Wide & Deep Book Recommender System

Author: Paul McSlarrow

Original paper: https://arxiv.org/abs/1606.07792.

This is not an implementation by the authors of the paper

Imports and Constants

```
import os
import random
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torch.utils.tensorboard import SummaryWriter
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from collections import Counter
from google.colab import drive

# torch.manual_seed(1)
```

```
RATINGS_PATH = '/content/drive/MyDrive/ratings.csv'
BOOKS_PATH = '/content/drive/MyDrive/books.csv'
BOOK_TAGS_PATH = '/content/drive/MyDrive/book_tags.csv'
TAGS_PATH = '/content/drive/MyDrive/tags.csv'
TRAIN_SIZE = 0.8
```

Dataset creation

This section joins multiple tables to produce a unified dataset. Some tables contain genre information, while others provide the count of each genre. The join creates a single table that attaches the **three most popular genres** for each `goodreads_book_id`.

While I am also using a different dataset from the paper, to try my best to keep consistency between the two, I will turn this into a binary classification problem rather than a 5-class classification problem, and binarize the rating label to be 0 or 1. If the rating was greater than or equal to 4, I'll say that the user liked it, otherwise, the user didn't particularly love it and will be set to 0. This will help me utilize binary cross entropy loss as the paper did, which also allows me to use the Sigmoid activation function to enforce confidence in predictions rather than turning it into a sort of regression problem.

Information

- **Shape**: `(5,976,479, 25)`
- **Columns**:

['user_id', 'book_id', 'rating', 'goodreads_book_id', 'best_book_id', 'work_id', 'books_count', 'isbn', 'isbn13', 'authors', 'original_publication_year', 'original_title', 'title', 'language_code', 'average_rating', 'ratings_count', 'work_ratings_count', 'work_text_reviews_count', 'ratings_1', 'ratings_2', 'ratings_3', 'ratings_4', 'ratings_5', 'image_url', 'small_image_url']

Example Row (via `__getitem__`)

```
{
    'user_id': np.int64(34197),
    'book_id': np.int64(155),
    'rating': np.int64(5),
    'categorical_genre': 'fantasy',
    'genre_idxs': 11305,
    'categorical_book_title': 'The Two Towers (The Lord of the Rings, #2)',
    'title_idx': 8779
}
```

```
drive.mount('/content/drive/')

#
# PyTorch Dataset
#

class BookRecommenderDataset(Dataset):
    def __init__(self, ratings_path, books_path, book_tags_path, tags_path):
        #
        # Raw datasets
        #
        self.reviews_df = pd.read_csv(ratings_path)
        self.books_df = pd.read_csv(books_path)
        self.book_tags_df = pd.read_csv(book_tags_path)
        self.tags_df = pd.read_csv(tags_path)

        #
        # Joining
        #
        reviews_books_df = pd.merge(self.reviews_df, self.books_df, on="book_id").reset_index(drop=True)
        tag_joined = pd.merge(self.book_tags_df, self.tags_df, on='tag_id')
        top_tags = tag_joined.sort_values(['goodreads_book_id', 'count'], ascending=[True, False])
        genre_per_goodreads_book_id = top_tags.groupby('goodreads_book_id')['tag_name'].apply(self.select_genres).reset_ind

        #
        # Master dataset
        #
        self.data = pd.merge(reviews_books_df, genre_per_goodreads_book_id, on='goodreads_book_id').iloc[:, [0, 1, 2, 25, 1

        #
        # Vocabularies
        #
        self.user_vocab = self.generate_vocab(self.data, 'user_id')
        self.book_vocab = self.generate_vocab(self.books_df, 'book_id')
        self.genre_vocab = self.generate_vocab(self.tags_df, 'tag_name')
        self.cross_vocab = self.generate_cross_vocab()

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        #
        # IDs
        #
        user_id = self.data.iloc[idx]['user_id']
        user_id_one_hot = self.one_hot(user_id, self.user_vocab)

        book_id = self.data.iloc[idx]['book_id']
        book_id_one_hot = self.one_hot(book_id, self.book_vocab)

        #
        # Genre
        #
        categorical_genre = self.data.iloc[idx]['tag_name']
        genre_idx = self.genre_vocab.get(categorical_genre)

        # Cross-features (user_id, book_id) combined
        # cross_one_hot = torch.zeros(len(self.cross_vocab))
        # cross_index = self.cross_vocab.get((user_id, book_id))
        # if cross_index is not None:
        #     cross_one_hot[cross_index] = 1.0

        #
        # Multi-hot vector for wide
        #
        wide_input = torch.cat([
            user_id_one_hot,
            book_id_one_hot,
            # cross_one_hot
        ], dim=-1)

        #
        # Label
        #
        rating = self.data.iloc[idx]['rating']
        rating = torch.tensor(1.0 if rating >= 4 else 0.0)
```

```
        return {
            'user_id': user_id,
            'book_id': book_id,
            'genre_idx': genre_idx,
            'rating' : rating,
            'wide_input': wide_input,
        }

    def __str__(self):
        print("Retrieving a single random instance.", end='\n\n')
        return str(self.__getitem__(735)) # favorite book <3

    def one_hot(self, id, vocab):
        a = torch.zeros(len(vocab))
        a[vocab[id]] = 1.0
        return a

    def head(self, n=5):
        return self.data.head(n)

    def generate_vocab(self, df, column_name):
        unique = set(df[column_name])
        vocab = {key: idx for idx, key in enumerate(sorted(unique))}
        vocab["<UNK>"] = len(vocab)
        return vocab

    def generate_cross_vocab(self):
        pairs = list(zip(self.data['user_id'], self.data['book_id']))

        unique_pairs = sorted(set(pairs))
        cross_vocab = {pair: idx for idx, pair in enumerate(unique_pairs)}
        return cross_vocab

    def select_genres(self, tags, n=1):
        l = tags[:5].tolist()

        for tag in tags:
            if tag not in {'to-read', 'favorites', 'currently-reading', 'books-i-own', 'book-club'}:
                return tag

#
# Creating the dataset and train/test split
#
dataset = BookRecommenderDataset(RATINGS_PATH, BOOKS_PATH, BOOK_TAGS_PATH, TAGS_PATH)
train_size = int(TRAIN_SIZE * len(dataset))
test_size = len(dataset) - train_size

#
# Custom collate fn to design batches
#
def custom_collate_fn(batch):
    batch_dict = {}
    batch_dict['user_id'] = torch.tensor([item['user_id'] for item in batch], dtype=torch.int64)
    batch_dict['book_id'] = torch.tensor([item['book_id'] for item in batch], dtype=torch.int64)
    batch_dict['genre_idx'] = torch.tensor([item['genre_idx'] for item in batch], dtype=torch.int64)
    batch_dict['rating'] = torch.tensor([item['rating'] for item in batch], dtype=torch.float64)
    wide_inputs = []
    for item in batch:
        wide_input = item['wide_input']
        if isinstance(wide_input, torch.Tensor):
            wide_inputs.append(wide_input)
        else:
            wide_inputs.append(torch.tensor(wide_input, dtype=torch.float32))
    batch_dict['wide_input'] = torch.stack(wide_inputs)
    return batch_dict

train_data, test_data = random_split(dataset, [train_size, test_size])
train_dataloader = DataLoader(train_data, batch_size=64, shuffle=True, collate_fn=custom_collate_fn)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True, collate_fn=custom_collate_fn)
```

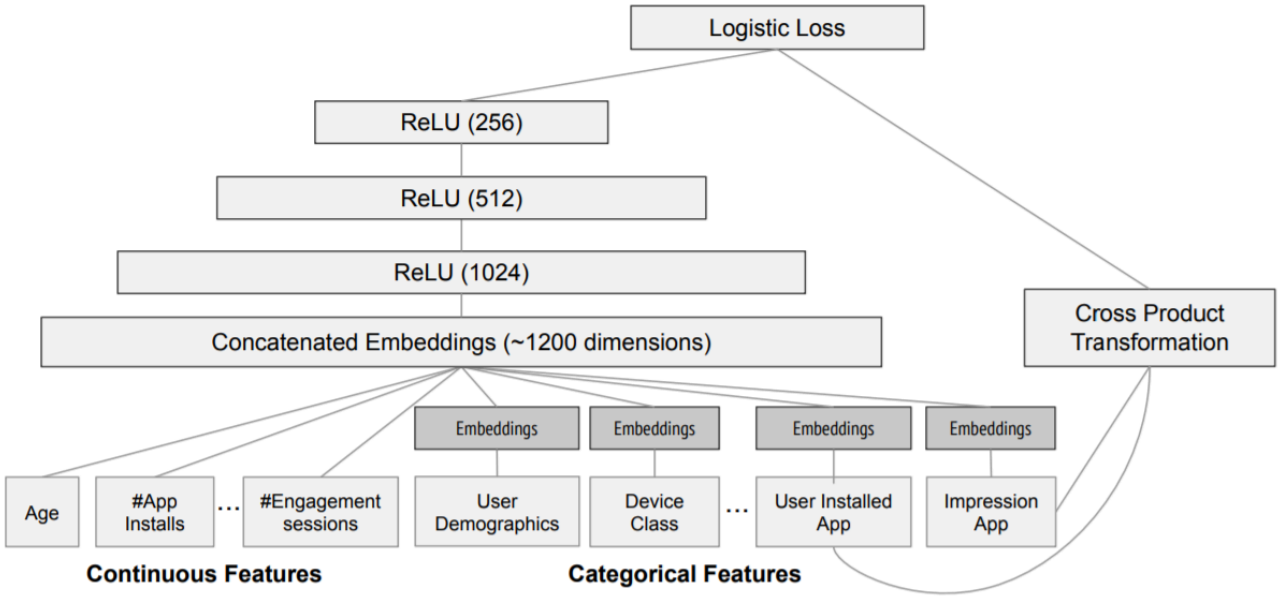
```
Drive already mounted at /content/drive/; to attempt to forcibly remount, call drive.mount("/content/drive/", force_remount=
```

```
print(dataset)
item = dataset.__getitem__(735)

```

```
Retrieving a single random instance.

{'user_id': np.int64(65), 'book_id': np.int64(155), 'genre_idx': 11305, 'rating': tensor(1.), 'wide_input': tensor([0., 0.,
```

Model Building

As a summary for those who have not read the paper, the ***Wide*** component uses a linear transformation using cross-products for better ***memorization*** of co-occurrence between feature items and target variables. Whereas the ***Deep*** component uses a deep neural network to learn deep connections between features to hopefully become more ***generalizable***, that is, it makes better recommendations for unseen examples. The deep component also utilizes embedding layers for categorical features, to learn dense representations of the otherwise sparse inputs.




```
USER_VOCAB_SIZE = len(dataset.user_vocab)
BOOK_VOCAB_SIZE = len(dataset.book_vocab)
GENRE_VOCAB_SIZE = len(dataset.genre_vocab)
CROSS_VOCAB_SIZE = len(dataset.cross_vocab)

USER_EMBEDDING_DIM = 32
BOOK_EMBEDDING_DIM = 32
GENRE_EMBEDDING_DIM = 20

class WideAndDeepModel(nn.Module):
    def __init__(self):
        super().__init__()

        #
        # Embedding layers
        #
        self.user_embedding = nn.Embedding(USER_VOCAB_SIZE, USER_EMBEDDING_DIM)
        self.book_embedding = nn.Embedding(BOOK_VOCAB_SIZE, BOOK_EMBEDDING_DIM)
        self.genre_embedding = nn.Embedding(GENRE_VOCAB_SIZE, GENRE_EMBEDDING_DIM)

        #
        # Learned parameters
        #
        self.deep_weight = nn.Parameter(torch.tensor(1.0))
        self.wide_weight = nn.Parameter(torch.tensor(1.0))

        #
        # Deep feed-forward layers
        #
        self.ffnn = nn.Sequential(
```

```
            nn.Linear(USER_EMBEDDING_DIM + BOOK_EMBEDDING_DIM + GENRE_EMBEDDING_DIM, 1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 1)
        )

        #
        # Wide Layers
        #
        self.wide = nn.Linear(USER_VOCAB_SIZE + BOOK_VOCAB_SIZE, 1)

    def forward(self, data):
        #
        # DEEP
        #
        embedded_user_ids = self.user_embedding(data['user_id']) # [B, E]
        embedded_book_ids = self.book_embedding(data['book_id']) # [B, E]
        embedded_genre = self.genre_embedding(data['genre_idx']) # [B, E]
        deep_input = torch.cat([embedded_user_ids, embedded_book_ids, embedded_genre], dim=1)
        deep_output = self.ffnn(deep_input)  # [64 predictions, 1]

        #
        # WIDE
        #
        wide_output = self.wide(data['wide_input'])  # [64 predictions, 1]

        #
        # Combine outputs (w/ learnable parameters)
        #
        logits = self.deep_weight * deep_output + self.wide_weight * wide_output
        return logits.view(-1) # [64,]
```

Training

Training Parameters & GPU

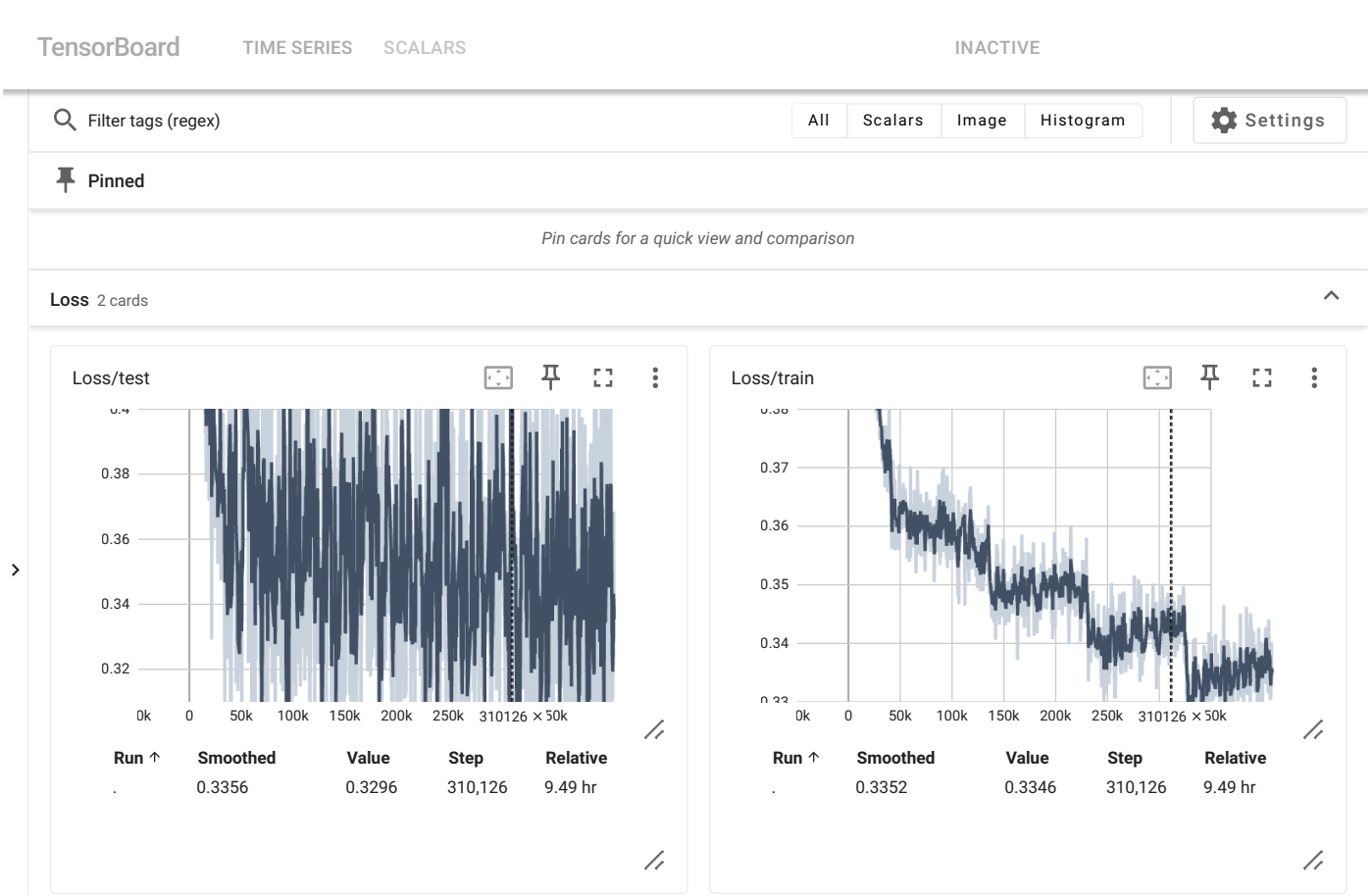
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

label_counts = Counter(dataset.data['rating'].apply(lambda x: 1.0 if x >= 4 else 0))
pos_weight = torch.tensor([label_counts[0] / label_counts[1]], device=device)

model = WideAndDeepModel()
loss_fn = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

writer = SummaryWriter('./logs/')
best_test_loss = float('inf')

SAVE_PATH = "/content/drive/MyDrive/book_recommender_best_model.pt"
EPOCHS = 5
GLOBAL_STEP = 0
```

Loop

```
#
# Training loop
#
for epoch in range(EPOCHS):
    running_loss = 0.0

    for batch_idx, data in enumerate(train_dataloader):
        model.train()
        optimizer.zero_grad()

        preds = model(data)  # [B]
        true_labels = data['rating']

        loss = loss_fn(preds, true_labels)
```

```
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        GLOBAL_STEP += 1

        if batch_idx % 100 == 99:
            avg_train_loss = running_loss / 100

            print(f"Epoch {epoch+1}, Batch {batch_idx+1}, Train Loss: {avg_train_loss:.4f}")

            writer.add_scalar("Loss/train", avg_train_loss, GLOBAL_STEP)

            #
            # Test set
            #

            model.eval()
            test_loss = 0.0
            total_test_batches = 0

            with torch.no_grad():
                test_data = next(iter(test_dataloader))

                test_preds = model(test_data)
                test_labels = test_data['rating']

                loss_val = loss_fn(test_preds, test_labels)
                test_loss += loss_val.item()
                total_test_batches += 1

            avg_test_loss = test_loss / total_test_batches

            writer.add_scalar("Loss/test", avg_test_loss, GLOBAL_STEP)

            #
            # Saving the best model based on test-loss (just a rough approx.)
            #
            if avg_test_loss < best_test_loss:
                best_test_loss = avg_test_loss
                torch.save(model.state_dict(), SAVE_PATH)

            running_loss = 0.0

writer.close()

```

```
%load_ext tensorboard
%tensorboard --logdir ./logs/
```



Calculating AUC

With only three features (user_id, book_id, genre), the model was able to achieve 0.779 AUC score. While it could theoretically be stronger than this, I am pleased to see that with such simple features, the wide and deep network was able to learn a relationship between these features and the rating one would give.

```
from sklearn.metrics import roc_auc_score

model.eval()
all_labels = []
all_preds = []

with torch.no_grad():
    for idx, batch in enumerate(test_dataloader):
        preds = model(batch)
        labels = batch['rating']

        probs = torch.sigmoid(preds).squeeze().cpu().numpy()

        all_preds.extend(probs)
        all_labels.extend(labels.cpu().numpy())

auc = roc_auc_score(all_labels, all_preds)
positive_rate = sum(all_labels) / len(all_labels)

print(f"AUC: {auc:.4f}")
print(f"Positive rate: {positive_rate:.4f}")

```

```
AUC: 0.7789
Positive rate: 0.6898
```

```
# import json
# with open("book_recommender_vocabs.json", "w") as f:
#     json.dump({"user_vocab": dataset.user_vocab,
#                "book_vocab": dataset.book_vocab,
#                "genre_vocab": dataset.genre_vocab}, f)
```

Conclusion