matrix is sparse and high rank.**
  + In this case, the NN will non-zero predictions for all pairs for dense embeddings (cf. 1), and make irrelevant app recommendations.
  + **Linear models can “learn” these exceptions with much fewer parameters, so why not do both?**

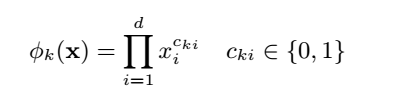
Overview

* **Training data: User data of their clicks and purchases**
* Upon receiving a query, the system retrieves the best match ranks them based on P(y|**x**), the **probability a user action label y (meaning the user will do some action y) given the features x.**
* The features include user features (country, language etc), context features (device, time of day, etc) and impression features (app age, and historical stats of app)

Wide & Deep Learning

-Wide Component

* This is the linear model of form: y = **w**T**x** + b
* y is the prediction and x is a vector of features and w is the model weights (parameters), b is the bias. **x =** [x1, x2, ... ,xn] **w=** [w1, w2, ... ,wn]
* The **feature set includes raw inputs and transformed features**
* A very important **transformed feature** is the **cross-product transformation**, defined as:



* *cki* is a Boolean. It is 1 if the *i*-th feature is part of the *k*-th transformation, φ*k* and zero otherwise
* Ok, what does this really mean???

Lets pretend you have a matrix of features (and these features are binary) as follows

Then, feature *cki* is 1***iff*** the underlying feature *k* is true AND underlying feature *i* is true and zero otherwise. This captures interactions between the binary features and adds nonlinearity to the generalized linear model. Cool stuff!

-The Deep Component—a feed-forward neural network

* The **original inputs are the features** (ex. feature string such as language=’en’). These sparse high dimensional features **are transformed into a low-dimensional** and dense real-valued vector called an **embedding vector** of dimensionality *O*(10) to *O*(100).
* The embedding vectors are initialized randomly and the values are trained to minimize a loss function (ex. Using stochastic gradient descent).
* **Afterwards,** the low-dimensional embedding feed into hidden layers of a neural network in a forward pass.
* **Each hidden layer does the following:**



Where *l* is the layer number and *f* is the activation function usually rectified linear units (ReLUs) (<https://en.wikipedia.org/wiki/Rectifier_(neural_networks)>

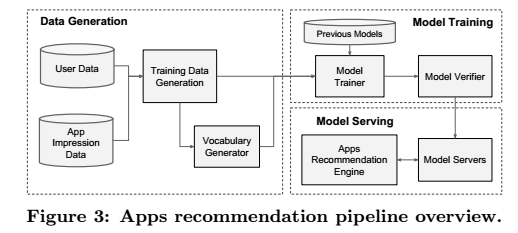
with functional form, **f(x) = max(0,x). So for x ≤ 0, f(x) = 0, and x>0, f(x) = x**

-It could also be another activation function, like sigmoidal

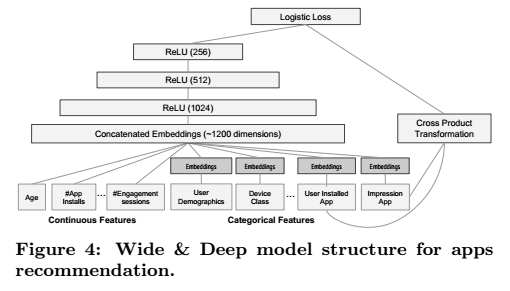
*a(l) ,b(l) ,W(l)* are the activation, bias, and model weights at the *l*-th layer respectively.

-Joint Training of Wide & Deep Model

* The wide component and deep component are combined using a **weighted sum** of their **output** **log odds** as the **prediction**, which is then fed to **one common logistic loss function** for the ***joint training.***



* The paper points out a distinction between *joint training*  and *ensemble*
  + *Ensemble*: individual models (ex. the linear regression and the neural net) are trained separately **without knowing each other** and **predictions only combined at inference time** (the time at which the prediction is needed) but **not** at training time
  + *Joint training:* The individual models are trained *together*. **All the parameters are optimized simultaneously** by taking **both** the wide and deep part (i.e. both models) **as well as the weights** of their **sum** into account at **training** time.
  + *Implications on model size:* Disjoint training of the models (*ensemble)* implies that each individual model must be larger to achieve reasonable accuracy. *Joint training* the wide model only needs to complement the weaknesses of the deep model with **cross-product feature transformations (recall the equation and matrix above)**. Also, see figure below.

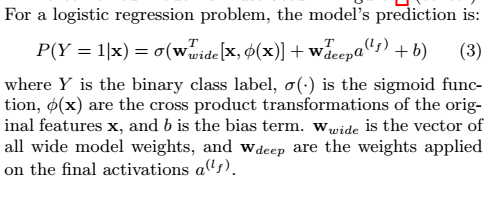


* Training of the Wide & Deep model is done by backpropagation the gradients from the **output** to **both** the wide and deep part **simultaneously using mini-batch optimization**. The authors of the paper used **Follow-the regularized-leader (FTRL) algorithm** with ***L*1 regularization as the optimizer for the wide part**, and **AdaGrad for the deep part**
  + FTRL <https://github.com/fmfn/FTRLp>
    - <https://www.quora.com/What-is-an-intuitive-explanation-of-Follow-the-Regularized-Leader-FTRL-algorithm>
    - <https://github.com/kastnerkyle/kaggle-criteo/blob/master/clf.py>
    - FTRL in tensorflow: <https://www.tensorflow.org/versions/r1.1/api_docs/python/tf/train/FtrlOptimizer>
  + AdaGrad: <http://sebastianruder.com/optimizing-gradient-descent/index.html#adagrad>

“One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that.

Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. The following algorithms aim to resolve this flaw.” –See link above for the other algorithms

* + Find optimizers including AdaGrad here: <https://www.tensorflow.org/api_guides/python/train>



System Implementation:

Cf. fig 3. The implementation consists of three stages: data generation, model training, and model serving

-Data Generation

**There are two kinds of features: continuous features, and categorical features**

**Categorical Features**

* They defined a period of time, t for the training data. They **make a label** called *app acquisition* and set the value to **1 if the impressed app was installed and 0 otherwise**.

**Continuous Features**

* Vocabularies: tables mapping feature strings to integers ID are also generated (i.e take a string like “sports” and turn this into a number, which is essentially the index in a big table).
  + The system computes the ID space for feature strings that occur more than a minimum number of times.
  + Continuous real valued functions are normalized to [0, 1] by mapping a feature *x* to a cumulative distribution function P(X ≤ *x*), divided into *nq* quintiles.
  + The normalized value is for values in the *i*-th quintiles. Quantile boundaries are computed during data generation.

-Model Training

The model structure is shown in figure 4 above.

* During training, the input layer takes in training data and vocabularies (the categorical and continuous features) to **generate sparse and dense features together with a label**
* **The wide component consists of the cross-product transformation of user installed apps and impression apps.**
* **The deep component uses a 32-dimensional embedding vector learned for each *categorical feature***
* **Concatenate** all the embeddings toeghet with the **dense features** resulting in a **dense** **vector** of 1200 dimensions (concatenated embeddings layer in fig 4)
* **Pass the concatenated vector into 3 ReLU layers**  and finally the logistic loss unit
* The Wide & Deep models are trained on over 500 billion data points (wow!)
  + Every time a new training set arrives the model needs to be re-trained. Since this is computationaly expensive, they implemented a **“warm-starting system”** which **initializes the new model with the embeddings and linear model weights from the previous model.**
  + They dry-run the model first to make sure it doesn’t give issues when it is online.

**-**Model Serving (testing)

* Once model is trained and verified, they load it into model servers. The servers receive a set of app candidates for each request and rank the apps from highest to lowest and show the apps to the user. **The scores are calculated by running a forward inference pass over the Wide & Deep model**
* They increase the speed (~10 ms) by using multithreaded parallelism by running smaller batches in parallel instead of scoring all candidate apps in a single batch inference step.

Experiment Results

* Compared app acquisitions and serving performance to a selection of users and found a statistically significant number of users downloading the recommended apps

Related Work

* Previous work focused on adding generalization to linear models🡪 Factorization machines
* This paper expands on that by learning nonlinear interactions between embeddings via neural networks instead of dot products