Victor

Follow 208 Followers About

Collaborative Filtering using Deep Neural Networks (in Tensorflow)

>|< Victor Jun 21, 2019 · 10 min read

In this story, we take a look at how to use deep learning to make recommendations from implicit data. It's based on the concepts and implementation put forth in the paper Neural Collaborative Filtering by He et al. from 2017. If this is your first foray into making recommendations from implicit data, I can recommend having a look at my implementations of ALS or BPR since we will be building on many concepts covered in depth in those stories.

Also, note that this is not an introduction to Tensorflow or deep neural networks in general.

Content:

Introduction

Neural networks have been used quite frequently to model recommendations but are most often used for auxiliary information, for example having a model "read" video descriptions and find similarities or analyze music to understand what genre a track is. These signals are then often combined with a different collaborative model (like matrix factorization) to create the final prediction score. This approach is, of course, very interesting in and of itself but in the paper, the goal is to see if we can utilize a neural network as the core of our collaborative model.

In short, our aim here is to use a deep neural network to build a collaborative filtering model based on implicit data.

Why deep learning?

On a theoretical level, why would we have a reason to believe that using a neural network will improve the performance of our model? The reasoning here is that the linearity of a regular matrix factorization approach limits the expressiveness of a model, i.e how complex of a relationship it can model between all of our users and items.

The fact that users and items are modeled in the same latent space and that scores are calculated using a linear operation, the dot-product, means we can get into situations where we can't represent the relationship between user ***a*** and user ***b*** without violating a previously established relationship between user ***b*** and user ***c***. This means there might be no way to accurately represent a new user in a latent space with respect to all other user representations. This problem becomes apparent when using a lower number of latent dimensions and extreme if you imagine only one latent dimension.

One way to minimize the problem would be to increase the dimensionality of the latent space to allow for more expressiveness and more complex relationships. This, however, can mean that the model becomes less generic, i.e overfits more easily as well as taking longer to train in order to reach desirable results.

So the main idea of using a deep neural network is to learn a *non-linear* function rather than a linear one and in doing so hopefully increase the expressiveness of the final model.

As a small aside, YouTube released a paper in 2016 describing how they use deep neural networks to power their recommendation engine, although it's quite different and more involved than what we will be building here, it's an interesting read.

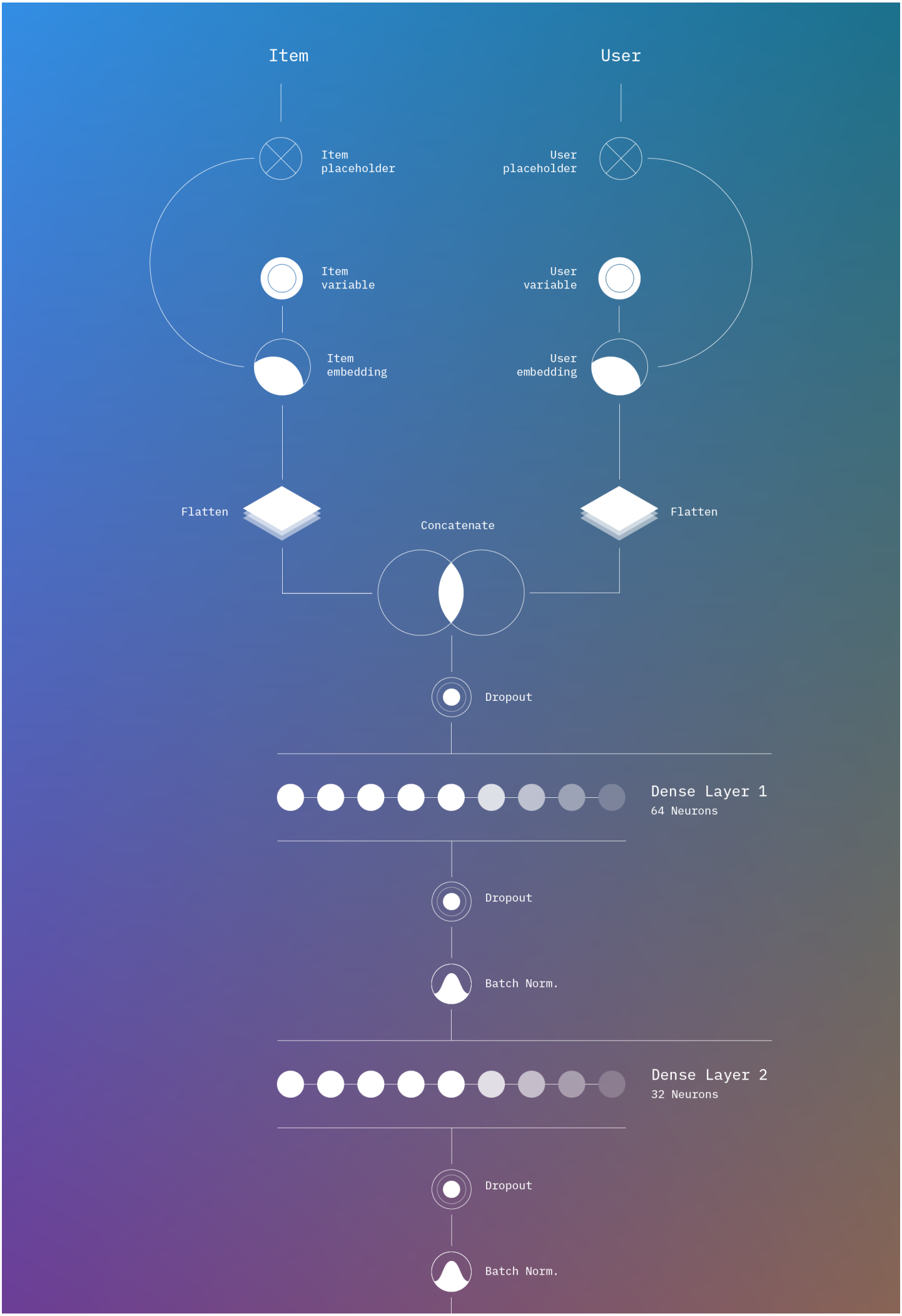
The model

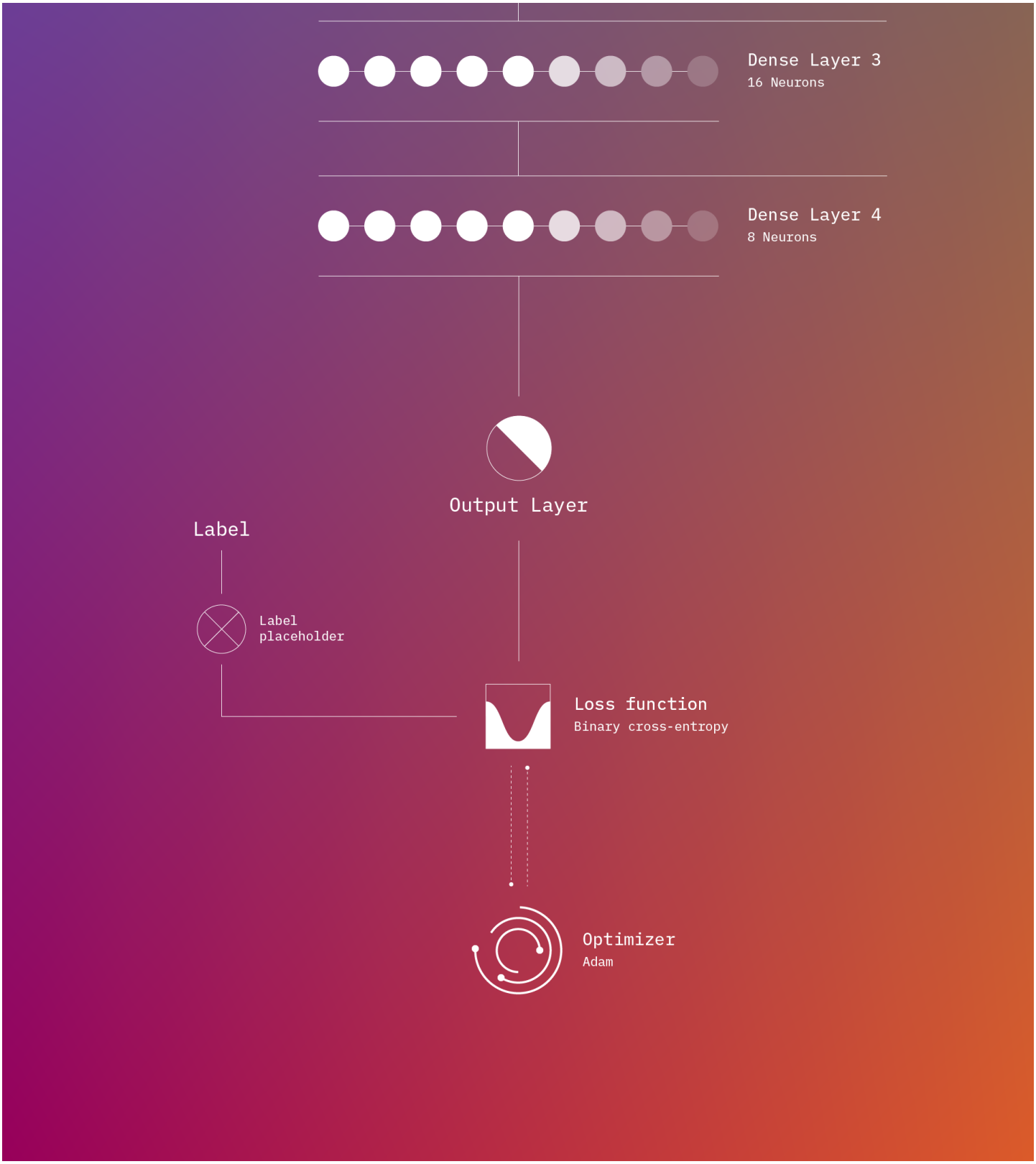
The model proposed in the paper and the one we will be implementing here is actually based on two different networks that are then combined to calculate the final score. On the one hand, we have a standard *matrix factorization network* (GMF) and on the other a *multilayer perceptron network* (MLP). The idea is to take advantage of both the linearity and non-linearity of the two networks.

Multilayer perceptron network (MLP)

A multilayer perceptron, or MLP, is basically just a fancy name for *"the simplest kind of deep neural network"*. We have at least one hidden layer of neurons and each neuron has some kind of non-linear activation function. The layers are then densely connected, so each neuron in one layer is connected to every neuron in the next layer and so on.

Our MLP network consists of four dense hidden layers and ReLu (a non-linear function) as the activation.


Item
User
Item placeholder
User placeholder
Item variable
User variable
Item embedding
User embedding
Flatten
Flatten
Concatenate
Dropout
Dense Layer 1
64 Neurons
Dropout
Batch Norm.
Dense Layer 2
32 Neurons
Dropout
Batch Norm.

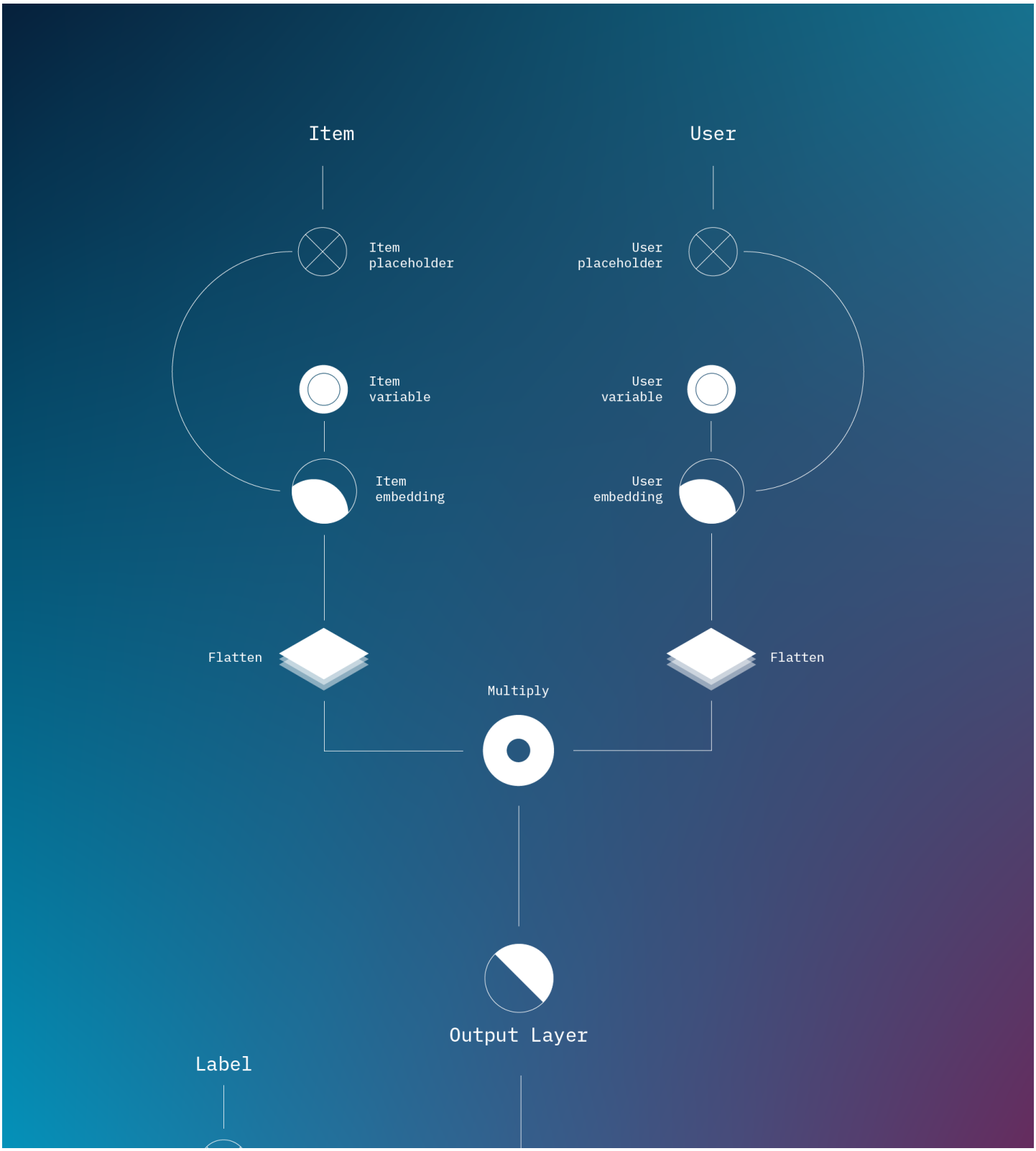


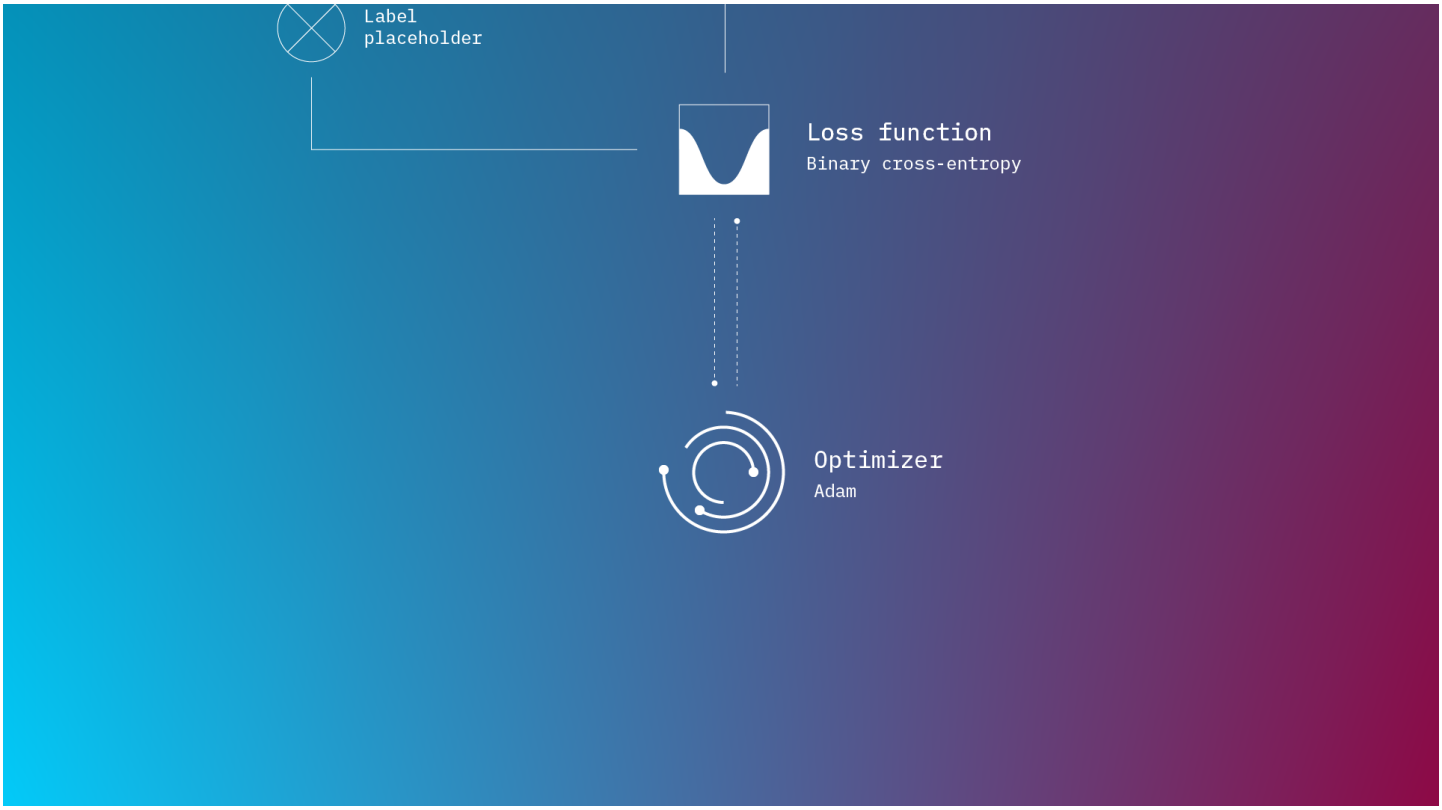
We first set up embeddings for our user and items, flatten them and concatenate them together. We then add a first dropout layer followed by our four hidden layers consisting of 64, 32, 16 and lastly 8 fully connected neurons. Between hidden layer 1 — 2 and 2 — 3 we add a dropout and batch normalization layer. Finally, we have our output layer that is just a single fully connected neuron. As our loss function, we use binary cross-entropy and for our optimizer we use Adam.

Note that this is just one implementation of this network and we can play around with using more/fewer layers with a different number of neurons. We can also change where and how we add dropouts and batch norms etc.

Matrix factorization network (GMF)

The GMF network is basically just a regular point-wise matrix factorization using SGD (Adam in this case) to approximate the factorization of a (*user x item)* matrix into the two matrices (*user x latent features)* and (*latent_features x items)* respectively.

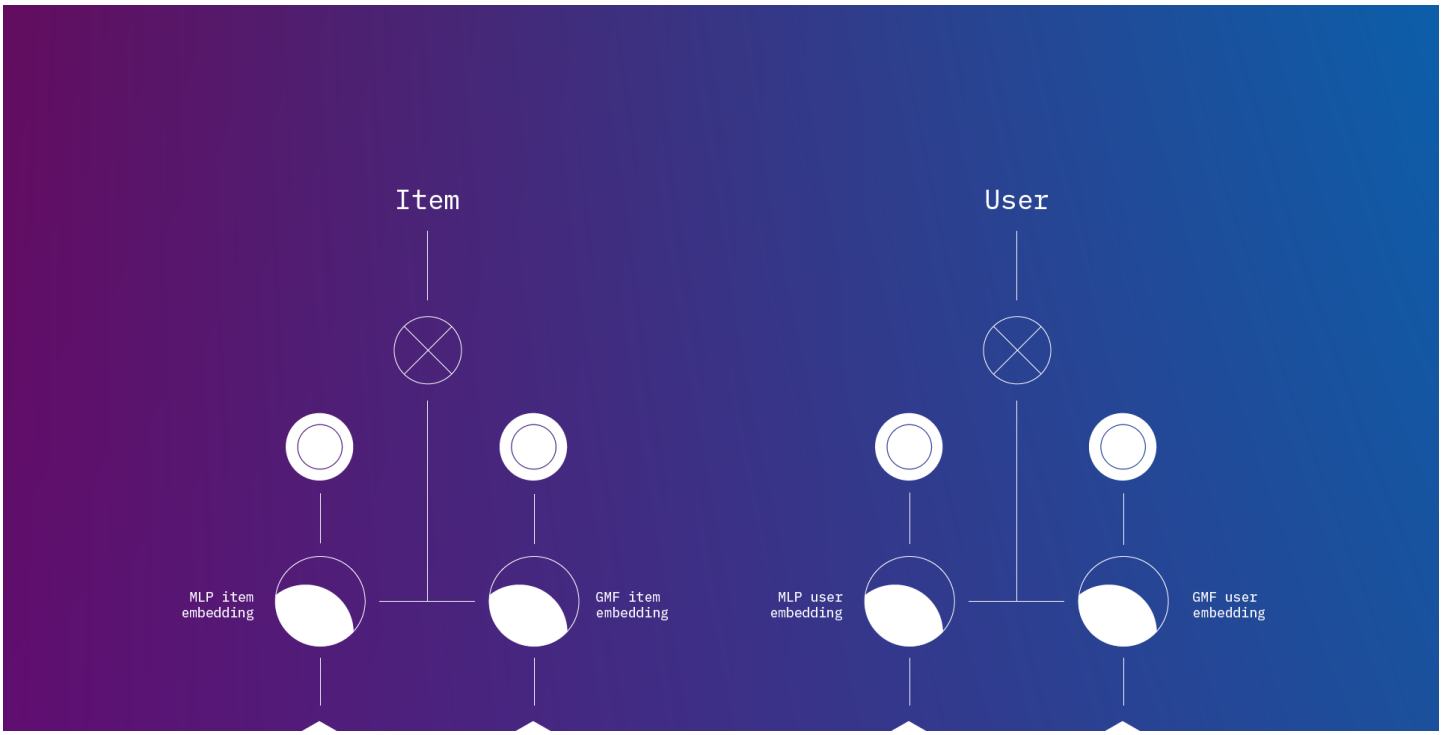


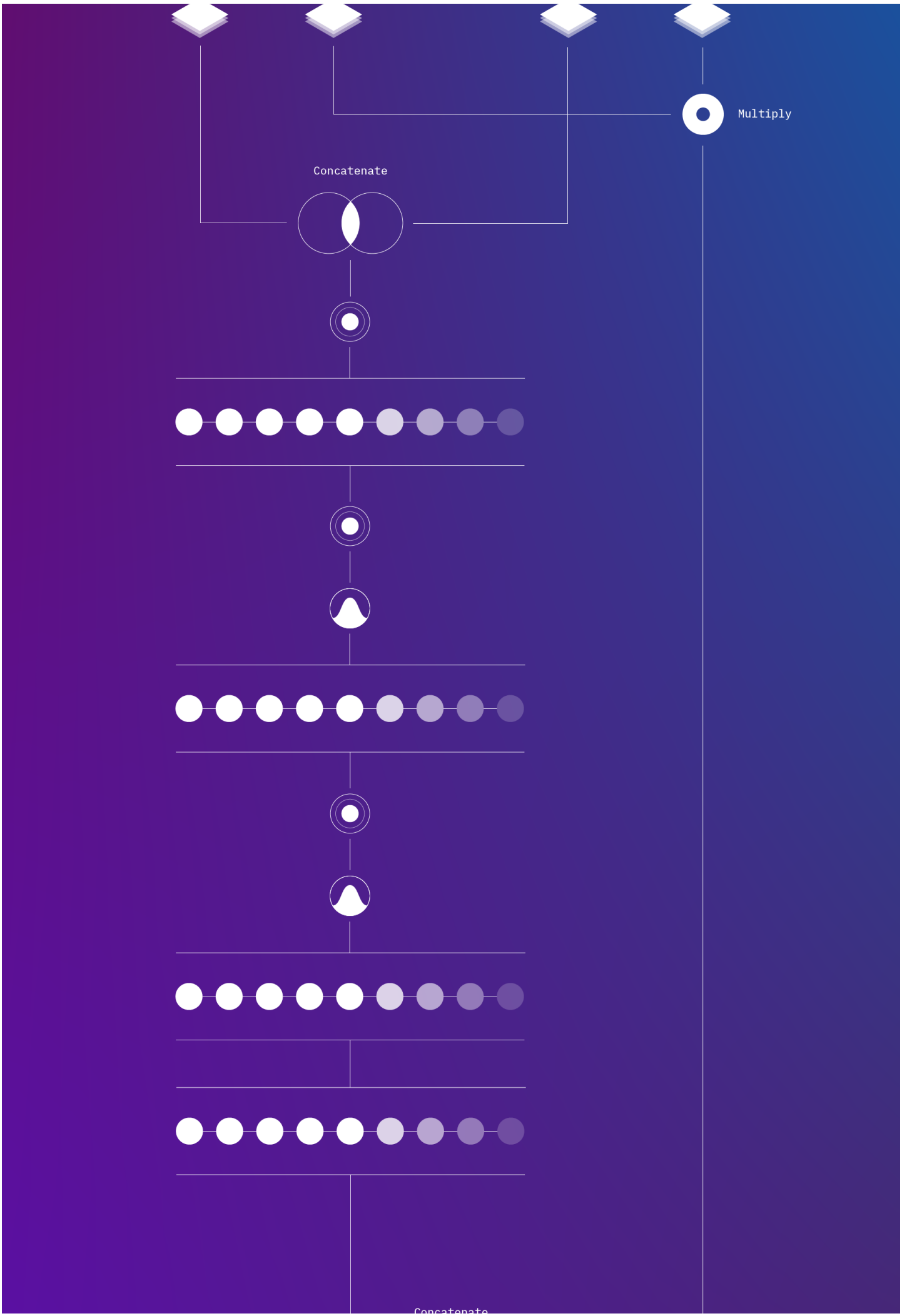


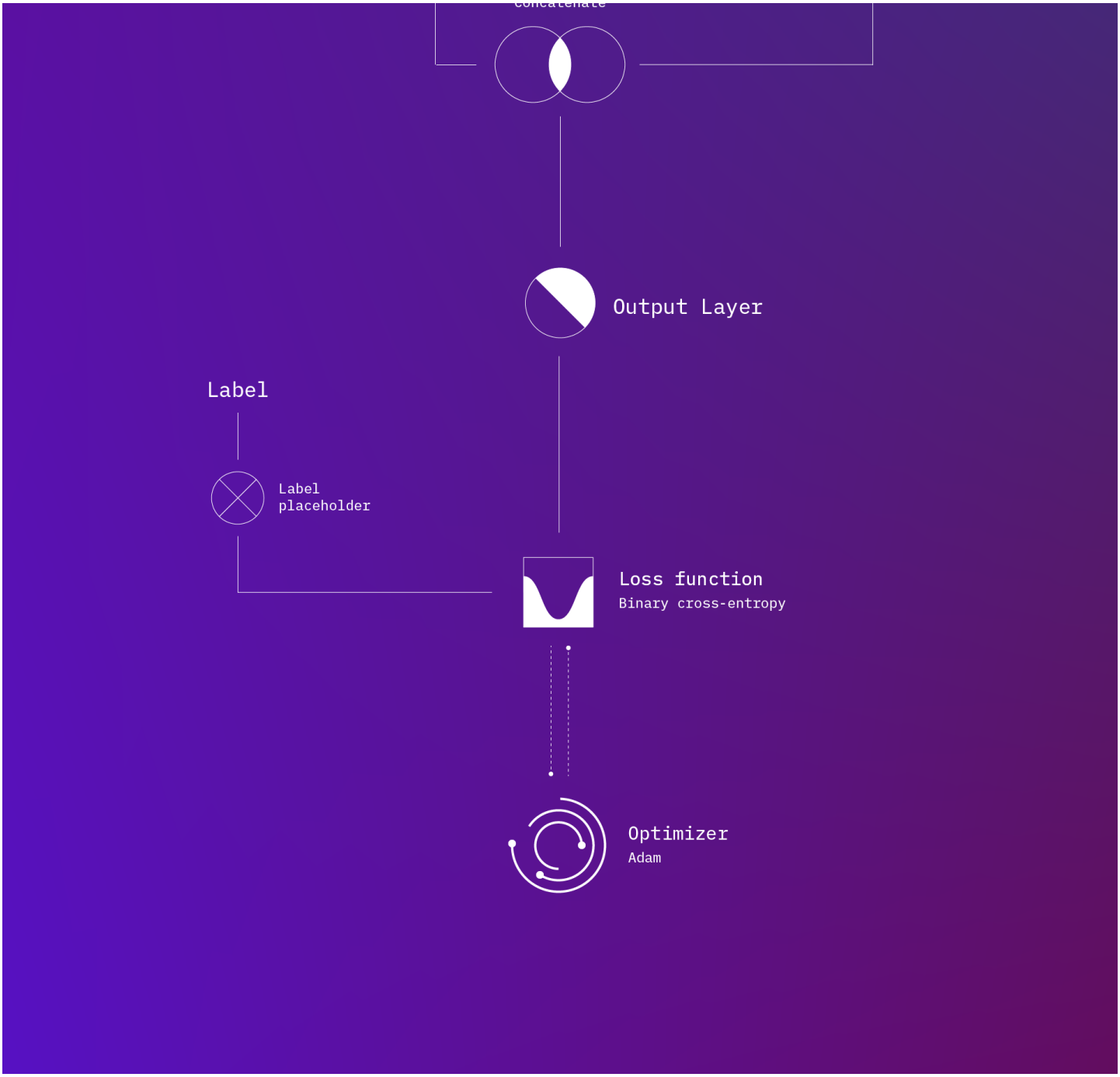
A much simpler network. Here we do the same setup as with MLP for our user and item embeddings and then multiply them together. Our final output layer is also a dense layer with a single neuron. We again use binary cross-entropy loss with Adam.

Combining the two networks (NeuMF)

Now for the final setup, we combine the GMF and MLP networks together into what we call a NeuMF network. Basically, we just concatenate the result of both networks together.




Multiply
Concatenate
Concatenate



We create two embeddings for our users, one for the GMF network and one for the MLP one. We do the same for our items and all four embeddings are flattened and then fed into their respective networks. The two networks are then concatenated together and we add a single output neuron to our now merged model. From here it's the same setup as before with a binary cross-entropy loss and Adam optimizer.

The dataset

We will be using the lastfm dataset. I think it makes sense to keep the dataset consistent between my different implementations but for the next one I'll probably switch things up a bit. Anyway, the dataset contains the listening behavior of 360,000

users including user IDs, artist IDs, artist names and the number of times a user played an artist.

Evaluating the model

Before we start writing code, let's have a look at how we can assess the accuracy of our model.

The paper uses top@K for evaluation so we'll be using the same here. The idea is to remove one of the known positive interactions for a user and then check if that item is present in the top K recommended items. So say we set $K=10$ we can get a percentage of how often our known but "removed" items would be recommended if we showed the user 10 recommendations. It does not, however, take into account where in that batch of 10 the item appeared.

What value we use for K here is a bit arbitrary, but if we were developing a recommender for use in a real implementation where the UI only shows the user say 5 recommended items we should, of course, use $K=5$ to model the real world conditions.

Ok, let's write it!

There will be a couple of steps to this implementation. First, we will look at how we load the last.fm dataset, then how to implement each part of the network independently and last how to combine them together into our final model.

There will be quite a bit of code so if you're only interested in the final network setup you can skip to the end.

1. Loading and preparing the data

As always, we start with loading our dataset and doing some wrangling.

Once we have our data loaded and in the format we want we split it into a train and a test sets. For the test set, we select the latest interaction for every user (the holdout item) and remove that interaction from our training data. We also sample 100 negative interactions for each user and add these to our test data. The reason for this is to avoid raking every item in our dataset when we do our evaluation. Instead, we will rank these 101 items and check if our holdout was among the K highest ranked ones.

To get this all to work we will need to define a couple of helper functions:

Now we have our data loaded and split but we still need to do some work before we can start working on our Tensorflow graph.

2. Helper functions

We define a couple of helper functions that we'll use as we train our model and to evaluate our results after training.

We first define *get_train_instances*. This function samples a number of negative (unknown) interactions for every known user-item interaction in our training data. In this example, we get 4 negative interactions for each positive one. We then also define *random_mini_batches* used to generate randomized batches of size 256 during training.

Last, we define *evaluate*, *eval_rating* and *get_hits,* a couple of functions needed to generate predictions from our test data and calculate the top@K value we discussed earlier.

3. The MLP model

Time to actually start implementing our model. We start by importing the libraries we'll need, defining the hyperparameters and constructing our Tensorflow graph.

There is nothing really special in the setup here. Most of the code is defining our placeholders, variables, and embeddings. After that, it's a standard dense network of four hidden layers with a single neuron output layer with some dropout and batch normalizations.

As mentioned before, feel free to play around with everything from the number of layers to the number of neurons, the dropouts, etc.

Now let's feed some data into our graph and start training.

We loop over our epochs and for each epoch, we get our training input from the *get_trian_instances* function defined before and generate a set of mini-batches. For each mini-batch, we then feed data into our Tensorflow graph as we execute it. Lastly, we can call our *evaluate* function from before and get our top@K percentage.

Running the training for 10 epochs on a small subset of our full dataset (to save some time) with K=10 gave me a hit@K value of 0.884. This means that in roughly 88% of

cases our holdout item was in the top 10 recommended items. If we instead change K to 5 I get a value of 0.847 or around 85%. So a pretty good result there.

4. The GMF model

Now let's look at the linear matrix factorization part we'll use together with our MLP neural network to build our final recommendation model.

We don't need to change or add any new helper function or evaluation functions. What's different is only how we define the graph.

As always we define our hyperparameters. These are mostly the same as for MLP with the additions of the latent_features parameter. As the name implies this will be the dimensionality of the latent space of our user and item factorized vectors.

So we'll create one vector for users of shape *users x latent features* and one for items of shape *latent features x items.* The goal is then to update and learn the values of these two matrices to that by multiplying them together we get as close as possible to our original user x item interaction matrix.

To do this we define our inputs and embeddings, our target matrix as the sum of these embeddings and a single neuron output layer. We again use binary cross-entropy as our loss and Adam to optimize it.

We can now run the graph using the same code as we used in the MLP example earlier. After 10 epochs on the same subset of the data as before we get a hit@K value of 0.902 or 90%. So slightly higher than for our neural network.

Note: Instead of our very simple matrix factorization function implemented here, we could potentially use a BPR or ALS model to factor our matrices. I have not tested this though so not sure how it would impact the performance or the final result.

5. Putting it all together, the NeuMF model.

So now that we have both the linear and non-linear component, let's combine them together into the final NeuMF model.

Again, all of the evaluation and helper code will be the same as before.

Both MLP and GML are implemented in parallel and then concatenate together before the final output layer. After that, we set up our loss and optimizer as before.

Now when running the model on the same subset of our dataset we get a hit rate of 92% for K=10.

Summary

Looking at the performance of our different models on a graph we can see the improvements in performance over time as we increase the number of negative samples for each known interaction.

If you wonder why we dropped from the roughly 90% we had before to around 70%, it's because I'm using an even smaller subset the data here (to save on my laptops poor CPU).

In this example, we're running for 10 epochs and with K=10 and just as in the examples in the paper (using the MovieLens and Pinterest datasets), we see that NeuMF outperforms both our stand-alone models.

References

- https://arxiv.org/pdf/1708.05031.pdf
- https://arxiv.org/pdf/1710.05613.pdf
- https://github.com/hexiangnan/neural_collaborative_filtering

- https://github.com/jfkirk/tensorrec

Machine Learning TensorFlow Recommender Systems Neural Networks Artificial Intelligence