Final project

<Recommend system model 실험>

2019040973

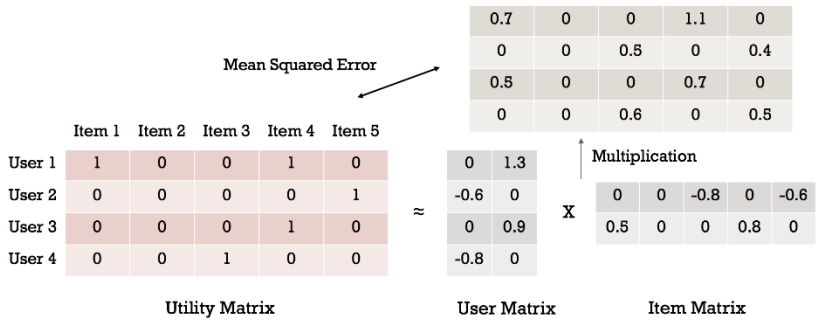
소프트웨어학부

유은정

1. Matrix Factorization

- **개념**

Collaborative filtering을 구현하는 방법 중 하나로 Matrix Factorization을 사용할 수 있다. Matrix Factorization이란, User와 Item간의 rating 관계를 계산하기 위해 User matrix와 Item matrix를 내적하여 Rating Matrix를 완성시키는 방법이다. Rating matrix는 sparse하므로 user가 평가한 item들에 대해서만 loss를 계산한 뒤 예측 model을 학습해 나간다.



- **Model, training code 설명**

```
class ModelClass(nn.Module):
    def __init__(
        self, user_num, item_num, factor_num,
    ):
        super(ModelClass, self).__init__()
        self.factor_num = factor_num

        self.embed_user = nn.Embedding(user_num, factor_num)
        self.embed_item = nn.Embedding(item_num, factor_num)
        predict_size = factor_num
        self.predict_layer = torch.ones(predict_size, 1).cuda()
        self._init_weight_()

    def _init_weight_(self):
        nn.init.normal_(self.embed_user.weight, std=0.01)
        nn.init.normal_(self.embed_item.weight, std=0.01)

        for m in self.modules():
            if isinstance(m, nn.Linear) and m.bias is not None:
                m.bias.data.zero_()

    def forward(self, user, item):
        embed_user = self.embed_user(user)
        embed_item = self.embed_item(item)
        output_GMF = embed_user * embed_item
        prediction = torch.matmul(output_GMF, self.predict_layer)
        return prediction.view(-1)
```

```
def create_model(user_num, item_num, factor_num=1):
    model = ModelClass(user_num, item_num, factor_num=20)
    loss_function = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    return model, loss_function, optimizer

model, criterion, optimizer = create_model(610, 193609, 20)
model.cuda()


train_data = RecommendationDataset(f"{args.dataset}/ratings.csv", train=True)
train_data, validation_data = train_test_split(train_data,test_size=0.1)
train_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True)
validation_loader = DataLoader(validation_data, batch_size=args.batch_size, shuffle=True)

for epoch in range(100):
    cost = 0;
    for users, items, ratings in train_loader:
        users = users.cuda()
        items = items.cuda()
        ratings = ratings.float().cuda()

        optimizer.zero_grad()
        ratings_pred = model(users,items)
        loss = criterion(ratings_pred, ratings)
        loss.backward()
        optimizer.step()
        cost += loss.item()*len(ratings)

    cost/= 81676

    print(f"Epoch: {epoch}")
    print("train cost: {: .6f}".format(cost))

    cost_validation = 0;
    for users, items, ratings in validation_loader:
        users = users.cuda()
        items = items.cuda()
        ratings = ratings.float().cuda()

        ratings_pred = model(users,items)
        loss = criterion(ratings_pred, ratings)
        cost_validation += loss.item()*len(ratings)
    cost_validation/= 9076
    print("validation cost: {: .6f}".format(cost_validation))
```

1) Model

- **__init__**

User num : user의 수

Item num : item의 수

Factor num : embedding 시킬 dim의 개수

Self.embed_user = nn.Embedding(user_num, factor_num) : user 정보를 embedding시킬 공간을 확보한다.

Self.embed_item = nn.Embedding(item_num, factor_num) : item 정보를 embedding시킬 공간을 확보한다.

- **__init_weight__**

User와 item의 weight를 초기화 시킨다. For문을 통해 bias 역시 초기화.

- **Forward**

입력 받은 user와 item을 embedding 시킨 후 두 값을 서로 내적 하여 prediction 값을 도출한다.

2) Training, validation

- data load

Data set을 가져온 후, train_test_split을 이용하여 training set과 validation set을 9:1로 나누어 저장. 각 data set은 train loader와 validation loader로 나누어 들어간다.

- training

Train loader를 이용하여 User, item, rating 정보를 받고 cuda()를 통해 gpu로 로드.

Gradient를 초기화하고, model에 data를 넣어 rating_pred 값을 받는다. 해당 값을 criterion에 넣어 실제 값과의 loss를 계산한다.

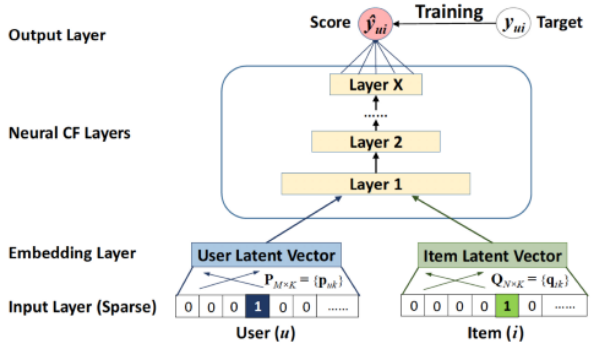
Loss를 backward하고 weight 값을 업데이트. loss 값을 저장하고, 학습한 data의 개수만큼 나누어 cost를 도출한다.

- validation

Validation loader를 이용하여 user, item, rating 정보를 받고, cuda()를 통해 gpu로 로드.

Validation set을 이용해 loss 값을 계산한다. 검증을 위한 단계이므로 weight를 업데이트 하지 않는다.

2. Neural Collaborative Filtering



- **개념**

Collaborative filtering을 구성하는 방법 중 하나로, neural collaborative filtering 방법을 사용할 수 있다. Neural collaborative filtering이란, data를 one-hot encoding 방식으로 embedding한 후, multi-layer perceptron 방식을 사용하여 여러 layer를 거쳐 model을 학습시키고 user의 rating 값을 추측하는 방식이다.

- **Model, training code 설명**

```
class ModelClass(nn.Module):
    def __init__(
        self, user_num, item_num, factor_num, num_layers, dropout,
    ):
        super(ModelClass, self).__init__()

        self.dropout = dropout

        self.embed_user_MLP = nn.Embedding(
            user_num, factor_num * (2 ** (num_layers - 1))
        )
        self.embed_item_MLP = nn.Embedding(
            item_num, factor_num * (2 ** (num_layers - 1))
        )

        MLP_modules = []
        for i in range(num_layers):
            input_size = factor_num * (2 ** (num_layers - i))
            MLP_modules.append(nn.Dropout(p=self.dropout))
            MLP_modules.append(nn.Linear(input_size, input_size // 2))
            MLP_modules.append(nn.ReLU())
        self.MLP_layers = nn.Sequential(*MLP_modules)
        predict_size = factor_num
        self.predict_layer = nn.Linear(predict_size, 1)
        self._init_weight_()

    def _init_weight_(self):
        nn.init.normal_(self.embed_user_MLP.weight, std=0.01)
        nn.init.normal_(self.embed_item_MLP.weight, std=0.01)
        for m in self.MLP_layers:
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
        nn.init.kaiming_uniform_(self.predict_layer.weight, a=1, nonlinearity="sigmoid")

        for m in self.modules():
            if isinstance(m, nn.Linear) and m.bias is not None:
                m.bias.data.zero_()

    def forward(self, user, item):
        embed_user_MLP = self.embed_user_MLP(user)
        embed_item_MLP = self.embed_item_MLP(item)
        interaction = torch.cat((embed_user_MLP, embed_item_MLP), -1)
        output_MLP = self.MLP_layers(interaction)
        concat = output_MLP

        prediction = self.predict_layer(concat)
        return prediction.view(-1)
```

```
def create_model(user_num, item_num, factor_num=1):
    model = ModelClass(
    user_num,
    item_num,
    20,
    3,
    0.0,
    )

    loss_function = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001)
    return model, loss_function, optimizer

model, criterion, optimizer = create_model(610, 193609, 20)
model.cuda()

train_data = RecommendationDataset(f"{args.dataset}/ratings.csv", train=True)
train_data, validation_data = train_test_split(train_data,test_size=0.1)
train_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True)
validation_loader = DataLoader(validation_data, batch_size=args.batch_size, shuffle=True)


for epoch in range(5):
    cost = 0;
    for users, items, ratings in train_loader:
        users = users.cuda()
        items = items.cuda()
        ratings = ratings.float().cuda()

        optimizer.zero_grad()
        ratings_pred = model(users,items)
        loss = criterion(ratings_pred, ratings)
        loss.backward()
        optimizer.step()
        cost += loss.item()*len(ratings)
    cost/= 81676

    print(f"Epoch: {epoch}")
    print("train cost: {: .6f}".format(cost))

    cost_validation = 0;
    for users, items, ratings in validation_loader:
        users = users.cuda()
        items = items.cuda()
        ratings = ratings.float().cuda()

        ratings_pred = model(users,items)
        loss = criterion(ratings_pred, ratings)
        cost_validation += loss.item()*len(ratings)
    cost_validation/= 9076
    print("validation cost: {: .6f}".format(cost_validation))
```

1) Model

- **__init__**

User num, item, num, factor num을 입력 받는다. 각 항목은 앞서 matrix factorization에서 설명한 것과 같은 개념이다. Num layer는 사용할 layer의 개수이다. drop out은 각 layer간의 연결을 일부 랜덤하게 삭제하여 generalization 효과를 높이는 방법인데, 이번 실험에서는 0.0으로 두어 사용하지 않았다.

Self.embed_user_MLP : user와 factor의 number를 입력 받아 embedding 공간을 확보한다. 앞선 모델과 다르게 layer가 추가되었으므로 그 공간을 확보하기 위해서 factor에 제곱 연산을 더해준다.

Self.embed_item_MLP : item과 factor의 number를 입력 받아 embedding 공간을 확보한다.

For I in range(num_layers) : layer를 거치며 linear와 relu를 적용하고 학습을 진행한다. 각 layer가 진행되고 나서의 Output size는 입력된 데이터의 절반으로 설정한다. Self.predict_layer는 한 번 더 linear layer를 거치며 output 값을 1개로 만들어준다.

- **__init_weight__** : weight 값을 초기화한다.
- **forward**

user과 item data를 받아서 embedding 시킨다. 그 후 해당 값들을 torch.cat을 통해서 합친 후 model에 넣어 학습시킨다. 마지막으로 output size가 1이 되도록 predict layer를 한 번 더 거친 후 예측된 rate를 도출한다.

2) Training, validation

이 부분은 앞선 모델과 똑같이 작동한다.

3. 실험 결과

1) batch size, learning rate에 따른 변화

총 100 epoch동안 학습을 진행했다. batch size는 256과 16, 그리고 learning rate는 0.001과 0.0001을 사용하여 각 조합별로 실험을 진행했다. 노란색으로 표현된 부분은 각 실험에서 overfitting이 발생하기 직전 epoch이다. (Matrix factorization model을 이용해 Batch = 256, lr =0.0001로 실험한 결과와 batch = 16, lr = 0.0001으로 실험한 결과는 100 epoch동안 overfitting이 발생하지 않아서 표시하지 않음.)

- **Neural collaborative filtering**

batch = 256, lr = 0.001, layer = 3	batch = 256, lr = 0.0001, layer = 3	batch = 16, lr = 0.001, layer = 3	batch = 16, lr = 0.0001, layer = 3
Epoch: 0	Epoch: 0	Epoch: 0	Epoch: 0
train cost: 1.933604	train cost: 9.188219	train cost: 0.949247	train cost: 1.498390
validation cost: 0.797456	validation cost: 1.359087	validation cost: 0.794905	validation cost: 0.773444
Epoch: 1	Epoch: 1	Epoch: 1	Epoch: 1
train cost: 0.730679	train cost: 0.963511	train cost: 0.716432	train cost: 0.728031
validation cost: 0.767727	validation cost: 0.830284	validation cost: 0.798939	validation cost: 0.756557
Epoch: 2	Epoch: 2	Epoch: 2	Epoch: 2
train cost: 0.683523	train cost: 0.762483	train cost: 0.660362	train cost: 0.688585
validation cost: 0.746165	validation cost: 0.772346	validation cost: 0.824451	validation cost: 0.741477
Epoch: 3	Epoch: 3	Epoch: 3	Epoch: 3
train cost: 0.649503	train cost: 0.708396	train cost: 0.617623	train cost: 0.659144
validation cost: 0.747724	validation cost: 0.757405	validation cost: 0.771017	validation cost: 0.740328
Epoch: 4	Epoch: 4	Epoch: 4	Epoch: 4
train cost: 0.623753	train cost: 0.682321	train cost: 0.571453	train cost: 0.636939
validation cost: 0.751966	validation cost: 0.756733	validation cost: 0.792156	validation cost: 0.736623
Epoch: 5	Epoch: 5	Epoch: 5	Epoch: 5
train cost: 0.596798	train cost: 0.667662	train cost: 0.523587	train cost: 0.616654
validation cost: 0.768917	validation cost: 0.756269	validation cost: 0.799653	validation cost: 0.737968
Epoch: 6	Epoch: 6	Epoch: 6	Epoch: 6
train cost: 0.571379	train cost: 0.658296	train cost: 0.476280	train cost: 0.595661
validation cost: 0.779272	validation cost: 0.760576	validation cost: 0.826839	validation cost: 0.744587

- **Matrix factorization**

batch = 256, lr = 0.001	batch = 256, lr = 0.0001	batch = 16, lr = 0.0001	batch=16, lr = 0.001
Epoch: 0	Epoch: 0	Epoch: 0	Epoch: 0
train cost: 12.671981	train cost: 13.353680	train cost: 13.335752	train cost: 7.420529
validation cost: 10.424196	validation cost: 13.293840	validation cost: 13.133173	validation cost: 2.905596
Epoch: 1	Epoch: 1	Epoch: 1	Epoch: 1
train cost: 6.756295	train cost: 13.351149	train cost: 12.739705	train cost: 1.887929
validation cost: 4.222849	validation cost: 13.288225	validation cost: 11.993493	validation cost: 1.708483
Epoch: 2	Epoch: 2	Epoch: 2	Epoch: 2
train cost: 3.056362	train cost: 13.328678	train cost: 11.138743	train cost: 1.205658
validation cost: 2.634501	validation cost: 13.240261	validation cost: 10.072340	validation cost: 1.434705
Epoch: 3	Epoch: 3	Epoch: 3	Epoch: 3
train cost: 2.008648	train cost: 13.230002	train cost: 8.979123	train cost: 0.983560
validation cost: 2.038181	validation cost: 13.088157	validation cost: 7.878173	validation cost: 1.336263
Epoch: 4	Epoch: 4	Epoch: 4	Epoch: 4
train cost: 1.527972	train cost: 13.016163	train cost: 6.841769	train cost: 0.874533
validation cost: 1.732206	validation cost: 12.825485	validation cost: 6.043956	validation cost: 1.294889
Epoch: 5	Epoch: 5	Epoch: 5	Epoch: 5
train cost: 1.255579	train cost: 12.700290	train cost: 5.258344	train cost: 0.804910
validation cost: 1.555256	validation cost: 12.476117	validation cost: 4.849616	validation cost: 1.265027
Epoch: 6	Epoch: 6	Epoch: 6	Epoch: 6
train cost: 1.084269	train cost: 12.306502	train cost: 4.229207	train cost: 0.754124
validation cost: 1.443664	validation cost: 12.061761	validation cost: 4.066694	validation cost: 1.253214
Epoch: 7	Epoch: 7	Epoch: 7	Epoch: 7
train cost: 0.968904	train cost: 11.853382	train cost: 3.521519	train cost: 0.706721
validation cost: 1.368630	validation cost: 11.597428	validation cost: 3.505086	validation cost: 1.251500
Epoch: 8	Epoch: 8	Epoch: 8	Epoch: 8
train cost: 0.887131	train cost: 11.355660	train cost: 3.000380	train cost: 0.665354
validation cost: 1.317651	validation cost: 11.096665	validation cost: 3.082881	validation cost: 1.241978
Epoch: 9	Epoch: 9	Epoch: 9	Epoch: 9
train cost: 0.828443	train cost: 10.825385	train cost: 2.605722	train cost: 0.623047
validation cost: 1.280593	validation cost: 10.569200	validation cost: 2.761785	validation cost: 1.237259
Epoch: 10	Epoch: 10	Epoch: 10	Epoch: 10
train cost: 0.784856	train cost: 10.271918	train cost: 2.301456	train cost: 0.579808
validation cost: 1.254135	validation cost: 10.023586	validation cost: 2.513070	validation cost: 1.236612
Epoch: 11	Epoch: 11	Epoch: 11	Epoch: 11
train cost: 0.752006	train cost: 9.703278	train cost: 2.063161	train cost: 0.535592
validation cost: 1.238795	validation cost: 9.467766	validation cost: 2.317502	validation cost: 1.237691
Epoch: 12	Epoch: 12	Epoch: 12	Epoch: 12
train cost: 0.725932	train cost: 9.127138	train cost: 1.873229	train cost: 0.491326
validation cost: 1.223474	validation cost: 8.909400	validation cost: 2.161245	validation cost: 1.250012
Epoch: 13	Epoch: 13	Epoch: 13	Epoch: 13
train cost: 0.705917	train cost: 8.552170	train cost: 1.719321	train cost: 0.449199
validation cost: 1.215400	validation cost: 8.355619	validation cost: 2.033821	validation cost: 1.261986
Epoch: 14	Epoch: 14	Epoch: 14	Epoch: 14
train cost: 0.690080	train cost: 7.984759	train cost: 1.592275	train cost: 0.410903
validation cost: 1.208421	validation cost: 7.813210	validation cost: 1.928459	validation cost: 1.273599
Epoch: 15	Epoch: 15	Epoch: 15	Epoch: 15
train cost: 0.677372	train cost: 7.431103	train cost: 1.486396	train cost: 0.376832
validation cost: 1.203849	validation cost: 7.287185	validation cost: 1.840772	validation cost: 1.286981
Epoch: 16	Epoch: 16	Epoch: 16	Epoch: 16
train cost: 0.667051	train cost: 6.897105	train cost: 1.397331	train cost: 0.346553
validation cost: 1.200807	validation cost: 6.783407	validation cost: 1.766593	validation cost: 1.304662
Epoch: 17	Epoch: 17	Epoch: 17	Epoch: 17
train cost: 0.658247	train cost: 6.387813	train cost: 1.321460	train cost: 0.321120
validation cost: 1.200026	validation cost: 6.306435	validation cost: 1.703129	validation cost: 1.314743
Epoch: 18	Epoch: 18	Epoch: 18	Epoch: 18
train cost: 0.650689	train cost: 5.907631	train cost: 1.256436	train cost: 0.297739
validation cost: 1.199517	validation cost: 5.860367	validation cost: 1.649401	validation cost: 1.351465
Epoch: 19	Epoch: 19	Epoch: 19	Epoch: 19
train cost: 0.643809	train cost: 5.460565	train cost: 1.200066	train cost: 0.279536
validation cost: 1.197073	validation cost: 5.448492	validation cost: 1.603009	validation cost: 1.357078
Epoch: 20	Epoch: 20	Epoch: 20	Epoch: 20
train cost: 0.637726	train cost: 5.049408	train cost: 1.150972	train cost: 0.262616
validation cost: 1.195604	validation cost: 5.072996	validation cost: 1.563300	validation cost: 1.384563
Epoch: 21	Epoch: 21	Epoch: 21	Epoch: 21
train cost: 0.631012	train cost: 4.675683	train cost: 1.108257	train cost: 0.248061
validation cost: 1.197200	validation cost: 4.734711	validation cost: 1.527887	validation cost: 1.403037
Epoch: 22	Epoch: 22	Epoch: 22	Epoch: 22
train cost: 0.624784	train cost: 4.339729	train cost: 1.070351	train cost: 0.236409
validation cost: 1.196106	validation cost: 4.432772	validation cost: 1.496967	validation cost: 1.420619
Epoch: 23	Epoch: 23	Epoch: 23	Epoch: 23
train cost: 0.618316	train cost: 4.040521	train cost: 1.036946	train cost: 0.226159
validation cost: 1.194786	validation cost: 4.165830	validation cost: 1.470619	validation cost: 1.437620
Epoch: 24	Epoch: 24	Epoch: 24	Epoch: 24
train cost: 0.611004	train cost: 3.775495	train cost: 1.007152	train cost: 0.216586
validation cost: 1.195214	validation cost: 3.930819	validation cost: 1.447295	validation cost: 1.441853
Epoch: 25	Epoch: 25	Epoch: 25	Epoch: 25
train cost: 0.603672	train cost: 3.541775	train cost: 0.980641	train cost: 0.208386
validation cost: 1.192977	validation cost: 3.723795	validation cost: 1.426190	validation cost: 1.466572
Epoch: 26	Epoch: 26	Epoch: 26	Epoch: 26
train cost: 0.596139	train cost: 3.335047	train cost: 0.956681	train cost: 0.202189
validation cost: 1.192688	validation cost: 3.541252	validation cost: 1.407446	validation cost: 1.481012

2) Layer에 따른 변화 (Neural collaborative filtering)

Neural collaborating filtering 방법에서 layer를 2,3,5,6개를 사용하여 각각 실험해보았다. Layer를 제외한 다른 hyperparameter들은 동일하게 정했다.

layer = 2	layer = 3	layer = 5	layer = 6
Epoch: 0	Epoch: 0	Epoch: 0	Epoch: 0
train cost: 1.899851	train cost: 1.601068	train cost: 1.131630	train cost: 1.022904
validation cost: 0.774361	validation cost: 0.766832	validation cost: 0.795259	validation cost: 0.788928
Epoch: 1	Epoch: 1	Epoch: 1	Epoch: 1
train cost: 0.731163	train cost: 0.731246	train cost: 0.718023	train cost: 0.718400
validation cost: 0.762089	validation cost: 0.748556	validation cost: 0.736056	validation cost: 0.767183
Epoch: 2	Epoch: 2	Epoch: 2	Epoch: 2
train cost: 0.702711	train cost: 0.698243	train cost: 0.662280	train cost: 0.649652
validation cost: 0.758770	validation cost: 0.743559	validation cost: 0.739028	validation cost: 0.761116
Epoch: 3	Epoch: 3	Epoch: 3	Epoch: 3
train cost: 0.685708	train cost: 0.670571	train cost: 0.612829	train cost: 0.556234
validation cost: 0.755797	validation cost: 0.732635	validation cost: 0.741299	validation cost: 0.773273
Epoch: 4	Epoch: 4		
train cost: 0.669914	train cost: 0.648057		
validation cost: 0.749773	validation cost: 0.727645		
Epoch: 5	Epoch: 5		
train cost: 0.654755	train cost: 0.630220		
validation cost: 0.747814	validation cost: 0.736853		
Epoch: 6			
train cost: 0.642935			
validation cost: 0.741014			
Epoch: 7			
train cost: 0.632592			
validation cost: 0.743734			

4. 사용할 Hyperparameters와 model 결정

- Training set, validation set 비율.

Training set은 model을 실질적으로 학습시키는 데 이용되는 데이터이고, validation set은 학습과정에서 overfitting이 발생하지 않도록 검증하기 위해 사용되는 데이터이다. Training set이 많을수록 model 학습에 도움이 될 것이지만, validation set이 너무 작다면 overfitting을 제대로 검증해내지 못하는 단점이 존재한다. 이번 실험에 제공된 데이터는 약 9만개로, validation set의 비율이 작아도 충분히 overfitting을 검증해 낼 수 있다는 판단이 들어서 training set과 validation set의 비율을 9:1로 설정하게 되었다.

- Batch size, lr

batch size와 learning rate는 각각 아래와 같은 특성을 지닌다.

Batch size ↑ : 한 번에 학습하는 데이터의 양이 많다. 업데이트가 느리다. Memory가 많이 사용된다. 비교적 Overfitting의 위험이 있다.

Batch size ↓ : 한 번에 학습하는 데이터의 양이 적다. 업데이트가 빠르다. Local minimum으로 빠질 위험이 있다.

Learning rate ↑ : 많이 학습하기 때문에 학습이 빠르다. Overshooting이 발생할 위험이 있다.

Learning rate ↓ : 조금씩 학습하기 때문에 학습이 느리다. Local minimum으로 빠질 위험이 있다.

큰 lr과 작은 batch size를 사용할 경우, 적은 데이터로 많은 학습을 해야 하므로 수렴이 힘들 것이고, 큰 batch size와 작은 lr을 사용할 경우, 많은 데이터로 조금씩 학습해 나가기 때문에 overfitting이 일어날 위험이 있다. 작은 batch size와 작은 lr을 사용할 경우, 적은 데이터를 적게 씩 학습해 나가기 때문에 학습이 매우 느릴 것이고 local minimum으로 빠질 위험이 있다.

위와 같이 learning rate와 batch size는 서로 연관관계가 있다. 앞선 실험에서도 해당 관계를 바탕으로 작은 batch, 큰 batch, 작은 lr, 큰 lr을 각각 조합하여 진행해 보았다. 사용한 parameter 값은 batch = 16, batch = 256, lr = 0.001, lr = 000.1이다.

실험 결과를 보면, overfitting이 발생하기 전에 가장 작은 validation cost를 보여준 batch size와 lr 조합은, **matrix factorization**에서는 batch size = 256, learning rate= 0.001, **neural collaborative filtering**에서는 batch = 16, lr = 0.0001 것으로 확인했다. 따라서 최종적으로 각 model 별로 가장 작은 validation cost를 보여준 해당 batch, lr 조합을 선택하여 사용하기로 결정했다.

- Layer (NCF)

Neural collaborative filtering 방법에서 layer를 2,3,5,6으로 잡고 실험해보았다. Layer를 7 이상으로 설정하게 되면 memory의 부족으로 프로그램이 실행되지 않았기에 실험하지 못했다.

먼저 큰 layer를 사용한 실험 결과를 살펴보면 학습이 빠르게 진행되는 것을 확인할 수 있었다. Layer를 5~6으로 설정했을 경우를 보면, 학습이 빠르게 진행되어서 1~2 epoch에서 minimum validation cost가 나왔다. 하지만 학습으로 도출된 minimum validation cost가 작은 layer를 선택했을 때에 비해서 비교적 컸다. 다음으로 작은 layer를 사용한 실험 결과를 살펴보면, 학습이 비교적 천천히 진행된다. Layer를 3으로 설정했을 때의 결과를 보면 학습이 천천히 진행되는 것을 확인할 수 있다. 또한 layer를 3으로 설정했을 때 모든 경우 중에서 가장 작은 validation cost를 보여주었다.

각 경우를 살펴보았을 때, training cost는 layer를 쌓을수록 더 개선되는 모습을 보여줬지만, validation cost는 layer를 3으로 설정했을 때 가

장 작게 도출되는 것을 확인할 수 있었다. 또한 가장 안정적으로 학습이 진행되는 것 역시 확인했다. 따라서 neural collaborative filterin 모델에서 layer는 3개를 사용하기로 결정했다.

- **Epoch**

먼저 각 model들을 100epoch를 이용하여 학습해 보았다. 100 epoch까지 학습을 완료하고 training loss와 validation loss를 각각 살펴보면, training loss는 꾸준히 감소하는 추세를 보였지만, validation loss는 감소하다 어느 순간부터 더 이상 감소하지 않고 오히려 증가하는 모습을 보였다. 이는 학습이 진행될수록 training data set에 대한 overfitting이 발생하기 때문이다.

앞서 각 model에서 가장 작은 validation cost를 보여줬던 파라미터들을 적용한 실험을 보면 matrix factorization은 27 epoch부터 overfitting이 발생했고, neural collaborating filtering은 5 epoch에서부터 overfitting이 발생했다. 따라서 각 model별로 epoch를 27, 5로 설정하는 것이 최적이라고 판단했다.

- **Model**

실험 결과를 보면, 전체적으로 matrix factorization보다 neural collaborative filtering에서 cost가 작게 도출되었고, minimum validation cost까지 도달하는 데에 걸린 epoch와 시간도 확연히 적었다. 또한 가장 좋은 결과를 보여준 validation cost를 살펴보면 matrix factorization은 1.192668, 그리고 neural collaborative filtering은 0.736623이다. 이와 같이 neural collaborative filtering 모델이 시간적으로도, validation cost로도 더 나은 성능을 보여주는 것을 확인했기 때문에 최종적으로 neural collaborative filtering을 이용하기로 결정했다.

- **최종적으로 사용할 model과 hyperparameters**

Model : Neural Collaborative Filtering
Training set, validation set의 비율 : 9:1
Batch size : 16
Learning rate : 0.0001
Epoch : 5회
Layer : 3개

5. 최종 Test 결과

Test data	result of 'Run.py'
userId,itemId	
500,1304	3.9252796173095703
588,1961	3.017786979675293
558,58559	4.305431365966797
307,73	3.434276580810547
181,2915	3.2946062088012695
382,36525	1.8510844707489014
552,1961	3.6994216442108154
528,543	3.0516793727874756
156,5989	4.185009479522705
431,5308	3.1623170375823975

6. References

1. IbrahemKandel, The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset https://www.sciencedirect.com/science/article/pii/S2405959519303455#fig2
2. Dohee's ML Lab, Neural Collaborative Filtering - MLP 실험 https://doheelab.github.io/recommender-system/ncf_mlp/
3. Dohee's ML Lab, Pytorch를 이용한 협업 필터링(Matrix Factorization) 구현 https://doheelab.github.io/recommender-system/ncf_mf/
4. Kung-Hsiang, Huang, Neural Collaborative Filtering Explanation & Implementation https://towardsdatascience.com/paper-review-neural-collaborative-filtering-explanation-implementation-ea3e031b7f96
5. Xiangnan He, Neural Collaborative Filtering* https://arxiv.org/pdf/1708.05031.pdf