Master IAII
Pr. Sara QASSIMI
08 Decembre 2023

Tutorial: Autoencoder for Collaborative Filtering

Objective:

This tutorial guides you through building a collaborative filtering autoencoder for recommender systems, from data loading to model training and results visualization. Experiment with hyperparameters and architecture to improve the model's effectiveness in predicting missing values in a collaborative filtering scenario.

Prerequisites

Ensure that you have Keras and other necessary libraries installed. If needed, update the future module using `sudo pip install -U future.`

Step 1: Import Libraries

```
# Import necessary libraries
from __future__ import print_function, division
from builtins import range
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from scipy.sparse import save_npz, load_npz

import keras.backend as K
from keras.models import Model
from keras.layers import Input, Dropout, Dense
from keras.regularizers import l2
from keras.optimizers import SGD
```

Step 2: Load Data

```python
# Configuration Variables
batch_size = 128
epochs = 20
reg = 0.0001

# Load Data
A = load_npz("Atrain.npz")
A_test = load_npz("Atest.npz")
mask = (A > 0) * 1.0
mask_test = (A_test > 0) * 1.0

# Data Copy for Shuffling
A_copy = A.copy()
mask_copy = mask.copy()
A_test_copy = A_test.copy()
mask_test_copy = mask_test.copy()

N, M = A.shape
print("N:", N, "M:", M)
print("N // batch_size:", N // batch_size)
```

- ☐ Configuration Variables: Set hyperparameters like batch size, epochs, and regularization.
- ☐ Load Data: Load training and test data in compressed sparse row (CSR) matrix format.
- ☐ Create Masks: Define masks to identify missing entries in the data.
- ☐ Data Copy: Create copies for shuffling during training to avoid data order issues.

Step 3: Data Preprocessing

Center the data by calculating the mean.

```
# Center the data
mu = A.sum() / mask.sum()
print("mu:", mu)
```

Step 4: Building the Autoencoder Model

```
# Build the Model - Simple Autoencoder
i = Input(shape=(M,))
x = Dropout(0.7)(i)
x = Dense(700, activation='tanh', kernel_regularizer=l2(reg))(x)
x = Dense(M, kernel_regularizer=l2(reg))(x)

model = Model(i, x)
model.compile(
  loss=custom_loss,
  optimizer=SGD(lr=0.08, momentum=0.9),
  metrics=[custom_loss],
)
```

- ☐ Model Architecture: Construct a simple autoencoder using Keras functional API.
- ☐ Dropout Layer: Add a dropout layer to prevent overfitting.
- ☐ Dense Layers: Two dense layers form the encoder and decoder.
- ☐ Model Compilation: Compile the model with a custom loss function and SGD optimizer.

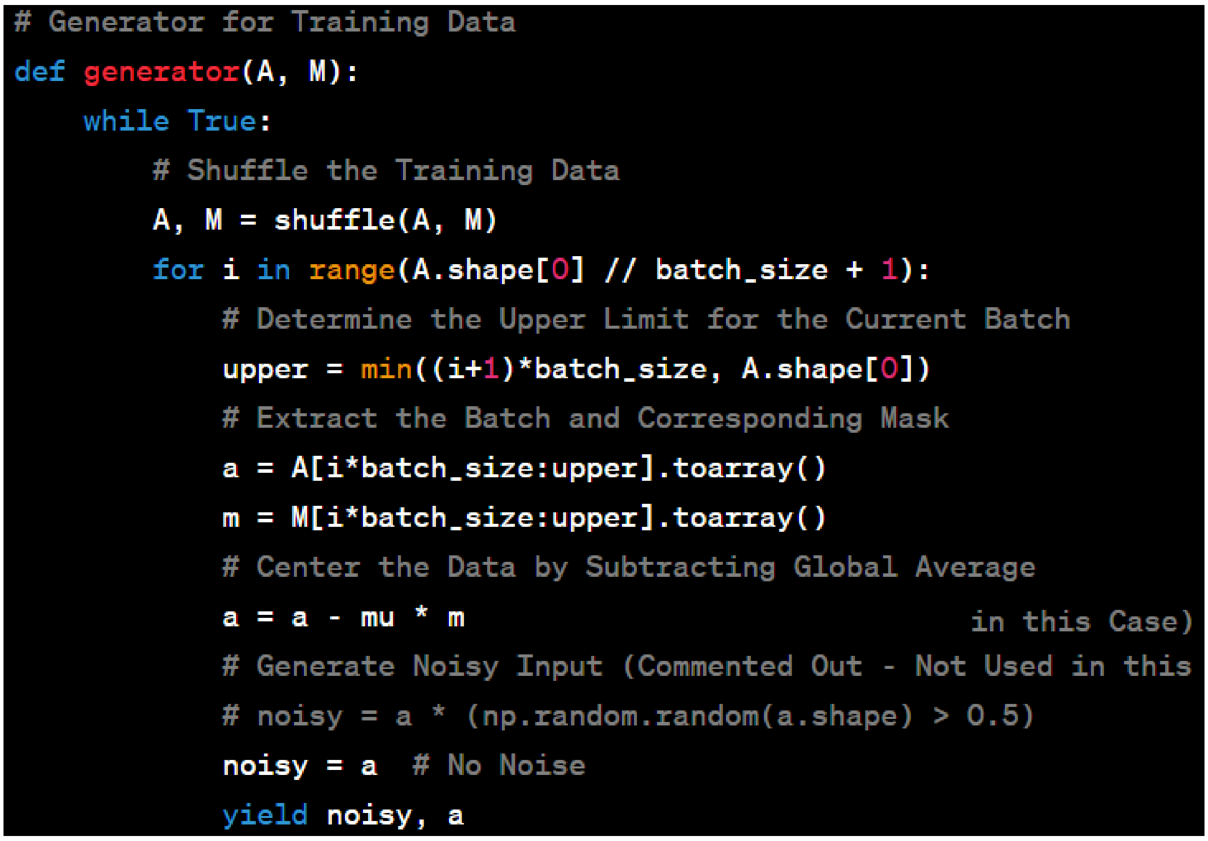
Step 5: Custom Loss Function

```
def custom_loss(y_true, y_pred):
    mask = K.cast(K.not_equal(y_true, 0), dtype='float32')
    diff = y_pred - y_true
    sqdiff = diff * diff * mask
    sse = K.sum(K.sum(sqdiff))
    n = K.sum(K.sum(mask))
    return sse / n
```

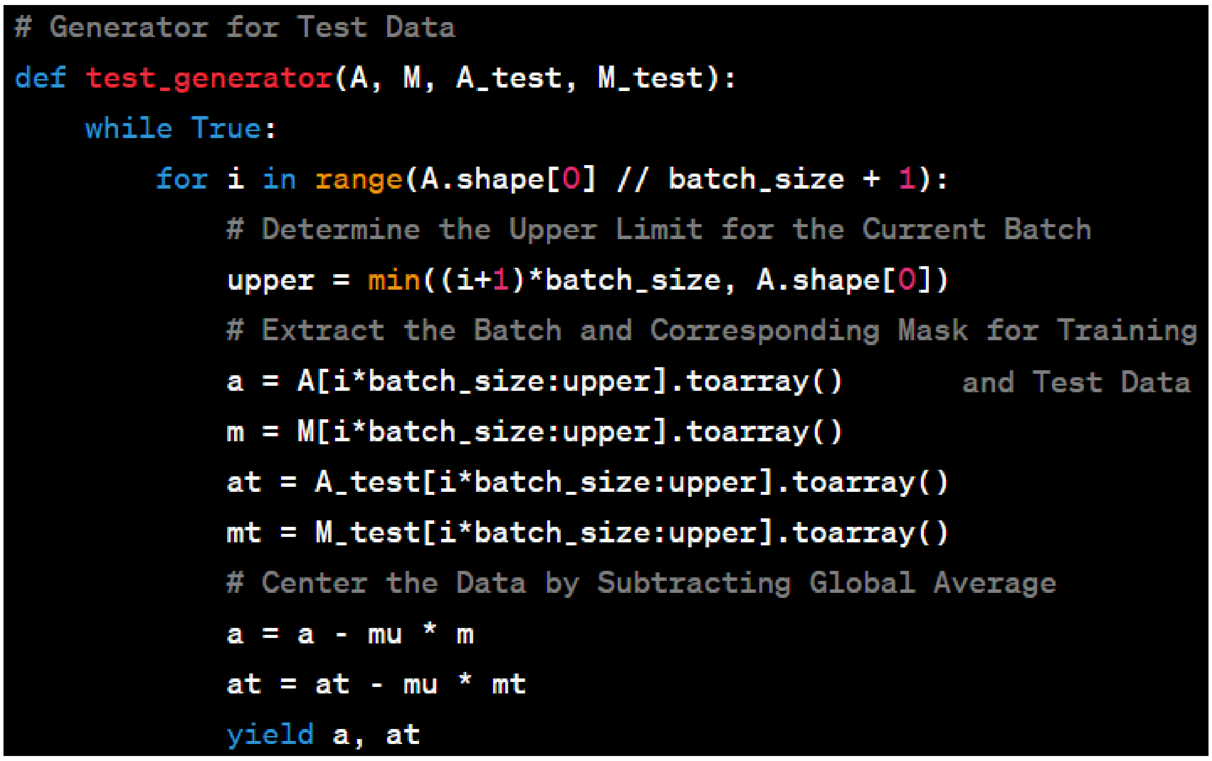
- ☐ Custom Loss Function: Define a loss function to handle missing values in predictions.
- ☐ Mask Creation: Create a mask to identify non-missing entries in the true values.
- ☐ Squared Differences: Calculate squared differences, considering only non-missing values.
- ☐ Sum of Squared Errors (SSE): Sum the squared differences.
- ☐ Normalize by Count: Normalize SSE by the count of non-missing values.

Step 6: Data Generators

Create data generators for training and testing.



```
# Generator for Training Data
def generator(A, M):
    while True:
        # Shuffle the Training Data
        A, M = shuffle(A, M)
        for i in range(A.shape[0] // batch_size + 1):
            # Determine the Upper Limit for the Current Batch
            upper = min((i+1)*batch_size, A.shape[0])
            # Extract the Batch and Corresponding Mask
            a = A[i*batch_size:upper].toarray()
            m = M[i*batch_size:upper].toarray()
            # Center the Data by Subtracting Global Average
            a = a - mu * m                                in this Case)
            # Generate Noisy Input (Commented Out - Not Used in this
            # noisy = a * (np.random.random(a.shape) > 0.5)
            noisy = a  # No Noise
            yield noisy, a
```

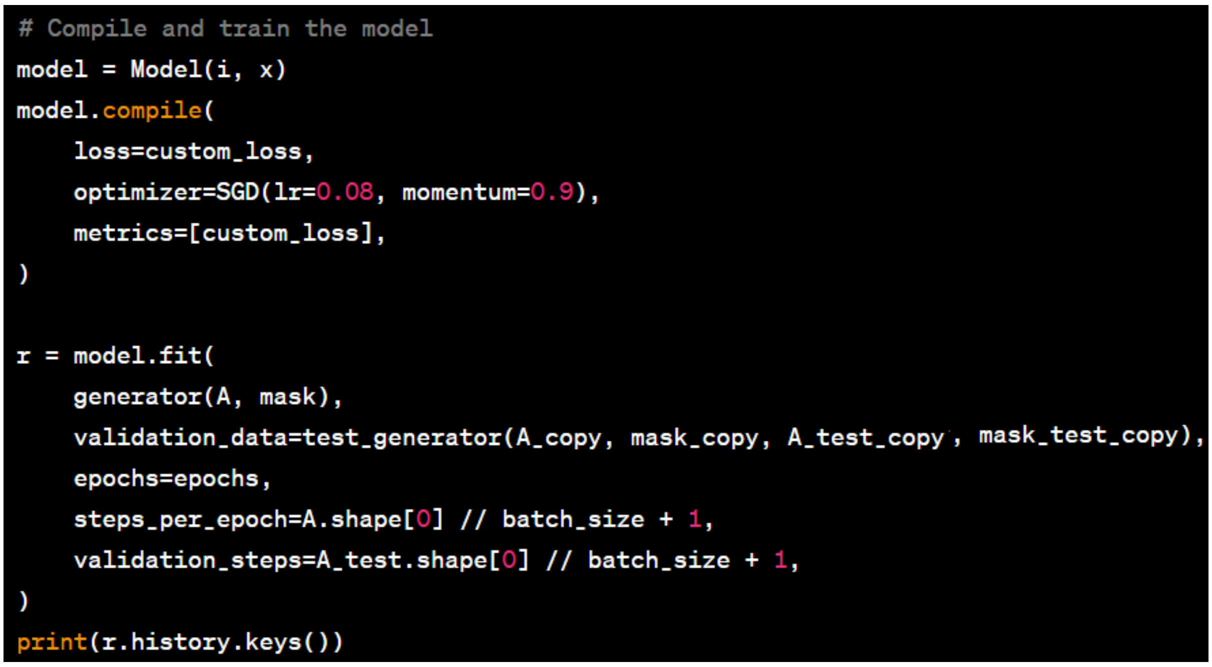


```
# Generator for Test Data
def test_generator(A, M, A_test, M_test):
    while True:
        for i in range(A.shape[0] // batch_size + 1):
            # Determine the Upper Limit for the Current Batch
            upper = min((i+1)*batch_size, A.shape[0])
            # Extract the Batch and Corresponding Mask for Training
            a = A[i*batch_size:upper].toarray()       and Test Data
            m = M[i*batch_size:upper].toarray()
            at = A_test[i*batch_size:upper].toarray()
            mt = M_test[i*batch_size:upper].toarray()
            # Center the Data by Subtracting Global Average
            a = a - mu * m
            at = at - mu * mt
            yield a, at
```

- ☐ Data Generators: Create generators for training and test data to handle large datasets.
- ☐ Shuffling: Shuffle data during training to avoid learning order-dependent patterns.
- ☐ Centering Data: Subtract global average (mu) from the ratings while preserving zeros.
- ☐ Noise (Commented Out): Additional code for adding noise to the input data (not used in this case).

Step 6: Compiling and Training the Model

Compile the model and train it using the generators.



```
# Compile and train the model
model = Model(i, x)
model.compile(
    loss=custom_loss,
    optimizer=SGD(lr=0.08, momentum=0.9),
    metrics=[custom_loss],
)

r = model.fit(
    generator(A, mask),
    validation_data=test_generator(A_copy, mask_copy, A_test_copy, mask_test_copy),
    epochs=epochs,
    steps_per_epoch=A.shape[0] // batch_size + 1,
    validation_steps=A_test.shape[0] // batch_size + 1,
)
print(r.history.keys())
```

- ☐ Model Training: Train the model using the fit generator function.
- ☐ Training and Validation Data: Use the data generators for both training and validation.
- ☐ Epochs and Batch Size: Set the number of epochs and steps per epoch.
- ☐ History: Collect training history for later visualization.

Step 7: Visualizing Results

Plot the training and testing losses.

```
# Plot Losses
plt.plot(r.history['loss'], label="train loss")
plt.plot(r.history['val_loss'], label="test loss")
plt.legend()
plt.show()

# Plot Mean Squared Error
plt.plot(r.history['custom_loss'], label="train mse")
plt.plot(r.history['val_custom_loss'], label="test mse")
plt.legend()
plt.show()
```

- ☐ Visualize Losses: Plot training and validation losses to assess model performance.
- ☐ Mean Squared Error (MSE): Plot MSE for both training and validation sets.
- ☐ Adjust Hyperparameters: Analyze results and consider adjusting hyperparameters for better performance.

Future Challenges and Steps for Improvement

While the current implementation serves as a solid foundation for collaborative filtering, there are several areas for improvement.One potential challenge is handling larger datasets more efficiently, considering the computational demands of training deep models. Additionally, exploring advanced techniques such as incorporating embeddings or exploring different neural network architectures may enhance the model's predictive performance. Regular model evaluation, hyperparameter tuning, and scalability considerations are key steps to further improve the code for real-world collaborative filtering scenarios. Experimenting with different optimization algorithms and regularization techniques could also be valuable in refining the model's generalization capabilities. Keep in mind that collaborative filtering is an evolving field, and staying informed about the latest advancements can contribute to the continual improvement of recommendation systems.