Click-through rate prediction using an ensemble of neural networks

Julian de Wit

Location : Hoek van Holland
Email : julian@dwssystemen.nl
Competition : Criteo Labs display advertising challenge
Date : 30 september 2014

1. Summary

This document describes the 4th prize solution to the Criteo Labs display advertising challenge hosted by Kaggle.com. The solution consisted of two steps. First the data was preprocessed. Rare and unseen test-set categorical values were all encoded as one category. The remaining features were one-hot encoded or hashed so that we ended up with roughly 200K separate sparse features. The second step was to use neural networks to train a number of different models with variations coming from different network architectures, bagging and preprocessing parameters. The different models were averaged into one final solution.

2. Features selection and extraction

At the start of the competition it quickly turned out that linear models using logistic regression would be very effective [1][2]. However, to improve the predictions one had to manually, or semi-automatically, do feature engineering to find interesting tranformations, interactions etc. Our goal was to try to use a deep neural network which is potentially a more 'powerful' model that could learn many interesting features automatically. This approach was reasonably succesful. However, some feature engineering was still necessary.

First we had to overcome technical problems. There were potentially more than 40M separate feature values and the first hidden layer of the network contained 128-512 units. This would result in a ~5 gigafloat weight matrix which would not fit on the GPU. Also, learning times would be infeasible. We chose to do feature reduction by encoding all rare feature values with one code. This also helps against overfitting. After this some categories still had 100K+ distinct values. It was decided to hash these features to a smaller space [3] to guarantee memory usage boundaries. Features with less than 10K distinct values were One-Hot encoded because we hoped that would give better results than hashing. In the end, the traindata contained roughly 128K-200K features.

Second, a solution had to be found for values in the test-test that were not in the train-set. Since the trained model had never encountered these values before, it could not have a good estimation of the weights for these features. We recoded the unseen test values to 'missing' or 'rare'. Basically this came down to giving an average weight. We do think however that more sophisticated approaches in estimating the unseen values, and especially unseen combinations of values, would give even better results.

Finally, the numeric features were standardized. For variation longtail features were log-transformed for some models. Both operations were not strictly necessary but standardizing helped greatly with convergence speed and the log-transformations helped for variation in the ensemble.

3. Modeling techniques and training

The initial idea was to build a 'drednet' [4] but it turned out that 2 hidden layers with respectively 128 and 256 units already gave a good performance. Increasing the number of units did not significantly increase the score. Dropout did also give no noticable improvements while it increased the training time. We kept dropout at the first hidden layer because it did seem to help when ensembling due to increased random variations in the model. We applied an approximated L2 regularization in the input layer. This helped somewhat against overfitting. The last layer was a softmax layer. We did not have time to implement a logistic to see if this worked better. In theory, it should give the same results.

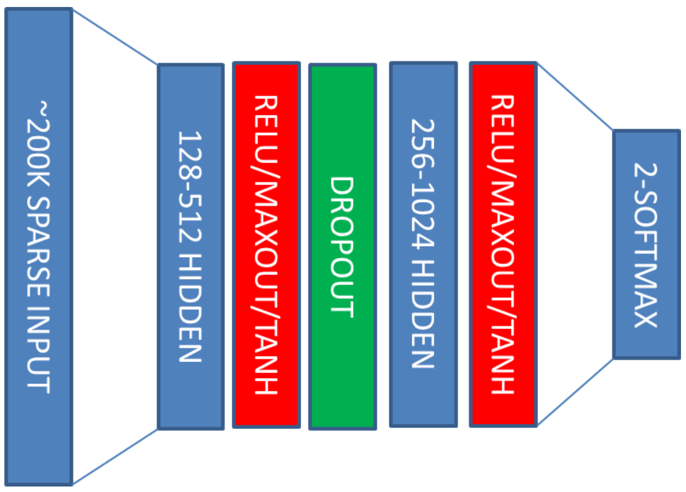

Figure 1. Network architecture(s)

On average a good model scored 0.451 on the 7th day holdout validation set. This translated to around 0.453 on the leaderboard. This was not too impressive since on the forum better single model scores were reported. However, the different trained models did very wel when we averaged them into an ensemble. The models were not weighted. We did some attempts in that direction but we quickly abandoned this since it gave no direct improvements. Other challenges also reported simply averaged ensembles with neural networks are quite good already [5].

4. Additional comments and observations

As reported earlier, our best single models did around 0.453 on the leaderboard. Althoug this is quite good especially since we did not do a lot of feature engineering, we do think that the neural network had a "blind spot". It is our guess that the reason for the was that the net could not give proper estimations for unseen, new feature values in the testset. Even more important would be interactions with new feature values. A good indication in this direction was that when we used the 1st day as holdout, the validation score was below 0.435. An explanation is that all the feature values of day 1 also occur in day 6-7 so there are almost no unseen values.

We did not have the time but would like to try out approaches that might help estimating new value combinations. We were thinking about matrix factorization[6], or to stay with neural networks, autoencoders and restricted boltzmann machines[7].

5. Code description

The software for this solution is a "home-grown" GPU based neural network. The code is provided in the repository[8]. Notice that this was a work-in-progress project and many things still have to be re-engineered and bugs might be present. The solution is written in C# using Visual Studio 2012. Gpu programming was done using the Cudafy.NET library.

The project consists of 2 parts. First there is the 'Sharpnet' neural network library. This is a generic neural network but all code that was not necessary for this challenge has been stripped out. The second 'Criteo' project contains the specialized software for this challenge. Below, a description is given for the files that are in this project.

There are classes that contain the datastuctures (*RawRecord* and *OneHotRecord*, *OneHotRecordReadOnly*). This is where the train and test data are loaded into in the various phases of the program.

The second category of classes perform the various steps in preprocessing. (*ProprocessingRawValues*, *PreprocessingOneHot*, *PreprocessingScale*. Every step can be run separately and results in a new train and test file.

Then there is the dataprovider (*OneHotRecordProvider*) and the trainer (*CriteoTrainer*) that are used respectively to feed data to the network and to orchestrate the training process.

Finally all classes come together in the main program. (*Program.cs*) Here the complete solution is done in a step-by-step manner. The end result is 2 single model submissions and one mini-ensemble submission.

6. Compiling and Running the solution

To compile the project it is first necessary to install the NVIDIA Cuda SDK (5.x). It is advised to use Visual Studio 2012 although it might be possible to use other versions. While not completely necessary it is very helpful to install the Cudafy.NET library. With the cudafy tools it is easier to check if everything is in working order. If the Cuda(fy) settings are done, compiling the project should be straightforward.

To run the software a 64bit windows OS is necessary. The project was done on a workstation with 32Gb memory. Perhaps it will still run with less memory but most likely 16Gb is not enough. This is due to the fact that training is done with the complete dataset in memory for high throughput. It is possible to let the network run in a 'streaming' fashing but this requires some recoding. The last requirement is a recent GPU board with Cuda compute capability 3.

7. References

1. http://people.csail.mit.edu/romer/papers/TISTRespPredAds.pdf
2. http://fastml.com/vowpal-wabbit-eats-big-data-from-the-criteo-competition-for-breakfast/
3. http://en.wikipedia.org/wiki/Feature_hashing
4. https://plus.google.com/+YannLeCunPhD/posts/UVT2fYTfoAC
5. http://blog.kaggle.com/2012/11/01/deep-learning-how-i-did-it-merck-1st-place-interview/
6. http://www.libfm.org/
7. https://www.cs.toronto.edu/~amnih/papers/rbmcf.pdf
8. https://github.com/juliandewit/kaggle_criteo