UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUIZ FERNANDO DA SILVA

EXTENSÃO DA API DEEP4REC:
TRABALHO DE CONCLUSÃO DE CURSO

CAMPINA GRANDE - PB

2019

LUIZ FERNANDO DA SILVA

EXTENSÃO DA API DEEP4REC: TRABALHO DE CONCLUSÃO DE CURSO

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Leandro Balby Marinho.

CAMPINA GRANDE - PB

2019

S586e Silva, Luiz Fernando da.
Extensão de API Deep4Rec: trabalho de conclusão de curso. / Luiz Fernando da Silva. – 2019.

9 f.

Orientador: Prof. Dr. Leandro Balby Marinho.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Application Programming Interface - API. 2. API Deep4Rec. 3. Sistemas de recomendação. 4. Deep learning. 5. Tensorflow. I. Marinho, Leandro Balby. II. Título.

CDU:004.6(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

LUIZ FERNANDO DA SILVA

EXTENSÃO DA API DEEP4REC: TRABALHO DE CONCLUSÃO DE CURSO

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Leandro Balby Marinho
Orientador – UASC/CEEI/UFCG

Professora Dra. Raquel Vigolvino Lopes
Examinadora – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de novembro de 2019.

CAMPINA GRANDE - PB

Extensão da API Deep4Rec: Trabalho de Conclusão de Curso

Luiz Fernando da Silva
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
luiz.silva@ccc.ufcg.edu.br

RESUMO

Sistemas de recomendação tem se tornado cada vez mais populares em diversos domínios, como música, filmes e séries. O estudo da aplicação de técnicas de deep learning para melhorar a performance desses sistemas ainda é recente e tem bastante obstáculos a serem superados. Neste trabalho, proponho estender a API Deep4Rec, uma API para sistemas de recomendação baseados em algoritmos e técnicas de Deep Learning. Essa API usa um dos frameworks mais populares de Deep Learning, o Tensorflow do Google. De forma mais específica, eu adicionei novas features de metadados ao dataset MovieLens para serem utilizadas nos algoritmos e acrescentei o algoritmo *Spectrall Collaborative Filtering*. Os algoritmos implementados na API podem ser utilizados por pesquisadores e desenvolvedores na área de sistemas de recomendação.

Keywords

Deep Learning, Tensorflow, API.

Repositório com artefatos e solução:

https://github.com/Luiz-FS/Deep4Rec

1. INTRODUÇÃO

Com a grande quantidade de informações e escolhas que os usuários se deparam nos diversos aplicativos online que usam diariamente, sistemas de recomendação aparecem como um serviço importante. De acordo com uma publicação feita em 2018 na revista forbes [8], diariamente são gerados 2,5 quintilhões de bytes. Nesse contexto o uso dos sistemas de recomendação tem se tornado indispensável para indexar e filtrar os dados de modo a prover uma melhor experiência para os usuário ao buscar por algum conteúdo, já que por meio desses sistemas será exibido apenas aquilo que for mais relevante para a pesquisa.

Esses sistemas são utilizado nas mais diversas aplicações, processando os dados produzidos pelos usuários (tais como, notas atribuídas a filmes, músicas, textos, etc) para gerar recomendações de conteúdos que possam ser de interesse dos mesmos. Exemplo, em uma plataforma de streaming de filmes e séries, o sistema de recomendação irá processar os dados dos conteúdos já assistidos pelos usuários e a as notas atribuídas a eles, de modo determinar os usuários que têm perfis semelhantes (Essa abordagem chama-se filtragem colaborativa) e com base nisso, fazer recomendações. Nesse contexto ele irá exibir para esses usuários, filmes e séries que são relevantes e que provavelmente irão satisfazer suas preferências com base no que ele já assistiu e no que outros usuários com perfis semelhantes assistiram e gostaram.

Nos últimos anos despertou-se um grande interesse no uso de deep learning em sistemas de recomendação. Segundo o *paper Deep Learning based Recommender System: A Survey and New Perspectives* [1], recentemente muitas indústrias têm usado *deep learnig* para melhorar ainda mais a qualidade de suas recomendações. *Deep learning* é um ramo da machine learning que usa redes neurais para modelar de forma não linear as características implícitas dos dados. Desse modo torna-se possível aproveitar melhor todas as características dos dados provendo uma melhor eficácia ao fazer recomendações para o usuário.

Embora existam vários algoritmos implementados, ainda há uma certa dificuldade para utilizá-los nas mais diversas aplicações. Já que muitos desses algoritmos não tem implementação pública, e os que têm, podem estar em frameworks e linguagens diferentes, além de não ter garantia de legibilidade e boa documentação.

A API Deep4Rec foi construída para resolver alguns desses problemas, mas ainda está em versão preliminar. Desse modo, o objetivo deste trabalho é estender a API Deep4Rec adicionando novas features de metadados aos datasets e implementando novos algoritmos de *deep learning* para sistemas de recomendação.[1]

2. SOLUÇÃO

2.1 Descrição

A Deep4Rec é uma API de deep learning para sistemas de recomendação que começou a ser desenvolvida pela ex aluna da UFCG (Universidade Federal de Campina Grande) Mariane Linhares como um projeto da disciplina Recuperação de Informação e Busca na Web. Essa API fornece um conjunto de algoritmos de deep learning e datasets prontos para serem utilizados de forma fácil e rápida. Com ela é possível utilizar alguns dos mais recentes e modernos algoritmos para sistemas de

recomendação e os mais populares datasets em um só local. Sendo necessário apenas escolher o modelo e o dataset para treiná-lo. A API foi desenvolvida usando *TensorFlow [3],* que é um dos frameworks mais populares de *Deep Learning* com ampla documentação e comunidade bastante ativa.

A API ainda é preliminar e está em fase de desenvolvimento. Dentre os algoritmos já implementados tem-se o Neural Matrix Factorization [7], Factorization Machine [4], Neural Machine Factorization [5] e Wide&Deep [6]. Já as base de dados presentes na API são: Movielens 100k, 1m e 20m [13], Frappe [11] e Census [12].

Atualmente as implementações consideram como dados de entrada apenas os usuários e itens, mas muitos desses algoritmos são equipados para lidar também com metadados de usuários e itens. Como extensão acrescentarei mais metadados de gênero dos filmes para melhorar a qualidade das recomendações após o treino dos modelos. Além disso, incrementarei o algoritmo *Spectral Collaborative Filtering [2]* que foi abordado no artigo *Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches [9]* que ganhou o prêmio de *best long paper* no RecSys [10] 2019 que é o principal evento internacional para apresentação de resultados de pesquisa, sistemas e técnicas na área de sistemas de recomendação.

2.2 Arquitetura

O projeto consiste em uma API para python que pode ser instalado via *package manager*, como o pip. Sua arquitetura é dividida em duas superclasses (Model e Dataset) e várias subclasses que são as implementações de cada algoritmo e dataset (ver o diagrama da figura 1). Essas superclasses são responsáveis por conter os tratamentos genéricos que são necessários para todos os modelos e datasets.

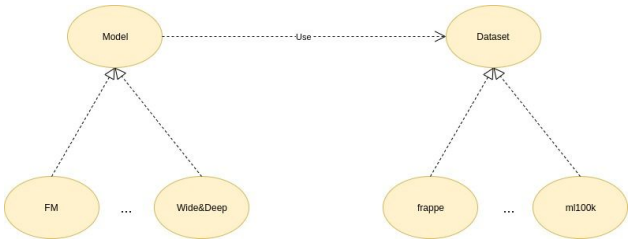

Figura 1: Diagrama de classes da API, dividido em duas superclasses, Model e Dataset suas subclasses que são os algoritmo.

A classe Model contém o código para inicializar o modelos bem como treiná-los. Os algoritmos que herdam da classe Model devem ter implementados pelo menos duas funções, o *__init__* do objeto e a função call que vai ser responsável por fazer as predições do algoritmo.

A classe Dataset contém o código para o tratamento dos datasets implementados, nela há métodos para fazer download dos dados, tratar e dividi-los em treino, teste e validação. Toda a manipulação dos dados necessárias para a utilização dos algoritmos é feita nela. As implementações de dataset que herdam da classe precisam determinar qual a url de download dos dados, quantas features serão utilizadas, bem como os tratamentos específicos necessários de cada base de dados.

2.3 Tecnologías utilizadas

Para o desenvolvimento da API foi utilizado a linguagem de programação python junto com a biblioteca TensorFlow [3] versão 1.x.x. Python é uma linguagem de programação robusta, fácil de ser aprendida e tem sido muito utilizada na área de ciência de dados para a construção e testes dos algoritmos, justamente por ser simples de trabalhar com ela, além disso fornece abstrações que são convenientes para acoplar os algoritmos. O Tensorflow [3] é uma biblioteca de código aberto em python para deep learning, ele fornece uma série de ferramentas, modelos e algoritmos que podem ser utilizados. Com ele é possível desenvolver algoritmos de forma mais simples, pois ele fornece uma abstração que permite que os desenvolvedores não se preocupem com os detalhes básicos do algoritmo e tente se focar na lógica, programando de forma mais declarativa.

O Tensorflow [3] permite a criação de gráficos de fluxos de dados que são estruturas que descrevem os processamentos feitos, onde cada nó representa uma operação matemática. Ele também permite uma execução mais rápida dos algoritmos pois sua implementação foi feita em C++ e apenas o front-end foi feito em python para uma maior facilidade no uso.

2.4 Estrutura do projeto

O projeto está estruturado em dois diretórios principais, datasets e *models*, que contém as bases de dados e algoritmos de recomendação respectivamente. Pode-se observar a organização dos diretórios na figura 2.

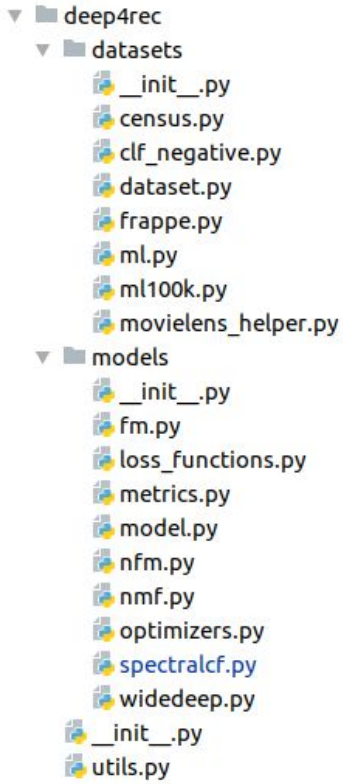

Figura 2: Estrutura de diretórios.

A API foi estruturada dessa forma para facilitar seu entendimento e implantação de novos algoritmos, pois assim ela opera da mesma forma como um sistema de recomendação funciona,

recebendo uma conjunto de dados de usuários e itens (*datasets*) para treinar os modelos (*models*) e gerar as recomendações, como mostrado na figura 3.

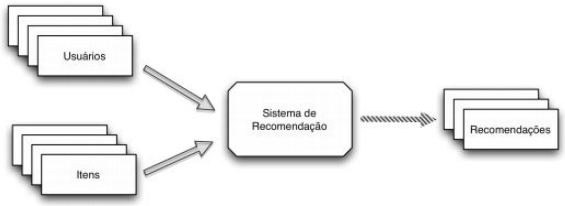

Figura 3: Processo de recomendação. https://www.maxwell.vrac.puc-rio.br/19273/19273_3.PDF.

Para fazer uso da API é preciso tê-la instalada via package manager (ainda não disponível na versão preliminar), ou baixar o código fonte. Após isso, é necessário importar os módulos de modelos e datasets (ver a figura 4).

```
from deep4rec import datasets
from deep4rec import models
```

Figura 4: Importando modulos da API Deep4Rec.

Os datasets são responsáveis por fazer o download, tratar e separar os dados em treino, teste, e validação. Ao importar o pacote datasets é possível utilizar a função datasets.*options() (figura 5)* para visualizar quais são os datasets disponíveis para uso. Após a escolha de qual banco de dados utilizar, basta chamar a função *datasets.build_dataset("NomeDoBancoDeDados") (figura 6).* O dataset gerado já contém os dados baixados e pré processados, sendo possível passá-los como parâmetro para os modelos para realizar o treinamento. Ele contém métodos para pegar o número de features, separar os dados e gerar um dataset do Tensorflow [3].

```
datasets.options()
```

Figura 5: Listando base de dados disponíveis.

```
ds = datasets.build_dataset("ml-100k")
```

Figura 6: Gerando um dataset com uma das base de dados implementadas.

Os models são responsáveis pelo processamentos dos dados usando algoritmos de deep learning para gerar predições de dados nunca antes vistos com base no treino feito com os dados de treinamento. Ao importar o pacote models é possível visualizar quais modelos já estão implementados usando a *função models.options()*. Para utilizar algum dos modelos listados é preciso instanciá-los da seguinte forma: models.NomeDoModelo(datset, **outros_parametros) (figura 7), após isso será gerado um modelo pronto para o treinamento. Para iniciar o treino do algoritmo basta chamar a função train (figura 8) com os parâmetros necessários para o algoritmo utilizado.

Toda os parâmetros utilizados tanto nos datasets quanto nos *models* podem ser consultados na documentação da API que está no próprio código fonte.

```
spfc = models.SpectralCF(ds)
```

Figura 7: Gerando uma instância do modelo especificado.

```
spfc.train(
    ds,
    batch_size=1024,
    epochs=200,
    loss_function="rmse",
    eval_metrics=['precision', 'recall']
)
```

Figura 8: Treinando modelo.

Após a finalização do treino do modelo, para gerar as predições utiliza-se a função predict que recebe a lista de usuários para os quais deverão ser geradas as predições. No exemplo da figura 9 é chamado o predict para os usuários 1 e 2. Essa função irá retornar uma lista das notas estimadas dos usuário passados como parâmetro em relação a todos os itens disponíveis.

Para gerar uma recomendação a partir das predições, é necessário ordenar a lista de itens e suas respectivas notas estimadas em ordem decrescente e pegar os top k itens que possuem maior nota, onde o k é a quantidade de itens que se deseja recomendar.

```
spfc.predict([1,2])
```

Figura 9: Gerando predições para os usuários 1 e 2.

2.5 Contribuições realizadas

2.5.1 Adição de novas features de metadados aos datasets

As *features* extraídas dos dados são de grande importância para o treinamento dos algoritmos de sistemas de recomendação, dessa forma eles podem extrair relações entre elas para prever os resultados. Muitos algoritmos possuem suporte para mais metadados dos usuários e itens. A adição de metadados de gêneros dos filmes, permite que os modelos possam criar relações entre usuários que assistem filmes de gêneros semelhantes, por exemplo, um usuário A gosta de filmes de romance e assistiu um filme Y e deu uma boa nota, então se um usuário B também gosta de romance é provável que ele também irá gostar do filme Y dado que A têm um perfil semelhante e gostou.

Para adicionar novos metadados as features já existentes é preciso codificar-los em dados numéricos (caso ainda não sejam) e adicioná-los à lista de dados, como no exemplo da figura 11. Mais especificamente no dataset MovieLens, foi necessário mapear um arquivo de texto contendo os ids dos filmes e os gêneros para um dicionário de python para que o mesmo possa ser utilizado no método que cria o dataset com os usuários e itens. Como podemos observar na figura 10, é feita a leitura do arquivo "u.item" e posteriormente cada linha do arquivo é mapeada para o dicionário *item_genres_index,* que será utilizado para adicionar os gêneros as features já utilizadas, como está representado na figura 11.

```python
def _load_genres_data(self):
    filepath = os.path.join(self.preprocessed_path, "u.item")
    with open(filepath, encoding="ISO-8859-1") as f:
        for line in f:
            item_data = line.split("|")
            item_id = item_data[0]
            item_genres = list(map(int, item_data[5:]))
            self.item_genres_index[item_id] = item_genres
```

Figura 10: Criando mapa de itens e gêneros.

```python
data.append(
    [
        self.user_index[user_id],
        self.item_index[movie_id],
        *self.item_genres_index[movie_id],
    ]
)
```

Figura 11: Adicionando os gêneros as features existentes.

As figuras 12 e 13 mostram o gráfico de erros do algoritmo *Factorization Machine* [4] durante o treino, o eixo x indica as épocas do treinamento e o eixo y indica o valor do erro calculado em determinada época. Quanto menor for o erro, melhor será os resultados das recomendações. Para a análise, o algoritmo foi treinado em 10 épocas (Cada época representa um passo do treino) e a função de custo utilizada foi o RMSE (Root Mean Square Error). O RMSE mede a raiz da distância média quadrada entre o valor predito pelo modelo e o valor real.

Como pode-se observar a adição das features de gênero teve um impacto positivo no treinamento do modelo. Ao final do treino, o RMSE do conjunto de teste do modelo sem as features ficou em cerca de 1.1, já do modelo com as features ficou abaixo de 1.

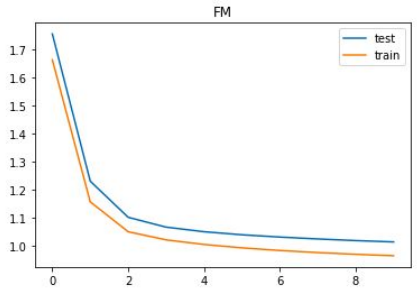

Figura 12: Plot do rmse sem features de gênero.

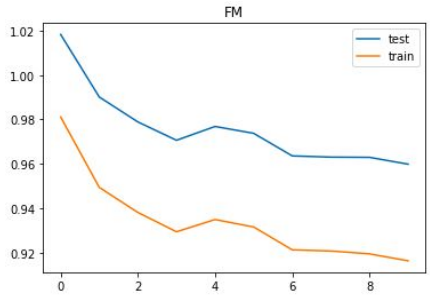

Figura 13: Plot do rmse com features de gênero.

2.5.2 Algoritmo Spectral Collaborative Filtering

O algoritmo *Spectral Collaborative Filtering [2]* foi criado para extrair conexões implícitas entre usuários e itens, reduzindo o problema do Cold-Start. O problema do *Cold-Start* é um dos grandes desafios dos sistemas de recomendação baseados em filtragem colaborativa. Ele ocorre quando um usuário interagiu com um número muito pequeno de itens. Portanto, o usuário compartilha poucos itens com outros usuários, reduzindo a eficiência dos sistemas de recomendação.

Modelos tradicionais como *Matrix Factorization* [7] são projetados para aproximar as conexões diretas ou próximas, mas conexões indiretas ocultas nas estruturas de grafo raramente são capturadas por eles. O SpectralCF se beneficia das informações das conexões extraídas do domínio espectral para descobrir as conexões ocultas entre usuários e itens. Dessa forma, diminui o problema *cold-start*.

O *Spectral Collaborative Filtering [2]* usa a teoria do grafo espectral (Teoria que estuda as propriedades de um grafo por meio de sua representação matricial, onde o espectro da matriz é o seu conjunto de autovalores) para modelar as conexões entre o grafo de conexões entre usuários e itens (Usuários e itens são vértices e as arestas são as interações entre eles, ou seja, se o usuário 1 interagiu com o item 2, haverá uma aresta ligando os dois. Ver a figura 14 para mais detalhes) e os autovalores da matriz de laplace gerada a partir do grafo.

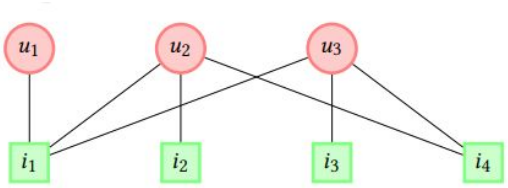

Figura 14: Grafo de conexão entre usuários e itens [2].

O algoritmo inicialmente cria uma matriz de interação R_{nxm} entre usuários e itens, onde n é o número de usuários e m o número de itens e os valores das células R_{ij} é igual a 1 se o i-ésimo usuário interagiu com o j-ésimo item e 0 caso contrário. A figura 15 mostra como foi implementado.

```python
def build_graph(self):
    graph = np.zeros((self.num_users, self.num_items), dtype=np.float32)

    for (user, items) in self.users_id_items_id.items():
        for item in items:
            graph[user, item] = 1

    return graph
```

Figura 15: Código para construir matriz de interação.

A partir da matriz de interação, constrói-se a matriz de adjacência que possui o seguinte formato:

$$A = \begin{bmatrix} 0 & R \\ R^T & 0 \end{bmatrix}$$

Onde $Anxn$ é a matriz de adjacência de ordem n que é a soma do número de usuários e itens e R^T é a matriz transposta da matriz R.

Posteriormente é construida a matriz de laplace, cuja fórmula é: $L = I - D^{-1} * A$ (ver a implementação na figura 16), onde I é a matriz identidade, D^{-1} é a inversa da matriz diagonal de graus da matriz de adjacência. Com a matriz de laplace criada, cacula-se os autovalores e os autovetores.

```
def laplacian_matrix(self, normalized=False):
    if not normalized:
        return self.D - self.A

    tmp = np.dot(np.diag(np.power(self.matrix_d, -1)), self.matrix_adj)
    return np.identity(self.num_users + self.num_items, dtype=np.float32) - tmp
```

Figura 16: Construção da Matriz de laplace.

Após o cálculo dos autovalores e autovetores aplica-se a fórmula da figura 17, cuja matriz U é a matriz de autovetores, ƛ é o vetor de autovalores, Θ são filtros aplicados a matriz [Xu Xi], **σ** é a função sigmóide aplicada ao modelo. A figura 18 mostra a implementação do código que gera as predições a partir da fórmula explicada anteriormente.

$$\begin{bmatrix} X^u_{new} \\ X^i_{new} \end{bmatrix} = \sigma\left((UU^\top + U\Lambda U^\top)\begin{bmatrix} X^u \\ X^i \end{bmatrix}\Theta'\right),$$

Figura 17: Fórmula do modelo Spectral Collaborative Filtering [2].

```
def predict(self, users):
    embeddings = tf.concat([self.user_embeddings, self.item_embeddings], axis=0)
    all_embeddings = [embeddings]

    for k in range(0, self.K):
        embeddings = tf.matmul(self.A_hat, embeddings)
        embeddings = tf.nn.sigmoid(tf.matmul(embeddings, self.filters[k]))
        all_embeddings += [embeddings]

    all_embeddings = tf.concat(all_embeddings, 1)
    self.u_embeddings, self.i_embeddings = tf.split(
        all_embeddings, [self.num_users, self.num_items], 0
    )

    self.u_embeddings = tf.nn.embedding_lookup(self.u_embeddings, users)
    all_ratings = tf.matmul(
        self.u_embeddings, self.i_embeddings, transpose_a=False, transpose_b=True
    )

    return all_ratings
```

Figura 18: Código para gerar predições.

Para a validação da implementação, comparando com os resultados obtidos com o *paper* original, foi usada a métrica de *recall* e foi gerado o gráfico de *loss*. O recall mensura a razão entre a quantidade de itens relevantes retornados na predição e o total de itens relevantes que existem. A fórmula a seguir mostra o cálculo. A função de loss, calcula a diferença entre os resultados esperados e os retornados pelo algoritmo.

$$recall = \frac{N\ Relevantes}{Total\ Relevantes}$$

Na implementação feita na API o recall obtido foi 0.055 (esse resultado pode ser consultado em: https://github.com/Luiz-FS/Deep4Rec/blob/create-spectralcf-model/examples/validation/Spectral%20Collaborative%20Filtering.ipynb) e a mesma medida no *paper* original foi 0.051.

Fazendo análise do gráfico de *loss* da API com a implementação original utilizando o *BPR [14] (Bayesian Personalized Ranking)* como função de *loss*, os resultados obtidos são similares. Ao final de 200 épocas, os valores de loss obtidos para a versão da API e a original são respectivamente 0.357544 e 0.351821.

Comparando os dois resultados pode-se observar que são semelhantes aos do *paper*, isso traz indícios fortes que o algoritmo feito na API está funcionando como esperado.

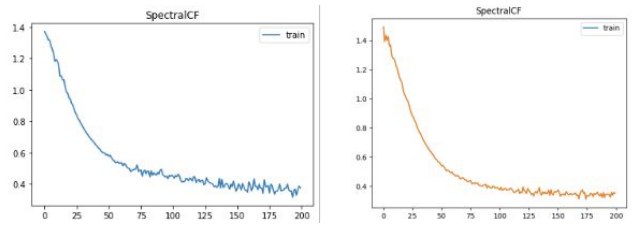

Figura 19: Gráficos de custo do algoritmo implementado na API e o original. O gráfico da esquerda é o implementado na API e o da direita, o original.

3. EXPERIÊNCIA

3.1 Processo de desenvolvimento

Para a adição de novas *features* aos datasets foi necessário entender como eram feitos os mapeamentos das *features* básicas já existentes, estudar a estrutura que o TensorFlow [3] fornece para esse fim e codificar os dados de gênero em vetores numéricos. Dessa forma foi possível adicionar os novos metadados de gênero aos datasets.

Para o desenvolvimento do novo algoritmo, foi necessário estudar detalhadamente o *paper do mesmo,* entendendo os passos da fórmula e mapeando para a API de forma que a implementação e os resultados fiquem os mais fiéis possíveis aos originais.

3.2 Desafios

O desenvolvimento do novo algoritmo foi desafiante, pois foi necessário estudar muitos conceitos que eu ainda não havia visto para entender o seu funcionamento e após isso implementar o algoritmo seguindo a estrutura da API.

Outro grande desafio foi resolver o problema que ocorreu no código ao usar o gradiente descendente para otimizar as variáveis do modelo. Ao aplicar o gradiente, o TensorFlow [3] não estava conseguindo criar o grafo de fluxo entre as variáveis do modelo e a função de custo que é utilizada. Para resolver esse problema foi necessário investigar em qual parte do código o gradiente não estava conseguindo criar conexões entre as variáveis e a função de custo, após isso, utilizar os métodos fornecido pelo TensorFlow [3] para permitir que essa conexão possa ser criada.

3.3 Trabalhos futuros

Para trabalhos futuros pode-se estender a API com os seguintes pontos:

- Adicionar mais algoritmos e datasets para deixá-la cada vez mais completa, deixando de ser uma API em fase preliminar.
- Dar suporte para mais features nos algoritmos já implementados atualmente.
- Disponibilizar a API para a instalação via package manager.
- Testar mais a API para ter mais confiabilidade.

3.4 REFERÊNCIA

[1] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep Learning Based Recommender System: A Survey and New Perspectives. ACM Comput. Surv. 52, 1, Article 5 (February 2019), 38 pages. https://dl.acm.org/citation.cfm?id=3285029.

[2] Lei Zheng, et al. 2018. Spectral Collaborative Filtering. 12th ACM Conference on Recommender Systems Pages 311-319. https://dl.acm.org/citation.cfm?id=3240343 .

[3] Tensorflow. Disponível em https://www.tensorflow.org/.

[4] Rendle, Steffen. 2010. Factorization Machines. IEEE International Conference on Data Mining Pages 995-1000. https://dl.acm.org/citation.cfm?id=1934620.

[5] He, Xiangnan; Chua, Tat-Seng. 2017. Neural Factorization Machines for Sparse Predictive Analytics.40th International ACM SIGIR Conference on Research and Development in Information Retrieval Pages 355-364. https://dl.acm.org/citation.cfm?id=3080777.

[6] Cheng, Heng-Tze; et al. 2016. Wide & Deep Learning for Recommender Systems. 1st Workshop on Deep Learning for Recommender Systems Pages 7-10. https://dl.acm.org/citation.cfm?id=2988454.

[7] He, Xiangnan; et al.. 2017. Neural Collaborative Filtering. 26th International Conference on World Wide Web Pages 173-182. https://dl.acm.org/citation.cfm?id=3052569.

[8] How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. Acessado e: 2019. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#6831604960ba.

[9] Dacrema, Maurizio; Cremonesi, Paolo; Jannach, Dietmar. 2019. Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches. 13th ACM Conference on Recommender Systems. https://dl.acm.org/citation.cfm?id=3347058.

[10] RecSys 2019 (Copenhagen) – RecSys. Acessado em: 2019. https://recsys.acm.org/recsys19/.

[11] How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. Acessado e: 2019. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#6831604960ba.

[12] Dacrema, Maurizio; Cremonesi, Paolo; Jannach, Dietmar. 2019. Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches. 13th ACM Conference on Recommender Systems. https://dl.acm.org/citation.cfm?id=3347058.

[13] RecSys 2019 (Copenhagen) – RecSys. Acessado em: 2019. https://recsys.acm.org/recsys19/.

[14] Frappe dataset. Acessado em 2019. https://raw.githubusercontent.com/hexiangnan/neural_factorization_machine/master/data/frappe/.

[15] Census dataset. Acessado em: 2019. https://archive.ics.uci.edu/ml/machine-learning-databases/adult. MovieLens dataset. Acessado em: 2019. https://movielens.org/.

[16] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence. AUAI Press, 452–461.