Creating a Hybrid Content-Collaborative Movie Recommender Using Deep Learning

Adam Lineberry Follow
Sep 10, 2018 · 11 min read

Written by Adam Lineberry and Claire Longo

Introduction

In this post we'll describe how we used deep learning models to create a hybrid recommender system that leverages both content and collaborative data. This approach tackles the content and collaborative

data separately at first, then combines the efforts to produce a system with the best of both worlds.

Using the MovieLens 20M Dataset, we developed an item-to-item (movie-to-movie) recommender system that recommends movies similar to a given input movie. To create the hybrid model, we ensembled the results of an autoencoder which learns content-based movie embeddings from tag data, and a deep entity embedding neural network which learns collaborative-based movie embeddings from ratings data.

We provide a brief summary of content and collaborative recommender systems, and discuss the benefits of a hybrid model. We'll be tracking how the different systems perform on one of our favorite movies: The Lord of the Rings: The Fellowship of the Ring.

The deep learning work was performed on a Paperspace GPU machine using PyTorch 0.4.1. The code for this project can be found in this GitHub repository.

Collaborative vs. Content Recommender Systems

Collaborative recommenders rely on data generated by users as they interact with items. Examples of this include:

- Users rating movies on a scale of 1–5
- Users purchasing or even viewing items on an online retail site
- Users reacting with "thumbs up" or "thumbs down" to songs on an online music streaming service
- Swiping left or right on a dating site

In the context of a movie recommender, collaborative filters find trends in how similar users rate movies based on rating profiles. The ratings data can be decomposed or otherwise processed using a variety of techniques to ultimately find user and movie embeddings in a shared latent space. The movie embeddings, which describe their location in the latent space, can then be used to make movie-to-movie recommendations.

One benefit of collaborative data is that it is always "self-generating"—users create the data for you naturally as they interact with items. This can be a valuable data source, especially in cases where high-quality item features are not available or difficult to obtain. Another benefit of collaborative filters is that it helps users discover new items that are outside the subspace defined by their historical profile.

However, there are some drawbacks to collaborative filters such as the well-known cold start problem. It is also difficult for collaborative filters to accurately recommend novel or niche items because these items typically do not have enough user-item interaction data.

Content recommenders rely on item features to make recommendations. Examples of this include:

- User generated tags on movies
- Item color
- Text description or user review of an item

Content filters tend to be more robust against popularity bias and the cold start problem. They can easily recommend new or novel items based on niche tastes. However, in an item-to-item recommender, content filters can only recommend items with features similar to the original item. This limits the scope of recommendations, and can also result in surfacing items with low ratings.

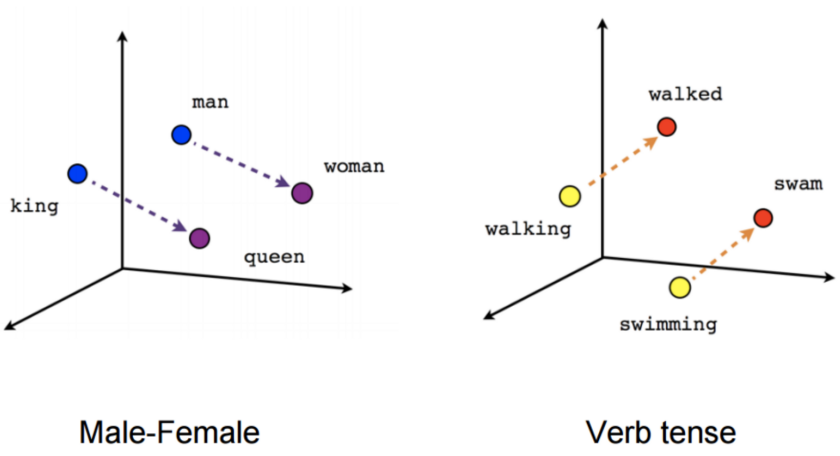
In the context of a movie-to-movie recommender, a collaborative filter answers the question: "What movies have a similar user-rating profile?", and a content filter answers the question: "What movies have similar features?". By creating a hybrid recommender we've attempted to create a system that recommends movies that other users rated in a similar manner, while still making on-topic recommendations based on the features of that movie.

What are Embeddings?

Embeddings are a popular topic in several areas of machine learning, such as natural language processing, predictive models with categorical features and recommender systems. There are many ways to compute embeddings, but the end goal is to map "things" to a latent space with complex and meaningful dimensions. In a movie

recommender, you can imagine latent dimensions measuring genres such as “sci-fi” and “romance” and other concepts such as “dialogue-driven versus action-packed”. Users and movies are mapped to this space by how strongly associated they are with each of the dimensions.

In the famous word2vec example, learned word embeddings were able to complete the analogy “man is to woman as king is to ___.” As seen in the [very simplified] figure below, the words have been mapped into a shared latent space such that the meaning of the word is represented geometrically.



Visualize Word Vectors (https://www.tensorflow.org/images/linear-relationships.png)

There are cases in machine learning and deep learning where one can choose between using a one-hot encoding (OHE) or a learned embedding to perform a particular task. OHE representations of data have the potentially undesirable trait that every item is orthogonal (therefore quantitatively entirely dissimilar) to all other items. When thinking of words as an example, this can be a weak representation of the data because the similarity/exchangeability of similar words such as “alien” and “extraterrestrial” is completely lost due to the orthogonality. For this reason, the use of word embeddings can extend the capabilities of some models.

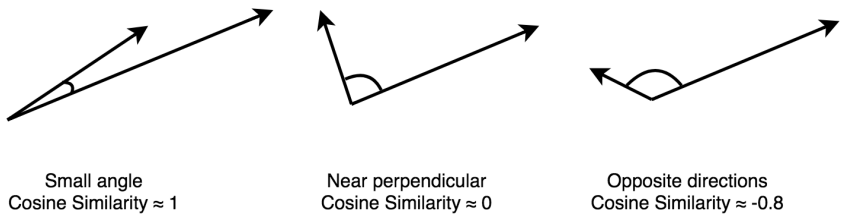
Cosine Similarity

We used cosine similarity to quantify the similarities between movies. Cosine similarity ranges from -1 to 1 and is calculated as the dot product between two vectors divided by their magnitudes.

$$sim(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

Equation for calculating cosine similarity between two vectors

In a nutshell, movie embedding vectors pointing in the same direction will receive high cosine similarity scores. The idea is that directions through the latent concept space capture the essence of movies. A simple example to help visualize this: if "sci-fi" and "romance" are dimensions in the latent space, then movies with similar ratios of sci-fi-ness to romance-ness will point in the same direction and thus receive high cosine similarity scores.



2D Visualization of vectors arranged in different ways and the resulting cosine similarities

This code constructs the cosine similarity matrix from the movie embeddings and outputs the top n most similar movies for a given input movie.

```
from sklearn.metrics.pairwise import cosine_similarity
import pandas as pd

class SimilarityPredictions(object):
    '''This class calculates a similarity matrix from
    Input: embeddings - a pandas dataframe of items an
    '''

    def __init__(self, embeddings):
        self.embeddings = embeddings
        self.ids = embeddings.index.tolist()

    def calculate_cosine_similarity_matrix(self):
        '''Calculates a cosine similarity matrix from
        similarity_matrix = pd.DataFrame(cosine_simila
            X=self.embeddings),
            index=self.ids)
        similarity_matrix.columns = self.ids
        return similarity_matrix
```

Finding Movie Embeddings from Content Data

Included in the MovieLens data is a set of around 500k user-generated movie tags. According to the MovieLens README: “Each tag is typically a single word or short phrase. The meaning, value, and purpose of a particular tag is determined by each user.”

This data is grouped by movie and the tags are concatenated to produce a corpus of documents. In this corpus, documents are the combination of all tags for a particular movie. An excerpt of the tag document for “Lord of the Rings: The Fellowship of the Ring” is shown below. As you can see, words/phrases like “adventure”, “fantasy” and “based on a book” appear frequently. This data also includes the names of actors and the book author.

```
adventure characters epic fantasy world fighting photography
Action adventure atmospheric based on a book based on book
beautifully filmed ensemble cast fantasy fantasy world high
fantasy imdb top 250 magic music nature nothing at all Oscar
(Best Cinematography) Oscar (Best Effects - Visual Effects)
```

```
scenic stylized Tolkien wizards adventure atmospheric
ensemble cast fantasy fantasy world magic stylized wizards
Watched adapted from:book author:J. R. R. Tolkein based on
book epic fantasy middle earth faithful to book fantasy good
versus evil high fantasy joseph campbell's study of
mythology influenced magic atmospheric boring high fantasy
Action adventure atmospheric based on a book beautifully
filmed fantasy high fantasy magic music mythology romance
stylized time travel Viggo Mortensen wizards Peter Jackson
Peter Jackson music must see Tolkien high fantasy Myth
Tolkien wizards Ian McKellen bast background universe
```

Next, we transformed the tag documents into a Term Frequency—Inverse Document Frequency (TF-IDF) representation. TF-IDF tokenizes unstructured text data into numeric features that can be more easily processed by machine learning algorithms. Roughly speaking, the frequency of each term in a document is scaled by the number of documents containing that term. As a result, words that differentiate movies (e.g., “alien”, “fantasy”) will be weighted higher than words used to describe all movies (e.g. “movie”, “cast”).

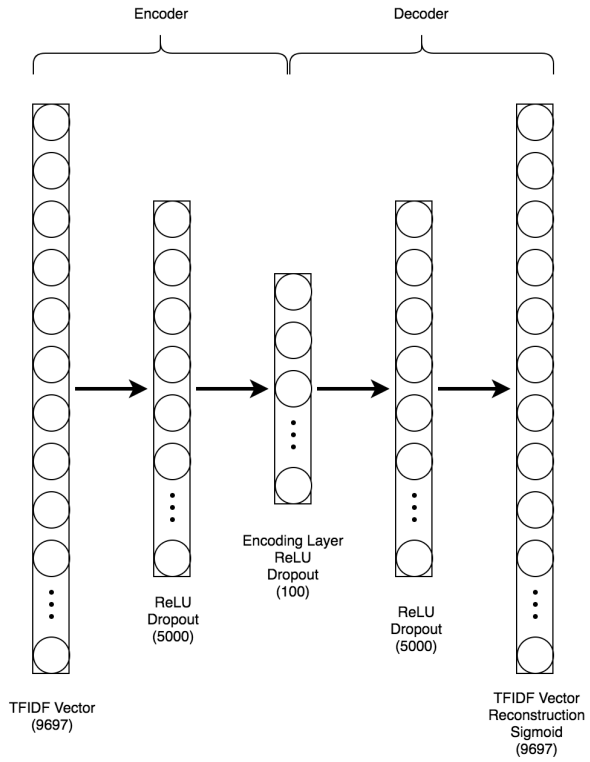
The TF-IDF vectorizer in sci-kit learn allows you to select the range of ngrams to consider. For this project, we found that unigrams were sufficient while still keeping our number of features manageable.

```
1  from sklearn.feature_extraction.text import TfidfVector
2  tfidf = TfidfVectorizer(ngram_range=(1, 1), min_df=0.00
3  tfidf_matrix = tfidf.fit_transform(tags['document'])
```

In TF-IDF space, each dimension represents how important a certain word is to a movie. This representation is less than ideal because the encoding of a single concept (e.g., alien/extraterrestrial) is fragmented and scattered across multiple dimensions. We would like to compress the TF-IDF data into a lower dimensional space where concepts are consolidated into shared dimensions. In the compressed space, the hope is that each dimension will represent complex and robust concepts.

To perform the compression we chose to use an autoencoder. Autoencoders are neural networks where the outputs are identical to the inputs. In our architecture (shown in the figure below), the high-dimensional TF-IDF input is gradually compressed into a 100 dimensional central hidden layer. This first half of the network is the “encoder”. The second half of the network, the “decoder”, attempts to

reconstruct the original input. By setting a mean-squared error loss function and performing backpropagation, the network (particularly the encoder) learns a function which maps data into lower dimensional space in such a way that most of the information is preserved.



Autoencoder Architecture

Let's start off by defining the encoder and decoder networks. They are defined separately to make it easy to encode the data once the network is trained.

```
class Encoder(nn.Module):
    def __init__(self, input_size, intermediate_size,
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_size, intermediate_size),
            nn.BatchNorm1d(intermediate_size),
            nn.ReLU(True),
            nn.Dropout(0.2),
            nn.Linear(intermediate_size, encoding_size
            nn.BatchNorm1d(encoding_size),
            nn.ReLU(True),
            nn.Dropout(0.2))

    def forward(self, x):
        x = self.encoder(x)
        return x


class Decoder(nn.Module):
    def __init__(self, output_size, intermediate_size,
        super().__init__()
        self.decoder = nn.Sequential(
```

Next, let's define a PyTorch Dataset class. Notice in the `__getitem__` method that `x` and `y` are equivalent—this is the fundamental concept of an autoencoder.

```
class AETrainingData(Dataset):
    """
    Format the training dataset to be input into the a
    Takes in dataframe and converts it to a PyTorch Te
    """

    def __init__(self, x_train):
        self.x = x_train

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

    def __getitem__(self, idx):
        """
        Returns a example from the data set as a pytor
        """
        # Get example/target pair at idx as numpy arra
```

With these building blocks in-hand, let's define a wrapper class that instantiates everything, handles the training loop, and performs the data encoding.

```
class AutoEncoder(object):

    def __init__(self, data, validation_perc=0.2, lr=0
                 intermediate_size=1000, encoded_size=

        # create training dataloader and validation te
        self.data = data
        self.val_idxs = get_cv_idxs(n=data.shape[0], v
        [(self.val, self.train)] = split_by_idx(self.v
        self.dataset = AETrainingData(self.train)
        self.dataloader = DataLoader(self.dataset, bat
                                     num_workers=multi
        self.val = torch.from_numpy(self.val.values).\
            type(torch.FloatTensor).cuda()

        # instantiate the encoder and decoder nets
        size = data.shape[1]
        self.encoder = Encoder(size, intermediate_size
        self.decoder = Decoder(size, intermediate_size

        # instantiate the optimizers
        self.encoder_optimizer = optim.Adam(
            self.encoder.parameters(), lr=lr, weight_d
        self.decoder_optimizer = optim.Adam(
            self.decoder.parameters(), lr=lr, weight_d

        # instantiate the loss criterion
        self.criterion = nn.MSELoss(reduction='element

        self.train_losses = []
        self.val_losses = []

    def train_step(self, input_tensor, target_tensor):
        # clear the gradients in the optimizers
        self.encoder_optimizer.zero_grad()
        self.decoder_optimizer.zero_grad()

        # Forward pass through
        encoded_representation = self.encoder(input_te
        reconstruction = self.decoder(encoded_represen

        # Compute the loss
        loss = self.criterion(reconstruction, target_t

        # Compute the gradients
```

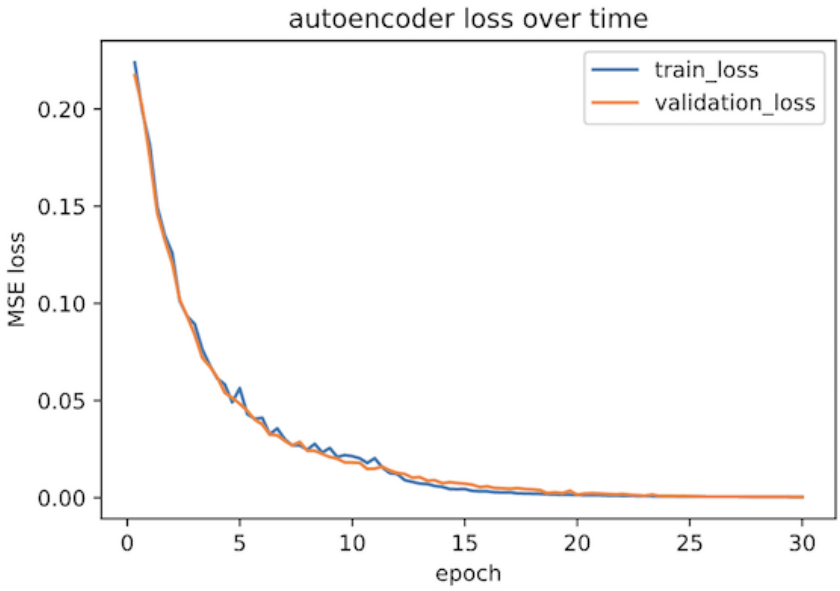
From here, it's straightforward to train the autoencoder and encode/compress the data:

```
# load tfidf matrix (pandas df) from pickle
path = "../../data/"
with open(f'{path}tfidf_matrix.pkl', 'rb') as fh:
    tfidf = pickle.load(fh)
print(tfidf.shape)  # (26744, 9697)

# instantiate the autoencoder object
ae = AutoEncoder(tfidf, validation_perc=0.1, lr=1e-3,

# train for 30 epochs
```

Below is a plot of the train and validation losses over time. As you can see, the model is not over- or under-fitting and eventually the losses converges to a very low value (on the order of 1e-4).



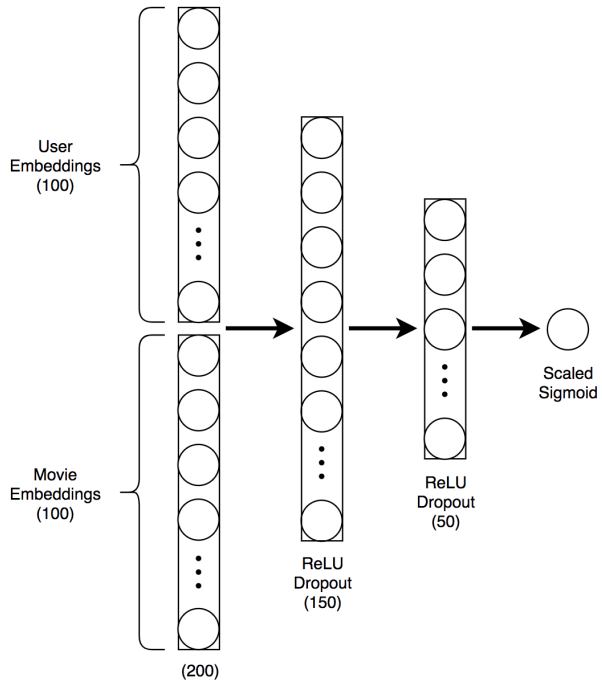
Using the content-based movie embeddings learned by the autoencoder, the top 20 most similar movies to "Lord of the Rings: The Fellowship of the Ring" as defined by cosine similarity are shown below.

Search this file...

	title
1	Lord of the Rings: The Two Towers, The (2002)
2	Chronicles of Narnia: The Lion, the Witch and the Wardrobe, The (200
3	Dark Crystal, The (1982)
4	Lord of the Rings: The Return of the King, The (2003)
5	Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Phil
6	Tales from Earthsea (Gedo Senki) (2006)
7	Almighty Thor (2011)
8	Chronicles of Narnia: Prince Caspian, The (2008)
9	Magic Voyage of Sindbad, The (Sadko) (1953)
10	Harry Potter and the Deathly Hallows: Part 2 (2011)
11	Conan the Destroyer (1984)
12	Spiderwick Chronicles, The (2008)

Finding Movie Embeddings from Collaborative Data

A fully connected neural network is used to find movie and user embeddings. In this architecture, a user embedding matrix of size `(n_users, n_factors)` and a movie embedding matrix of size `(n_movies, n_factors)` are randomly initialized and subsequently learned via gradient descent. Each training data point is a user index, movie index and a rating (on a scale of 1–5). The user and movie indices are used to lookup the embedding vectors (which are rows of the embedding matrices). These vectors are then concatenated and passed in as the input to the neural net.



Embedding Net Architecture

This is similar to the technique taught by Jeremy Howard in the fast.ai MOOC, and we used the fastai library as well. Let's read in the data, create a validation set, and construct a fastai ColumnarModelData object which is essentially a handy wrapper around PyTorch's Dataset and DataLoader classes.

```
from fastai.learner import *
from fastai.column_data import *

# load ratings data into pandas dataframe
path = "../data/ml-20m/"
ratings = pd.read_csv(path + "ratings.csv")

# get validation set indices
val_idxs = get_cv_idxs(n=len(ratings), val_pct=0.05)

# create user and movie indices
# (the id's are not sequential and therefore not valid
u_uniq = ratings.userId.unique()
user2idx = {o:i for i,o in enumerate(u_uniq)}
ratings.userId = ratings.userId.apply(lambda x: user2i
m_uniq = ratings.movieId.unique()
movie2idx = {o:i for i,o in enumerate(m_uniq)}
ratings.movieId = ratings.movieId.apply(lambda x: movi

# set number of latent dimensions (n_factors) to 100
n_users=int(ratings.userId.nunique())
```

Next, let's define the neural net. Since the output of the network is constrained to be within 1–5, a scaled sigmoid activation function is used at the output layer. Passing the linear activation though a sigmoid gives the model a bit more flexibility and thus makes it easier to train. For example, if the model is trying to output a 5, instead of forcing it to output a value very near 5 from the linear calculation, using a sigmoid allows it to output any high value (because inputs ~6 or greater all get mapped to ~1.0 by the sigmoid function). An alternative would be to stop short of the sigmoid and let the model learn to output the correct rating directly from the linear output.

```
def get_emb(ni,nf):
    e = nn.Embedding(ni, nf)
    e.weight.data.uniform_(-0.01,0.01)
    return e

class EmbeddingNet(nn.Module):
    def __init__(self, n_users, n_movies):
        super().__init__()
        (self.u, self.m) = [get_emb(*o) for o in [
            (n_users, n_factors), (n_movies, n_factors

        # layer 1 fully connected 150 units
        self.lin1 = nn.Linear(n_factors*2, 150)

        # layer 2 fully connected 50 units
        self.lin2 = nn.Linear(150, 50)

        # layer 3 fully connected 1 unit (output)
        self.lin3 = nn.Linear(50, 1)

        # dropouts
        self.drop1 = nn.Dropout(0.5)
        self.drop2 = nn.Dropout(0.4)
        self.drop3 = nn.Dropout(0.25)

    def forward(self, cats, conts):

        # extracting user and movie indices from Colum
        users,movies = cats[:,0],cats[:,1]

```

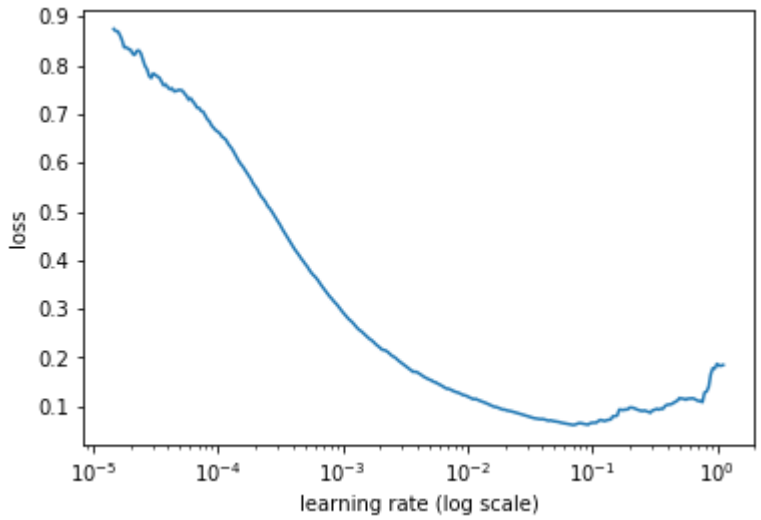
The fastai library makes it very simple to fit the model. The `fit` function used here takes care of the training loop and provides a nice animated status bar with time remaining and loss values for each minibatch.

```
# instantiate the model and send to gpu
model = EmbeddingNet(n_users, n_movies).cuda()
# instantiate Adam optimizer with weight decay
opt = optim.Adam(model.parameters(), lr=1e-3, weight_de
# fit model using MSE loss function
```

This model took a while to train due to the large size of the training data (20 million). At various points throughout training we adjusted the dropout levels to combat over- or under-fitting. The learning rate was also regularly tweaked using information from the learning rate finder tool included in the fastai library:

```
# wrap model in fastai's BasicModel class to gain
# access to the learning rate finder functionality
bm = BasicModel(model)
learner = StructuredLearner(data, bm)
learner.lr_find()
```

In the learning rate finder procedure, the learning rate is initially set to a very small value (1e-5) and is iteratively increased throughout the course of one epoch, up to a high ceiling value (10). At each stage, the loss is recorded so that a plot such as the one below may be produced. Interpreting this plot is a heuristic process that is explained in more detail in the fastai course, but the takeaway is this: select a learning rate where the loss is still decreasing quickly and has not leveled off yet. In this case, selecting 1e-2 would be a reasonable choice.



With a dataset of this size (and the associated long training times), it is likely that you will need the ability to persist and depersist your trained model such that you can continue training at a later time. Once finished, you will also need to save your learned embeddings to disk for post-processing. The following code will take care of these items.

```
# code to persist model parameters for future use
with open('../models/embedding_net_state_dict.pytorch'
    pickle.dump(model.state_dict(), fh)

# code to depersist model parameters and load them int
with open('../models/embedding_net_state_dict.pytorch'
    state_dict = pickle.load(fh)
model.load_state_dict(state_dict)

# extract embedding matrices and embed them for postpr
```

Once the net is satisfactorily trained, you will have robust movie and user embeddings ready to use for a variety of practical tasks. Using the collaborative-based movie embeddings learned by this neural net, the top 20 most similar movies to "Lord of the Rings: The Fellowship of the Ring" as defined by cosine similarity are shown below.

Search this file...

title
Star Wars: Episode IV - A New Hope (1977)
Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark
Lord of the Rings: The Return of the King, The (2003)
Star Wars: Episode V - The Empire Strikes Back (1980)
Schindler's List (1993)
Lord of the Rings: The Two Towers, The (2002)
Shawshank Redemption, The (1994)
Saving Private Ryan (1998)
Good Will Hunting (1997)
Matrix, The (1999)
Inception (2010)
Dark Knight, The (2008)

Bringing it all Together

As seen above, the collaborative recommendations for Lord of the Rings appear to be primarily popular and highly rated blockbuster movies with strong action and adventure themes. The content recommendations appear to be less influenced by popularity and may include some hidden gems in the fantasy genre.

To ensemble the results from the content and the collaborative models we simply averaged the cosine similarities. By ensembling collaborative and content-based results we are able to make recommendations that can hopefully draw from the strengths of both methods. The following are the ensembled recommendations for Lord of the Rings.

Search this file...

	title
1	Lord of the Rings: The Two Towers, The (2002)
2	Lord of the Rings: The Return of the King, The (2003)
3	Star Wars: Episode V - The Empire Strikes Back (1980)
4	Star Wars: Episode IV - A New Hope (1977)
5	Princess Bride, The (1987)
6	Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark
7	Star Wars: Episode VI - Return of the Jedi (1983)
8	WALL·E (2008)
9	Batman Begins (2005)
10	Harry Potter and the Deathly Hallows: Part 2 (2011)
11	Untouchables, The (1987)
12	How to Train Your Dragon (2010)

Conclusion

In this post we developed a movie-to-movie hybrid content-collaborative recommender system. We discussed and illustrated the pros and cons of content and collaborative-based methods. We also showed how to develop recommender systems using deep learning instead of traditional matrix factorization methods.

In our observations the collaborative filter was good at spanning gaps across genres and consistently recommending movies with high ratings. The content filter was good at identifying very similar styles (e.g., magical realms set in a medieval time) and lesser-viewed, potentially hidden gem movies. The aim of the hybrid approach is to combine the strengths of both systems.

We also found some movies we want to see! If you would like to see recommendations for a particular movie of your choice, feel free to reach out on Twitter, LinkedIn or email.

Adam Lineberry

Claire Longo

Adam is a data scientist in Denver, CO with interests in deep learning, big data, Formula 1 racing and skiing.

Claire is a data scientist in Denver, CO working in the fashion industry. Her research interests are in recommender systems.