How to Implement a Recommendation System with Deep Learning and PyTorch

Applying neural network to make a simple recommendation system for MovieLens ratings dataset

Ilia Zaitsev Follow

Aug 17, 2018 · 7 min read

Recently I've started watching fast.ai lectures—a great online course on Deep Learning and its applications. In one of his lectures, the author discusses the building of a simple neural network based recommendation system with application to the MovieLens dataset. While the lecture is an excellent source of information on this topic, it mostly relies on the library developed by the authors to run the training process. The library is quite flexible and provides several levels of abstractions.

However, I strongly wanted to learn more about the **PyTorch** framework which sits under the hood of authors code. In this post, I am describing the process of implementing and training a simple embeddings-based collaborative filtering recommendation system using PyTorch, Pandas, and Scikit-Learn. We're going to follow the steps described in the lecture without using the mentioned library.

> ***TL;DR**: Please use this link to navigate straight to the Jupyter notebook with the PyTorch implementation discussed in this article.*

Photo by Tommy van Kessel on Unsplash

Collaborative Filtering

Whenever you're visiting an online store, a video or audio streaming service, or any other content delivery platform, you almost certainly get a bunch of recommendations based on your preferences, previous purchases, and visited pages. One of the most straightforward algorithms to implement a system capable of giving advice based on previous experience is a technique called collaborative filtering. The main idea is to predict the reaction of a user on a specific item based on reactions of "similar" users where the "similarity" is calculated using the ratings or reviews left by these users.

Conceptually, we are building a matrix (table) where rows identify users and columns—their ratings. Note that for a real dataset this matrix is going to be *very sparse*, i.e., most of its cells are empty because usually, we have much more items comparing to the number of users who bought/watched them. Then we use a training algorithm to infer similarities between users rating patterns to "fill the gaps", to predict ratings that are missing.

Users vs Movies

	A Game of Thrones	The Lord of The Rings	Star Trek	Star Wars	Titanic
Alice	5	4	3	3	5
Bob	5	n/a	4	5	3
Carol	n/a	3	5	n/a	2
Danny	4	4	5	3	4
Eve	5	3	?	3	5

Tabular representation of movie ratings

For example, let's pretend that we have a group of people who rated a set of movies as the picture above shows. Note that some of them have rated all the movies (Alice and Danny) while others haven't (Bob and Carol). Consider a new customer Eve who has already watched all of the movies except *Star Trek*. How can we predict her rating for this specific item? For this purpose, we are going to use an averaged opinion of her neighbors who have similar preferences. Eve likes *A Game of Thrones* and *Titanic* but is not a fan of *The Lord of The Rings* and *Star Wars*. Alice's and Danny's ratings pattern resembles Eve's row. Therefore, we can suppose that Eve should probably have a more or less positive opinion about *Star Trek*.

In the next section, we are going to take a look at the real dataset with movies ratings and discuss how to prepare it to train a neural network.

MovieLens Dataset

Two data frames represent the dataset we're going to analyze: (a) users ratings per movie and (b) meta-information about movies, specifically, their title and genre. To train the model, we only need the data frame (a) while the second one we're going to use for the trained model interpretation only.

(a) Ratings

userId	movieId	rating
1	1	5
2	2	4
3	3	2
4	1	4
5	2	3
6	3	2
7	1	5
8	3	1
9	4	4

(b) Movies

movieId	title	genres
1	Toy Story	Animation|Children's|Comedy
2	Jumanji	Adventure|Children's|Fantasy
3	Grumpier Old Man	Comedy|Romance
4	Waiting to Exhale	Comedy|Drama
5	Father of the Bride Part II	Comedy

A machine learning algorithm expects an array with numerical values. Technically speaking, our dataset fits those requirements as soon as all its columns are numerical. However, we shouldn't pass users and movies IDs directly into the algorithm because it will try to infer the dependencies between values which don't exist. These numbers are not related to each other and used for identification purposes only.

A conventional approach to alleviate the issue would be to use a one-hot encoding, what means to replace the categorical columns with "dummy" 0/1 columns. It is a good-working technique in cases when there are a few categories but we have thousands of movies and users.

Here where the embeddings come into play. Instead of assigning a separate column to each of categories, we are representing them as *vectors in N-dimensional space*. In other words, we use a look-up matrix that returns an array with *N* numbers for a given user's or movie's ID:

```
embedding = [
    [0.25, 0.51, 0.73, 0.49],
    [0.81, 0.11, 0.32, 0.09],
    [0.15, 0.66, 0.82, 0.91]
]
movie_id = 1
movie_vector = embedding[movie_id - 1]
```

We initialize the matrix with random values at first and then adjust them during the training process. For example, if we would have only five movies and five users, and pick *N* equal to four, then our randomly-

initialized embeddings matrices could look like the picture below shows.

Movies Embeddings

movieId	e1	e2	e3	e4
1	0.92	0.05	0.63	0.64
2	0.59	0.20	0.31	0.96
3	0.47	0.64	0.35	0.00
4	0.36	0.41	0.38	0.15
5	0.51	0.44	0.98	0.04

Users Embeddings

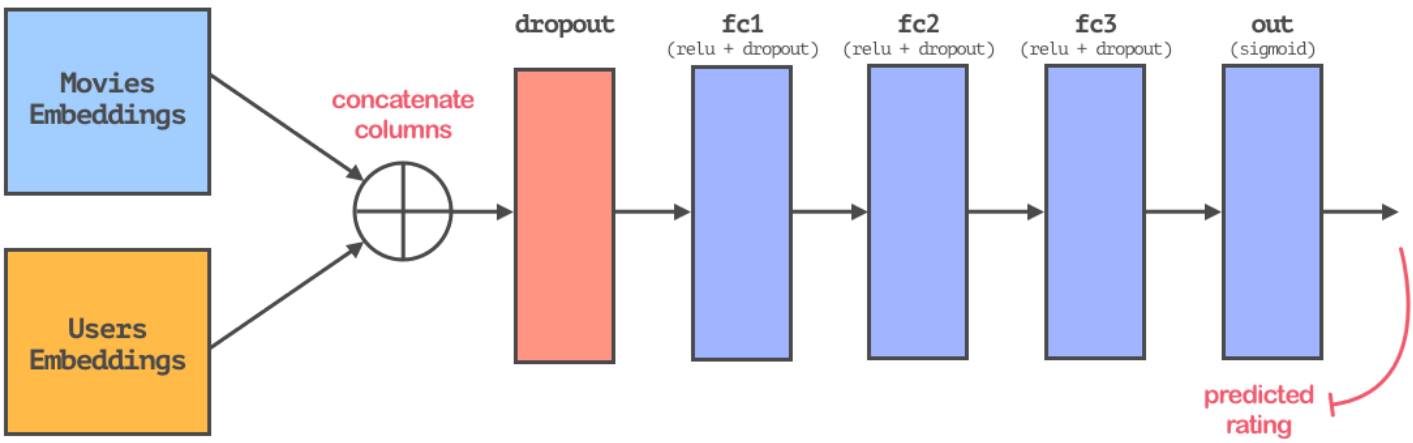
userId	e1	e2	e3	e4
1	0.70	0.75	0.36	0.21
2	0.67	0.41	0.29	0.13
3	0.91	0.25	0.54	0.27
4	0.93	0.53	0.49	0.45
5	0.34	0.05	0.15	0.61

Tables with randomly initialized embeddings vectors

This trick allows us to feed highly-dimensional categorical variables into a neural network. In the next section, we're going to show how this model could be built using PyTorch framework.

Embeddings Network

The PyTorch is a framework that allows to build various computational graphs (not only neural networks) and run them on GPU. The conception of tensors, neural networks, and computational graphs is outside the scope of this article but briefly speaking, one could treat the library as a set of tools to create highly computationally efficient and flexible machine learning models. In our case, we want to create a neural network that could help us to infer the similarities between users and predict their ratings based on available data.



The picture above schematically shows the model we're going to build. At the very beginning, we put our embeddings matrices, or look-ups, which convert integer IDs into arrays of floating-point numbers. Next,

we put a bunch of fully-connected layers with dropouts. Finally, we need to return a list of predicted ratings. For this purpose, we use a layer with sigmoid activation function and rescale it to the original range of values (in case of MovieLens dataset, it is usually from 1 to 5).

The snippet shows how one can write a class that creates a neural network with embeddings, several hidden fully-connected layers, and dropouts using **PyTorch** framework.

```
class EmbeddingNet(nn.Module):
    """
    Creates a dense network with embedding layers.

    Args:

        n_users:
            Number of unique users in the dataset.

        n_movies:
            Number of unique movies in the dataset.

        n_factors:
            Number of columns in the embeddings matrix

        embedding_dropout:
            Dropout rate to apply right after embeddin

        hidden:
            A single integer or a list of integers def
            units in hidden layer(s).

        dropouts:
            A single integer or a list of integers def
            layers rates applyied right after each of

    """
    def __init__(self, n_users, n_movies,
                 n_factors=50, embedding_dropout=0.02,
                 hidden=10, dropouts=0.2):

        super().__init__()
        hidden = get_list(hidden)
        dropouts = get_list(dropouts)
        n_last = hidden[-1]

        def gen_layers(n_in):
            nonlocal hidden, dropouts
            assert len(dropouts) <= len(hidden)

            for n_out, rate in zip_longest(hidden, dro
                yield nn.Linear(n_in, n_out)
                yield nn.ReLU()
                if rate is not None and rate > 0.:
```

```
                    yield nn.Dropout(rate)
                n_in = n_out

        self.u = nn.Embedding(n_users, n_factors)
        self.m = nn.Embedding(n_movies, n_factors)
        self.drop = nn.Dropout(embedding_dropout)
        self.hidden = nn.Sequential(*list(gen_layers(n
        self.fc = nn.Linear(n_last, 1)
        self._init()
```

For example, to create a network with 3 hidden layers with 100, 200, and 300 units with dropouts between them, use:

```
net = EmbeddingNet(
    n_users, n_movies,
    n_factors=150,
    hidden=[100, 200, 300],
    dropouts=[0.25, 0.5])

print(net)

------------------------------------------------------------
--------

EmbeddingNet(
  (u): Embedding(6040, 150)
  (m): Embedding(3706, 150)
  (drop): Dropout(p=0.02)
  (hidden): Sequential(
    (0): Linear(in_features=300, out_features=100,
bias=True)
    (1): ReLU()
    (2): Dropout(p=0.25)
    (3): Linear(in_features=100, out_features=200,
bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5)
    (6): Linear(in_features=200, out_features=300,
bias=True)
    (7): ReLU()
  )
  (fc): Linear(in_features=300, out_features=1, bias=True)
)
```

Training Loop

Finally, the last variable in our "equation" is the training process. We pick Mean-Squared Error loss as a metric of the quality of our network. The higher the error, the less accurate our ratings predictions. We also

use the learning rate cosine annealing with restarts technique to match the default configuration available in the **fastai** library out of the box.

```
# training loop parameters
lr = 1e-3
wd = 1e-5
bs = 2000
n_epochs = 100
patience = 10
no_improvements = 0
best_loss = np.inf
best_weights = None
history = []
lr_history = []

# use GPU if available
identifier = 'cuda:0' if torch.cuda.is_available() els
device = torch.device(identifier)

# setting up network, optimizer and learning rate sche
net.to(device)
criterion = nn.MSELoss(reduction='sum')
optimizer = optim.Adam(net.parameters(), lr=lr, weight
iterations_per_epoch = int(math.ceil(dataset_sizes['tr
sched_func = cosine(t_max=iterations_per_epoch * 2, et
scheduler = CyclicLR(optimizer, sched_func)

fmt = '[{epoch:03d}/{total:03d}] train: {train:.4f} -

# start training
for epoch in range(n_epochs):
    stats = {'epoch': epoch + 1, 'total': n_epochs}

    for phase in ('train', 'val'):
        training = phase == 'train'
        running_loss = 0.0
        n_batches = 0
        iterator = batches(*datasets[phase], shuffle=t

        for batch in iterator:
            x_batch, y_batch = [b.to(device) for b in
            optimizer.zero_grad()

            with torch.set_grad_enabled(training):
                outputs = net(x_batch[:, 1], x_batch[:
                loss = criterion(outputs, y_batch)
```

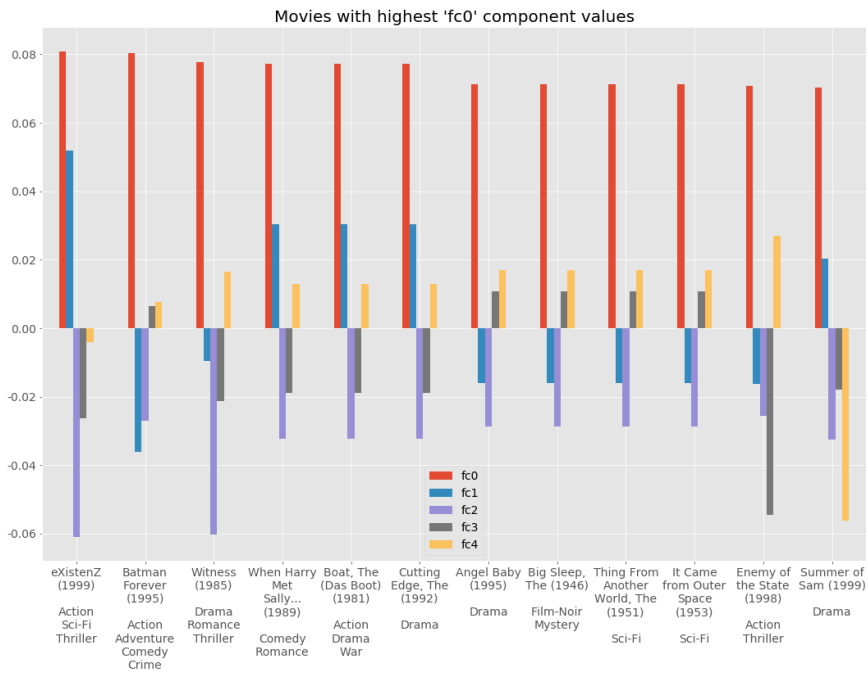
Please follow this link to see the full source code required to prepare the dataset and to train the model.

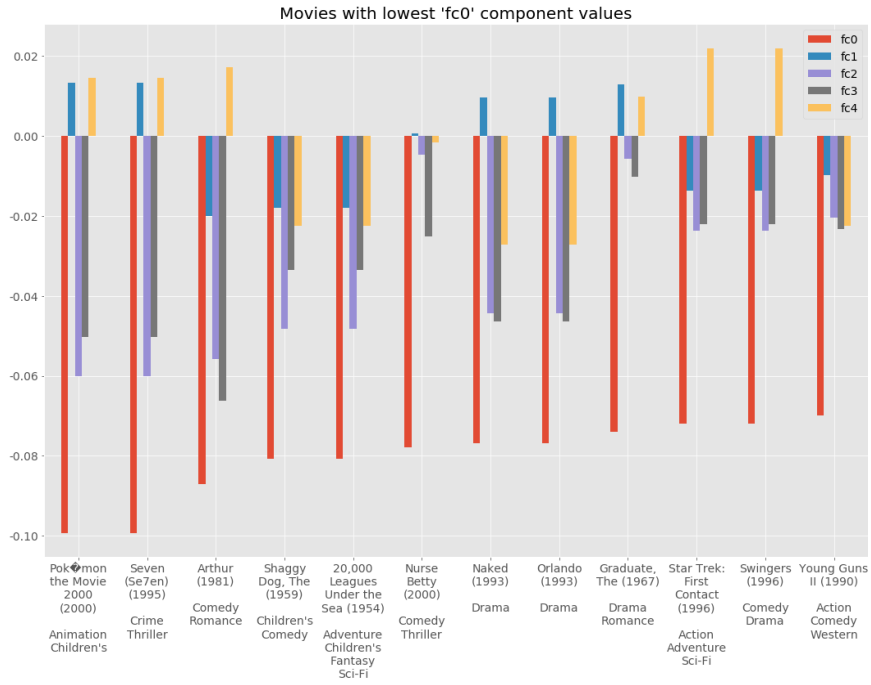
Bonus: Embeddings Visualization

When the model is trained and validated, we can ask a question: *How can we interpret the results*? Neural networks are usually considered to be black-box algorithms, and it could be difficult to interpret their weights in a meaningful way without applying specific visualization techniques.

However, in our case, we can try to interpret the training results using the embeddings matrices. After the training is completed, these matrices don't contain random values anymore and should somehow reflect the characteristics of our dataset. What if we try to visualize these embeddings to check if any patterns could be discovered?

For this purpose, let's apply the Principal Components Analysis to reduce the dimensionality and then pick a few samples from the movies embeddings. Then let's plot the examples with the high positive values of the first component, and the examples with the high negative values as the pictures below show.





From the visualizations above, we could guess that the “red” component mostly reflects the “amount of seriousness” in the movie. Comedies, animations, and adventures have the negative values of this component, while more “dramatical” movies have high positive levels of this component.

Conclusion

After a few hours spent with PyTorch, I can strongly advise this library to anyone who wants to build machine learning models. This library is worth your attention especially if you’re already familiar with Python. The library is quite intuitive and easily expandable and allows you to leverage the best features of the Python language for building models of various level of complexity.

. . .

Interested in Python language? Can’t live without Machine Learning? Have read everything else on the Internet?

Then probably you would be interested in my blog where I am talking about various programming topics and provide links to textbooks and guides I’ve found interesting.