Factorization Machines with Tensorflow

Fri 10 February 2017

Update 2020-01-18: The APIs used in the examples are deprecated. A port to TensorFlow2 of this code can be found at https://github.com/gmodena/tensor-fm

I wanted to learn more about Factorization Machines and get a bit familiar with Tensorflow. This article gives an example of how to prototype the former in the latter.

Introduction

The idea of Factorization Machines (FMs from now on) is to learn a polynomial kernel by representing high-order terms as a low-dimensional inner product of latent factor vectors.

§The method gained notoriety in the early 2010s, when it was the winning solution in a few data mining competitions (KDD, Kaggle). Nowadays it is considered a solid framework for modeling highly sparse data. Similar features will end up being close together in an inner product space embedding, making it possible to model infrequent interactions in the training data. This makes the model very much suitable for tasks like click prediction and recommendation. FMs also rather versatile, and can mimic other factorization models by feature engineering.

The problem

I won't be original here. How would you go about recommending movies to a user? The canonical approach is to suggest new movies (or any type of item), based on movies the user liked (rated) in the past. We might want to take into account information such as user's gender, occupation and nationality. But also things like when a movie was rated, from which device, country etc. The first step towards building such a system could be trying to estimate how a user would rate a candidate recommendation. That is, we would like to solve a regression problem. Usually we'd represent training data as feature vectors, with indicator variables (one-hot-encoding, hashing trick) to denote categorical data. This representation will typically result in a high dimensional, highly sparse, feature space.

Polynomial regression

To justify FMs, it is useful to make a step back and think at interactions between features in terms of linear and polynomial regression.

In regression we want to model data represented by a design matrix $X \in R^{n \times p}$ of n observations with p features. We denote with $\mathbf{x}_i \in R^p$ the i-th feature vector (the history of a user's preferences), $y_i \in \mathbb{R}$ is its corresponding target (the movie rating). Moving forward I'll abuse notation and drop the i subscript from the feature vector and target.

We can learn the linear contribution of each feature as:

$\hat{y}(\mathbf{x}) = w_0 + \sum_{i=1}^{p} w_i x_i$

Where $w_0 \in \mathbb{R}$, $\mathbf{w} \in \mathbb{R}^p$ - the bias and weights for $\mathbf{x}$ - are parameters we'll learn from data.

The contribution of *job = data scientist* and *city = Amsterdam* is captured by learning the weights for $w_{ds}x_{ds} + w_{amsterdam}\,x_{amsterdam}$.

This model is efficient, and can be computed and stored with $O(p)$ complexity. The caveat is that the contribution of each feature is weighted individually. What if some *data scientists in Amsterdam* are more interested in certain movies than another demographic? To capture the interaction of *data scientist* and *Amsterdam* appearing together, we need to learn $w_{ds}x_{ds} + w_{amsterdam}\,x_{amsterdam} + w_{ds,amsterdam}\,x_{ds}x_{amsterdam}$

Fitting an order 2 polynomial will do the trick:
$\hat{y}(\mathbf{x}) = w_0 + \sum_{i=1}^{p} w_i x_i + \sum_{i=1}^{p}\sum_{j=i+1}^{p} x_i x_j w_{ij}$

However this time we have to learn additional $W \in \mathbb{R}^{p \times p}$ parameters to model pairwise interactions.

This model is more powerful, but comes with $O(p^2)$ complexity. If dealing with sparse data, we might not be able to learn W reliably. When doing machine learning at scale (millions of users, represented by hundred thousands of features, that rate thousands of items) we would typically be constrained to pick a less expressive model that guarantees faster runtime and lower memory footprint.

Factorization Machines

The "trick" of FMs is to model W as a lower dimensional factor matrix V, and do some algebra manipulation to fit the polynomial in linear time.

A bit more formally

This section follows from Rendle, 2010 and Rendle, 2012.

We can define an order two FM, describing two-ways interactions between features, as follows:

$\hat{y}(\mathbf{x}) = w_0 + \sum_{i=1}^{p} w_i x_i + \sum_{i=1}^{p}\sum_{j=i+1}^{p} x_i x_j \sum_{f=1}^{k} v_{if} v_{jf}$ (eq 1)

Where, like before, $w_0 \in \mathbb{R}$, $\mathbf{w} \in \mathbb{R}^p$ are the weights, $V \in \mathbb{R}^{p \times k}$ is a rank k factorisation of W.

The first part of (eq 1.) models linear interactions; the nested sum captures pairwise interactions. The effect of pairwise interactions w_{ij} is modelled as the dot product $w_{ij} \approx \langle \mathbf{v}_i, \mathbf{v}_j \rangle = \sum_{f=1}^{k} v_{if} v_{jf}$.

A key insight in Rendle's paper is that we can rewrite the interactions as:
$\sum_{i=1}^{p}\sum_{j=i+1}^{p} x_i x_j \sum_{f=1}^{k} v_{if} v_{jf} =$
$\frac{1}{2}\sum_{i=1}^{p}\sum_{j=1}^{p}\sum_{f=1}^{k} x_i x_j v_{if} v_{jf} - \frac{1}{2}\sum_{i=1}^{p}\sum_{f=1}^{k} x_i x_j v_{if} v_{jf} =$
$\frac{1}{2}(\sum_{i=1}^{p}\sum_{j=1}^{p}\sum_{f=1}^{k} x_i x_j v_{if} v_{jf} - \sum_{i=1}^{p}\sum_{f=1}^{k} x_i x_j v_{if} v_{jf}) =$
$\frac{1}{2}\sum_{f=1}^{k}((\sum_{i=1}^{p} v_{if} x_i)(\sum_{j=1}^{p} v_{jf} x_j) - \sum_{i=1}^{p} v_{if}^2 x_i^2) =$
$\frac{1}{2}\sum_{f=1}^{k}((\sum_{i}^{p} v_{if} x_i)^2 - \sum_{i=1}^{p} v_{if}^2 x_i^2)$

This leads to a reformulation of (eq 1) as
$\hat{y}(\mathbf{x}) = w_0 + \sum_{i=1}^{p} w_i x_i + \frac{1}{2}\sum_{f=1}^{k}((\sum_{i}^{p} v_{if} x_i)^2 - \sum_{i=1}^{p} v_{if}^2 x_i^2)$

which has $O(pk)$ complexity.

Rendle also notes that FMs can be generalized to higher degrees, but pretty much leaves it at that. Conceptually, the generalization is straightforward, but the algebra becomes a bit dauting. Interesting work in simplyfing this aspect can be found in Blondel et. al. 2016.

Learning FMs

We can learn FMs by minimising common loss functions.

- Binary classification: $l(\hat{y}(\mathbf{x}), y) = -\ln \sigma(\hat{y}(\mathbf{x})y)$, where σ is the sigmoid function
- Regression: $l(\hat{y}(\mathbf{x}), y) = (\hat{y}(\mathbf{x}) - y)^2$

In both cases we should apply L^2 regularization to avoid overfitting. Model parameters can be grouped, and each group can be assigned an independent regularization value λ. The same holds for $\mathbf{w}$ and w_0, and factorization layers $f \in \{1, \dots, k\}$. In practice, using many independent regularization values will introduce substantial computational overhead.

In Rendle 2012 and related work, it is shown that FMs can be learnt by ALS or, if we interpret them within a probabilistic framework, by MCMC sampling. For the purpose of this article I'll do gradient descent using adagrad as the optimisation strategy of choice.

Tensorflow

Tensorflow is an open source numerical computation framework released by Google in 2015. It has gained popularity in the Deep Learning community, where it is used to model large neural networks. By simplifying things a lot, we can think of neural networks as a series of matrix multiplication operations.

Tensorflow let's a programmer *declare* these operations, build a dependency graph of relationships, and execute it on a C++ backend. Nodes of the graph are called `operations`. Each `operation` takes one or more multi-dimensional arrays (`Tensors`) and performs some computation that generate zero or more `Tensors`. Conceptually the framework is simple, yet comes with a rich library of built-in and contributed modules. In particular, it contains a wide range of optimizers (SGD, adagrad, adam, ...).

We have all the components to implement an order 2 FM, and run it on a (nvidia) GPU without having to worry about low level CUDA details. For brownie points, we can even do it in Python. Scaling to clusters of CPUs or GPUs is also relatively easy.

FMs with Tensorflow

In this section I'll show how to implement FMs with Tensorflow, and learn movie ratings from the dummy data shown in Rendle 2010

```
import numpy as np
# Example dummy data from Rendle 2010
# http://www.csie.ntu.edu.tw/~b97053/paper/Rendle2010FM.pdf
# Stolen from https://github.com/coreylynch/pyFM
# Categorical variables (Users, Movies, Last Rated) have been one-hot-e
x_data = np.matrix([
#    Users  |     Movies     |    Movie Ratings   | Time | Last Movies 
#   A  B  C | TI  NH  SW  ST | TI   NH   SW   ST  |      | TI  NH  SW  
    [1, 0, 0,  1,  0,  0,  0,   0.3, 0.3, 0.3, 0,     13,   0,  0,  0,
    [1, 0, 0,  0,  1,  0,  0,   0.3, 0.3, 0.3, 0,     14,   1,  0,  0,
    [1, 0, 0,  0,  0,  1,  0,   0.3, 0.3, 0.3, 0,     16,   0,  1,  0,
    [0, 1, 0,  0,  0,  1,  0,   0,   0,   0.5, 0.5,   5,    0,  0,  0,
    [0, 1, 0,  0,  0,  0,  1,   0,   0,   0.5, 0.5,   8,    0,  0,  1,
    [0, 0, 1,  1,  0,  0,  0,   0.5, 0,   0.5, 0,     9,    0,  0,  0,
    [0, 0, 1,  0,  0,  1,  0,   0.5, 0,   0.5, 0,     12,   1,  0,  0,
])
# ratings
y_data = np.array([5, 3, 1, 4, 5, 1, 5])

# Let's add an axis to make tensoflow happy.
y_data.shape += (1, )
```

First we'll declare a model in Python, then we'll execute it within a `Session` context on the C++ backend. The code that follows is not the most pythonic, but for the purpose of this notes I'd rather be explicit about the Tensorflow API.

```
import tensorflow as tf
n, p = x_data.shape

# number of latent factors
k = 5

# design matrix
X = tf.placeholder('float', shape=[n, p])
# target vector
y = tf.placeholder('float', shape=[n, 1])

# bias and weights
w0 = tf.Variable(tf.zeros([1]))
W = tf.Variable(tf.zeros([p]))

# interaction factors, randomly initialized
V = tf.Variable(tf.random_normal([k, p], stddev=0.01))

# estimate of y, initialized to 0.
y_hat = tf.Variable(tf.zeros([n, 1]))
```

We use `Placeholders` for the inputs and targets. The actual data will be assigned at run time in the `Session`. `X` and `y` won't be further modified by the backend; we use `Variables` to hold bias, weights and factor layers. These are the parameters that will be updated when fitting the model.

In the following code we compute WX and use `reduce_sum()` to add together the row elements of the resulting `Tensor` (axis 1). `keep_dims` is set to `True` to ensure that input/output dimensions are respected.

```
linear_terms = tf.add(w0,
                tf.reduce_sum(
                tf.multiply(W, X), 1, keep_dims=True))
```

In the snippet above we just implemented linear regression.

We do the same for the interaction terms.

```
interactions = (tf.multiply(0.5,
                tf.reduce_sum(
                    tf.sub(
                        tf.pow( tf.matmul(X, tf.transpose(V)), 2),
                        tf.matmul(tf.pow(X, 2), tf.transpose(tf.pow(V, 
                    1, keep_dims=True)))
```

And add everything together to obtain the target estimate.

```
y_hat = tf.add(linear_terms, interactions)
```

Since we are solving a regression problem, we'll learn the model parameters by minimizing the sum of squares loss function. We also add a regularization term to prevent overfitting.

```
# L2 regularized sum of squares loss function over W and V
lambda_w = tf.constant(0.001, name='lambda_w')
lambda_v = tf.constant(0.001, name='lambda_v')

l2_norm = (tf.reduce_sum(
            tf.add(
                tf.multiply(lambda_w, tf.pow(W, 2)),
                tf.multiply(lambda_v, tf.pow(V, 2)))))

error = tf.reduce_mean(tf.square(tf.sub(y, y_hat)))
loss = tf.add(error, l2_norm)
```

To train the model we instantiate an `Optimizer` object and `minimize` the loss function.

```
eta = tf.constant(0.1)
optimizer = tf.train.AdagradOptimizer(eta).minimize(loss)
```

We are ready to compile the graph, and launch it on the Tensorflow backend. We use a python `context manager` construct to handle the `Session`.

```
# that's a lot of iterations
N_EPOCHS = 1000
# Launch the graph.
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(N_EPOCHS):
        indices = np.arange(n)
        np.random.shuffle(indices)
        x_data, y_data = x_data[indices], y_data[indices]
        sess.run(optimizer, feed_dict={X: x_data, y: y_data})

    print('MSE: ', sess.run(error, feed_dict={X: x_data, y: y_data}))
    print('Loss (regularized error):', sess.run(cost, feed_dict={X: x_d
    print('Predictions:', sess.run(y_hat, feed_dict={X: x_data, y: y_da
    print('Learnt weights:', sess.run(W, feed_dict={X: x_data, y: y_dat
    print('Learnt factors:', sess.run(V, feed_dict={X: x_data, y: y_dat
```

At each iteration (up to `N_EPOCHS`) we execute `optimizer`, which updates the model parameters by gradient descent. Note how we are moving data from the Python memory space to the C++ Tensorflow backend via `feed_dict={}`. Since we are dealing with toy data, we can pass the dataset all at once. In practice, we will want to work with mini batches (eg. use a generator over the input). We shuffle data (`np.random.shuffle`), to avoid biasing the gradient.

On my system, the `print()` statements generate the following output:

```
MSE:  0.602002
Loss (regularized error): 0.648635
Predictions: [[ 5.47903728]
 [ 1.89887238]
 [ 4.07966614]
 [ 5.53690434]
 [ 2.12006783]
 [ 4.45852327]
 [ 5.5077672 ]]
Learnt weights: [ 0.14918193  0.21650925 -0.09897757  0.0068595  -0.040
  0.09454302  0.00364213  0.11416676  0.09191741  0.18406411  0.1066898
  0.13829312 -0.15434285  0.09454302  0.        ]
Learnt factors: [[ 0.00705004  0.08103083 -0.01872271  0.00544881 -0.10
  -0.00434609 -0.02191383 -0.02647865  0.03837667  0.06087479  0.044025
   0.04876902 -0.15307751  0.00196489  0.00550045]
 [ 0.0546066   0.2176538  -0.01303235  0.09689698 -0.15466486  0.128942
   0.03864328  0.03093899  0.01689298  0.08874346  0.18454561  0.155368
   0.15537314 -0.26848108  0.03202138 -0.00277198]
 [-0.06930697 -0.18463977  0.15689404 -0.12465551  0.13754503 -0.183412
  -0.0379773  -0.01738735 -0.14120492 -0.09994929 -0.17874891 -0.167369
  -0.20745522  0.22959492 -0.05229255  0.00680638]
 [-0.01874499 -0.18115361  0.03584651 -0.04214986  0.1509551  -0.095532
  -0.03395364 -0.00030072  0.00359597 -0.07835191 -0.16964239 -0.123653
  -0.11962538  0.25852579 -0.03928068 -0.01048287]
 [-0.01707606 -0.16321027  0.05072568 -0.0653125   0.14090565 -0.060631
  -0.01582224  0.02301382 -0.01347715 -0.07161148 -0.15456919 -0.113869
  -0.09024142  0.24246764 -0.02350813 -0.00971563]
 [ 0.12528357  0.06116699 -0.03955081  0.08235611 -0.01942447  0.091316
  -0.05486022  0.0194256   0.12990384  0.0041019   0.05420737  0.065460
   0.0393628   0.03204706 -0.05093096  0.00359864]
 [-0.18544145 -0.23963821  0.17378353 -0.22184615  0.15998147 -0.236894
  -0.01744101 -0.04991474 -0.1909833  -0.12747602 -0.21516703 -0.196893
  -0.27546856  0.25545105 -0.01022351  0.00644435]
 [ 0.04372156  0.19893605 -0.05931458  0.09953506 -0.16382113  0.085437
   0.03261247  0.00653346  0.02534093  0.07904716  0.17823988  0.157494
   0.12673798 -0.28109354  0.04243347 -0.00722708]
 [-0.04402747 -0.21142682  0.01911086 -0.09846341  0.17138375 -0.113948
  -0.04136728 -0.0086472  -0.04622135 -0.10258008 -0.19500685 -0.164327
  -0.14555235  0.28950861 -0.04734635  0.00215936]
 [-0.09106413 -0.01975513  0.04174449 -0.06765321 -0.04338175 -0.020663
  -0.01545047 -0.05105948 -0.08453009 -0.0035413  -0.03348914 -0.008010
   0.0212482  -0.05964642 -0.00983996 -0.01162685]]
```

Experiments

To get a feeling of the overall perfomance and correctness of the model, I trained and tested FMs on the movielens 100k dataset. The goal is to predict the rating of each movie; I trained on `ua.base` portion of the dataset, and used `ua.test` to test. After one-hot-encoding categorical variables, I obtained a 90570×2623 design matrix for the training set, and 9430×2623 for the test set.

I trained a model using the adam optimizer, with `learning_rate` initalized to 0.01. I built a model with 20 factors, with adam running for 100 iterations. With this setting I obtained the following:

- MSE train: 0.779672
- MSE test: 0.907205352128

The numbers look pretty much in line with known performance in this benchmark. FM uses gradient descent on a non-convex objective, but in practice when using adam on movielens I did not notice much sensitivity to local minima. I trained the model on an AWS `p2.xlarge` instance both on CPU and GPU (Nvidia Tesla K80). My best of ten run times, training for 10 iterations, are 120s and 20s respectively. CPU performance might be impacted by the poor parallelism on this instance, but all in all the speed up of running FM on a GPU is not bad at all.

Conclusion

In this article I gave a brief summary of Factorization Machines. I showed how to implement the model in a few lines of python in Tensorflow. I tested the model accuracy on movielens 100k, and reported results in line with known benchmarks. It would be nice to do a more thorough evaluation of the model, on larger datasets, and compare the performance with the reference `libfm` implementation.