Recommender System (Spring 2022)

Homework #5 (80 Pts, May 29)

Student ID 2019311195

Name 김지유

(1) [30 pts] We provide template code and dataset in Python. Refer to 'models/FM_explicit.py,' write your code to implement the "FieldAwareFactorizationMachine" function in 'models/FFM_explicit.py.'

The prediction by FFM is defined as follows:

$$\phi_{FFM}(w,x) = \sum_{j_1=1}^{n} \sum_{j_2=j_1+1}^{n} (w_{j_1,f_2} w_{j_2,f_1}) x_{j_1} x_{j_2}$$

where f_1 and f_2 denote the field of j_1 and j_2, respectively.

Note: Fill in your code here. You also have to submit your code to i-campus.

Answer:

 # class FMM_implicit의 forward 함수에서 second를 수정하였습니다.

```
import os
import math
from time import time
from tqdm import tqdm

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

from sklearn.metrics import roc_auc_score, log_loss
from torch.utils.data import DataLoader

class FFM_implicit(torch.nn.Module):
    def __init__(self, train_data, train_label, valid_data, valid_label,
field_dims, embed_dim,
                 num_epochs, early_stop_trial, learning_rate, reg_lambda,
batch_size,  device):
```

```
        super().__init__()

        self.train_data = train_data
        self.train_label = train_label
        self.valid_data = valid_data
        self.valid_label = valid_label
        self.field_dims = field_dims
        self.embed_dim = embed_dim

        self.num_epochs = num_epochs
        self.early_stop_trial = early_stop_trial
        self.learning_rate = learning_rate
        self.reg_lambda = reg_lambda
        self.batch_size = batch_size

        self.device = device

        self.build_graph()

    def build_graph(self):
        self.linear = FeaturesLinear(self.field_dims)
        self.ffm = FieldAwareFactorizationMachine(self.field_dims, self.embed_dim)

        self.criterion = nn.BCELoss()
        self.optimizer = torch.optim.Adam(self.parameters(), 
lr=self.learning_rate, weight_decay=self.reg_lambda)

        self.to(self.device)

    def forward(self, x):
        first = self.linear(x)
        #second = torch.sum(self.ffm(x), dim=1, keepdim=True)
        second = torch.sum(torch.sum(self.ffm(x), dim=1), dim=1, keepdim=True)

        x = first + second
        output = torch.sigmoid(x.squeeze(1))
        return output

    def fit(self):
        train_loader = DataLoader(range(self.train_data.shape[0]), 
batch_size=self.batch_size, shuffle=True)

        best_AUC = 0
        num_trials = 0

        for epoch in range(1, self.num_epochs+1):
            # Train
            self.train()
```

```
            for b, batch_idxes in enumerate(train_loader):
                batch_data = torch.tensor(self.train_data[batch_idxes],
dtype=torch.long, device=self.device)
                batch_labels = torch.tensor(self.train_label[batch_idxes],
dtype=torch.float, device=self.device)

                loss = self.train_model_per_batch(batch_data, batch_labels)

            # Valid
            self.eval()
            pred_array = self.predict(self.valid_data)
            AUC = roc_auc_score(self.valid_label, pred_array)
            logloss = log_loss(self.valid_label, pred_array)

            if AUC > best_AUC:
                best_AUC = AUC
                torch.save(self.state_dict(),
f"saves/{self.__class__.__name__}_best_model.pt")
                num_trials = 0
            else:
                num_trials += 1

            if num_trials >= self.early_stop_trial and self.early_stop_trial>0:
                print(f'Early stop at epoch:{epoch}')
                self.restore()
                break

            print(f'epoch {epoch} train_loss = {loss:.4f} valid_AUC = {AUC:.4f}
valid_log_loss = {logloss:.4f}')
        return

    def train_model_per_batch(self, batch_data, batch_labels):
        self.optimizer.zero_grad()

        logits = self.forward(batch_data)
        loss = self.criterion(logits, batch_labels)
        loss.backward()

        self.optimizer.step()

        return loss

    def predict(self, pred_data):
        self.eval()

        pred_data_loader = DataLoader(range(pred_data.shape[0]),
batch_size=self.batch_size, shuffle=False)

```

```python
        pred_array = np.zeros(pred_data.shape[0])

        for b, batch_idxes in enumerate(pred_data_loader):
            batch_data = torch.tensor(pred_data[batch_idxes], dtype=torch.long,
device=self.device)
            with torch.no_grad():
                pred_array[batch_idxes] = self.forward(batch_data).cpu().numpy()

        return pred_array

    def restore(self):
        with open(f"saves/{self.__class__.__name__}_best_model.pt", 'rb') as f:
            state_dict = torch.load(f)

        self.load_state_dict(state_dict)

class FieldAwareFactorizationMachine(torch.nn.Module):
    def __init__(self, field_dims, embed_dim):
        super().__init__()

        self.num_fields = len(field_dims)
        self.offsets = np.array((0, *np.cumsum(field_dims)[:-1]), dtype=np.long)
        self.embeddings = torch.nn.ModuleList([
            torch.nn.Embedding(sum(field_dims), embed_dim) for _ in
range(self.num_fields)
        ])

        for embedding in self.embeddings:
            torch.nn.init.xavier_uniform_(embedding.weight.data)

    def forward(self, x):
        # ========================= EDIT HERE ========================
        #output = None
        x = x + x.new_tensor(self.offsets).unsqueeze(0)
        xs = [self.embeddings[i](x) for i in range(self.num_fields)]
        output = list()
        for i in range(self.num_fields - 1):
            for j in range(i + 1, self.num_fields):
                output.append(xs[j][:, i] * xs[i][:, j])
        output = torch.stack(output, dim=1)
        # ========================= EDIT HERE ========================
        return output


class FeaturesLinear(torch.nn.Module):
    def __init__(self, field_dims, output_dim=1):
        super().__init__()
        self.fc = torch.nn.Embedding(sum(field_dims), output_dim)
        self.bias = torch.nn.Parameter(torch.zeros((output_dim,)))
```

```
        self.offsets = np.array((0, *np.cumsum(field_dims)[:-1]), dtype=np.long)

    def forward(self, x):
        x = x + x.new_tensor(self.offsets).unsqueeze(0)

        return torch.sum(self.fc(x), dim=1) + self.bias
```

(2) [30 pts] Refer to 'models/FM_explicit.py,' write your code to implement the function "forward" in 'models/DeepFM_explicit.py.'

The prediction by DeepFM is defined as follows:

$\hat{y}_{\text{DeepFM}} = \sigma(\hat{y}_{\text{FM}} + \hat{y}_{\text{MLP}})$ where σ denotes a sigmoid activation function.

Note: Fill in your code here. You also have to submit your code to i-campus.

Answer:

```
import os
import math
from time import time
from tqdm import tqdm

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

from sklearn.metrics import roc_auc_score, log_loss
from torch.utils.data import DataLoader


class DeepFM_implicit(torch.nn.Module):
    def __init__(self, train_data, train_label, valid_data, valid_label,
field_dims, embed_dim, mlp_dims, dropout,
                 num_epochs, early_stop_trial, learning_rate, reg_lambda,
batch_size, device):
```

```python
        super().__init__()

        self.train_data = train_data
        self.train_label = train_label
        self.valid_data = valid_data
        self.valid_label = valid_label
        self.field_dims = field_dims
        self.embed_dim = embed_dim
        self.embed_output_dim = len(field_dims) * embed_dim
        self.mlp_dims = mlp_dims
        self.dropout = dropout

        self.num_epochs = num_epochs
        self.early_stop_trial = early_stop_trial
        self.learning_rate = learning_rate
        self.reg_lambda = reg_lambda
        self.batch_size = batch_size

        self.device = device

        self.build_graph()

    def build_graph(self):
        self.linear = FeaturesLinear(self.field_dims)
        self.embedding = FeaturesEmbedding(self.field_dims, self.embed_dim)
        self.fm = FactorizationMachine()
        self.mlp = MultiLayerPerceptron(self.embed_output_dim, self.mlp_dims,
self.dropout)

        self.criterion = nn.BCELoss()
        self.optimizer = torch.optim.Adam(self.parameters(),
lr=self.learning_rate, weight_decay=self.reg_lambda)

        self.to(self.device)

    def forward(self, x):
        # ========================= EDIT HERE ========================
        #output = None
        embed_x = self.embedding(x)
        embed_output_dim = len(self.field_dims) * self.embed_dim
        output = self.linear(x) + self.fm(embed_x) + self.mlp(embed_x.view(-1,
embed_output_dim))
        output = torch.sigmoid(output.squeeze(1))
        # ========================= EDIT HERE ========================
        return output

    def fit(self):
        train_loader = DataLoader(range(self.train_data.shape[0]),
```

```
batch_size=self.batch_size, shuffle=True)

        best_AUC = 0
        num_trials = 0
        for epoch in range(1, self.num_epochs+1):
            # Train
            self.train()
            for b, batch_idxes in enumerate(train_loader):
                batch_data = torch.tensor(self.train_data[batch_idxes],
dtype=torch.long, device=self.device)
                batch_labels = torch.tensor(self.train_label[batch_idxes],
dtype=torch.float, device=self.device)

                loss = self.train_model_per_batch(batch_data, batch_labels)

            # Valid
            self.eval()
            pred_array = self.predict(self.valid_data)
            AUC = roc_auc_score(self.valid_label, pred_array)
            logloss = log_loss(self.valid_label, pred_array)

            if AUC > best_AUC:
                best_AUC = AUC
                torch.save(self.state_dict(),
f"saves/{self.__class__.__name__}_best_model.pt")
                num_trials = 0
            else:
                num_trials += 1

            if num_trials >= self.early_stop_trial and self.early_stop_trial>0:
                print(f'Early stop at epoch:{epoch}')
                self.restore()
                break

            print(f'epoch {epoch} train_loss = {loss:.4f} valid_AUC = {AUC:.4f}
valid_log_loss = {logloss:.4f}')
        return

    def train_model_per_batch(self, batch_data, batch_labels):
        self.optimizer.zero_grad()

        logits = self.forward(batch_data)
        loss = self.criterion(logits, batch_labels)
        loss.backward()

        self.optimizer.step()

        return loss
```

```python

    def predict(self, pred_data):
        self.eval()

        pred_data_loader = DataLoader(range(pred_data.shape[0]),
batch_size=self.batch_size, shuffle=False)

        pred_array = np.zeros(pred_data.shape[0])

        for b, batch_idxes in enumerate(pred_data_loader):
            batch_data = torch.tensor(pred_data[batch_idxes], dtype=torch.long,
device=self.device)
            with torch.no_grad():
                pred_array[batch_idxes] = self.forward(batch_data).cpu().numpy()

        return pred_array

    def restore(self):
        with open(f"saves/{self.__class__.__name__}_best_model.pt", 'rb') as f:
            state_dict = torch.load(f)

        self.load_state_dict(state_dict)

```

```python
class MultiLayerPerceptron(torch.nn.Module):
    def __init__(self, input_dim, embed_dims, dropout, output_layer=True):
        super().__init__()
        layers = list()

        for embed_dim in embed_dims:
            layers.append(torch.nn.Linear(input_dim, embed_dim))
            layers.append(torch.nn.BatchNorm1d(embed_dim))
            layers.append(torch.nn.ReLU())
            layers.append(torch.nn.Dropout(p=dropout))
            input_dim = embed_dim

        if output_layer:
            layers.append(torch.nn.Linear(input_dim, 1))

        self.mlp = torch.nn.Sequential(*layers)

    def forward(self, x):
        """
        :param x: Float tensor of size ``(batch_size, embed_dim)``
        """
        return self.mlp(x)

```

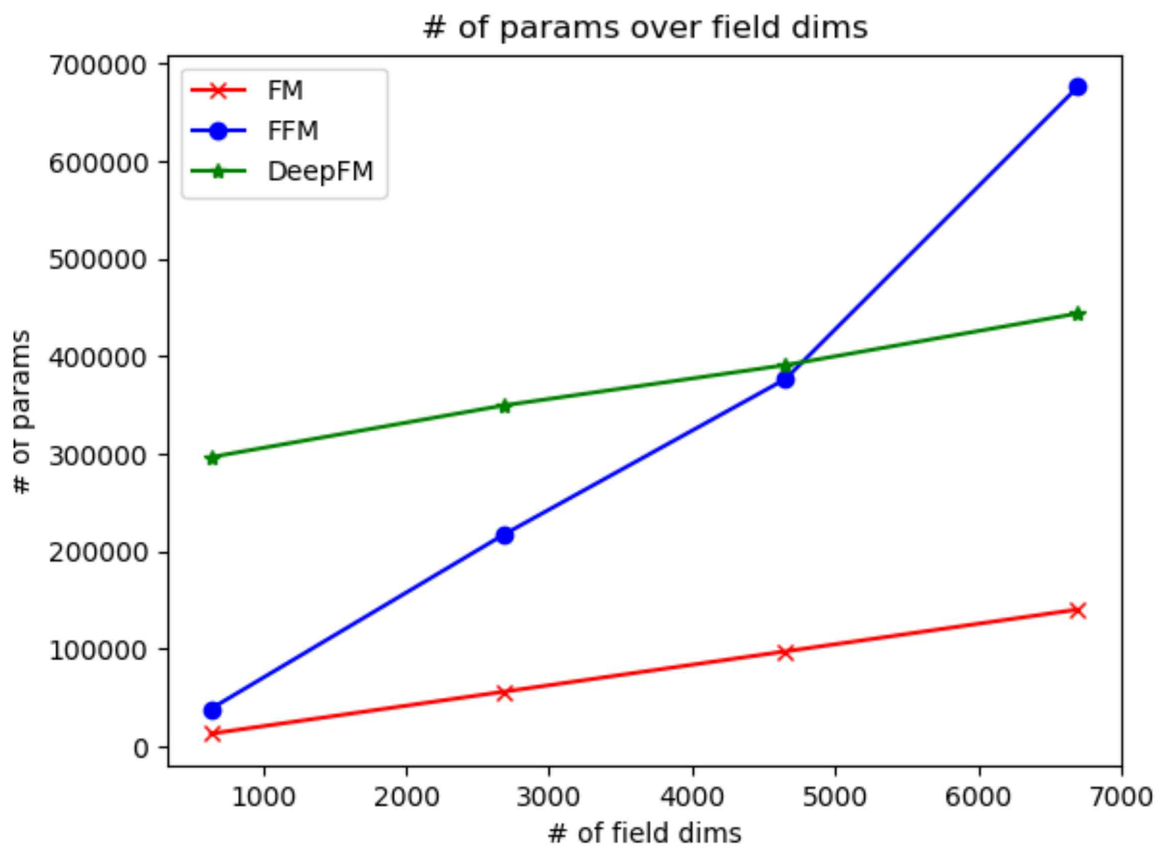
```python
class FactorizationMachine(torch.nn.Module):
```

```
    def __init__(self, reduce_sum=True):
        super().__init__()
        self.reduce_sum = reduce_sum

    def forward(self, x):
        """
        :param x: Float tensor of size ``(batch_size, num_fields, embed_dim)``
        """
        square_of_sum = torch.sum(x, dim=1) ** 2
        sum_of_square = torch.sum(x ** 2, dim=1)
        ix = square_of_sum - sum_of_square

        if self.reduce_sum:
            ix = torch.sum(ix, dim=1, keepdim=True)

        return 0.5 * ix


class FeaturesEmbedding(torch.nn.Module):
    def __init__(self, field_dims, embed_dim):
        super().__init__()
        self.embedding = torch.nn.Embedding(sum(field_dims), embed_dim)
        self.offsets = np.array((0, * np.cumsum(field_dims)[:-1]), dtype=np.long)
        torch.nn.init.xavier_uniform_(self.embedding.weight.data)

    def forward(self, x):
        """
        :param x: Long tensor of size ``(batch_size, num_fields)``
        """
        x = x + x.new_tensor(self.offsets).unsqueeze(0)

        return self.embedding(x)


class FeaturesLinear(torch.nn.Module):
    def __init__(self, field_dims, output_dim=1):
        super().__init__()
        self.fc = torch.nn.Embedding(sum(field_dims), output_dim)
        self.bias = torch.nn.Parameter(torch.zeros((output_dim,)))
        self.offsets = np.array((0, *np.cumsum(field_dims)[:-1]), dtype=np.long)

    def forward(self, x):
        """
        :param x: Long tensor of size ``(batch_size, num_fields)``
        """
        x = x + x.new_tensor(self.offsets).unsqueeze(0)

        return torch.sum(self.fc(x), dim=1) + self.bias
```

(3) [20 pts] Given the data ('naver_movie_dataset.csv'), draw the plots of the number of parameters over the number of fields for FM, FFM, and DeepFM. For varying the field dimensions, explain the result of the model architectures. Run '2_plot.py' to run the source code.

Note: Please show the results for two datasets in the code. Show your plots and briefly explain why.

Answer:



파라미터의 개수는 FM, FFM, DeepFM 순서로 적었다. 하지만, field dims의 개수가 증가할수록 FFM이 DeepFM보다 많아지는 것을 볼 수 있었다. FM은 하나의 feature가 하나의 latent vector를 가지지만 FFM은 하나의 feature가 여러 개의 latent feature를 가질 수 있기 때문에 FM보다 FFM이 더 많은 파라미터를 갖는다. DeepFM은 FM과 MLP를 합친 모델이기 때문에 당연히 FM보다 더 많은 파라미터를 갖는다. 처음에는 DeepFM의 MLP 때문에 DeepFM이 FFM보다 많은 파라미터를 갖지만, field dims가 증가할수록 FFM의 latent feature 역시 커지기 때문에 결국 FFM이 DeepFM보다 더 많은 파라미터를 갖게 된다.