CS 584 PyTorch Tutorial 2

Jason Choi

https://github.com/boslbi92/pytorch_tutorial

Agenda

1. Neural Collaborative Filtering (NCF)
    - Movie Ratings Prediction

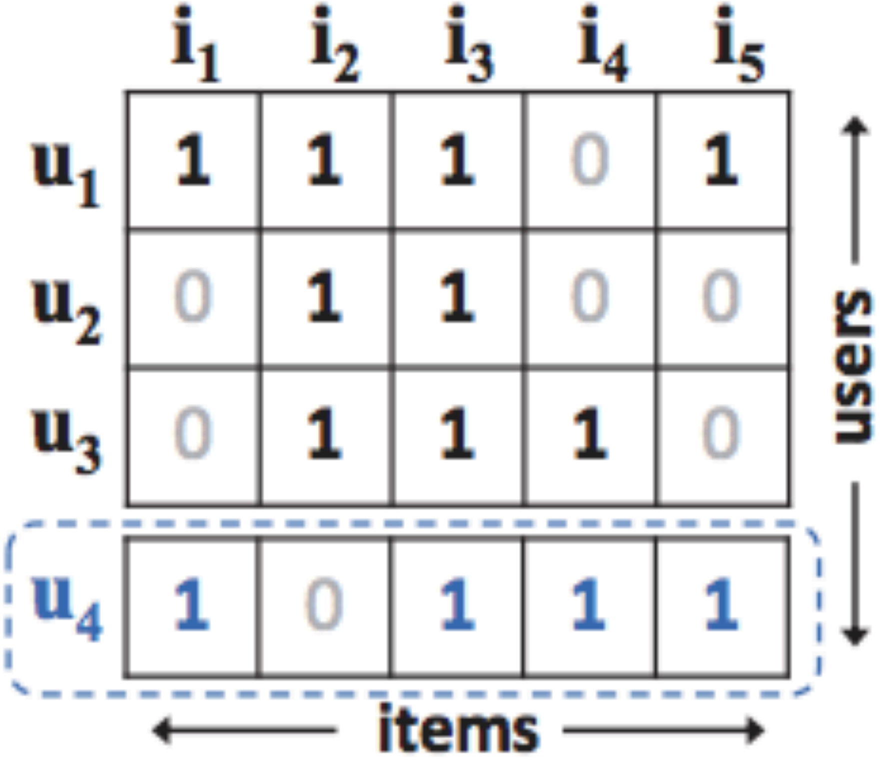
2. Implementing NCF with PyTorch

3. Coding tutorial

Movie Ratings Prediction

1. MovieLens dataset
   - **[user_id, item_id, rating]**
   - How to handle sparse input?

2. Neural collaborative filtering
   - $f(V_{user}, V_{item}) \rightarrow rating$



(a) user–item matrix

Neural Collaborative Filtering

1. Multi-layer perceptron (MLP)
   - Model interactions by non-linear transformations

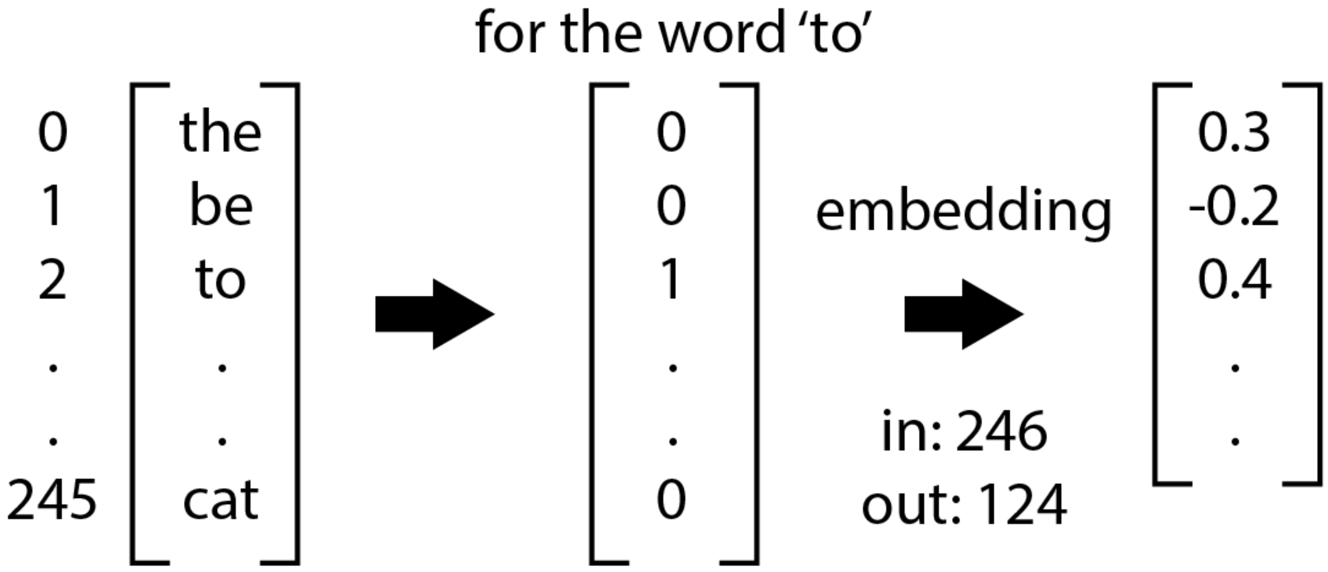
2. Generalized matrix factorization (GMF)
   - Model interactions by learning latent user and item vectors

3. Neural collaborative filtering (NCF)
   - Concatenate outputs form first & second modules

Multi-layer Perceptron

- Represent user → V_{user}
- Represent item → V_{item}
- Represent interaction → $[V_{user}; V_{item}]$
- Apply nonlinear transformations
  - N-layers of nonlinear transformations to model interaction

Use of Embeddings Layer



Assign random vectors to each user and item

- Represent user → V_{user}
- Represent item → V_{item}
- Represent interaction → $[V_{user}; V_{item}]$

Multi-layer Perceptron

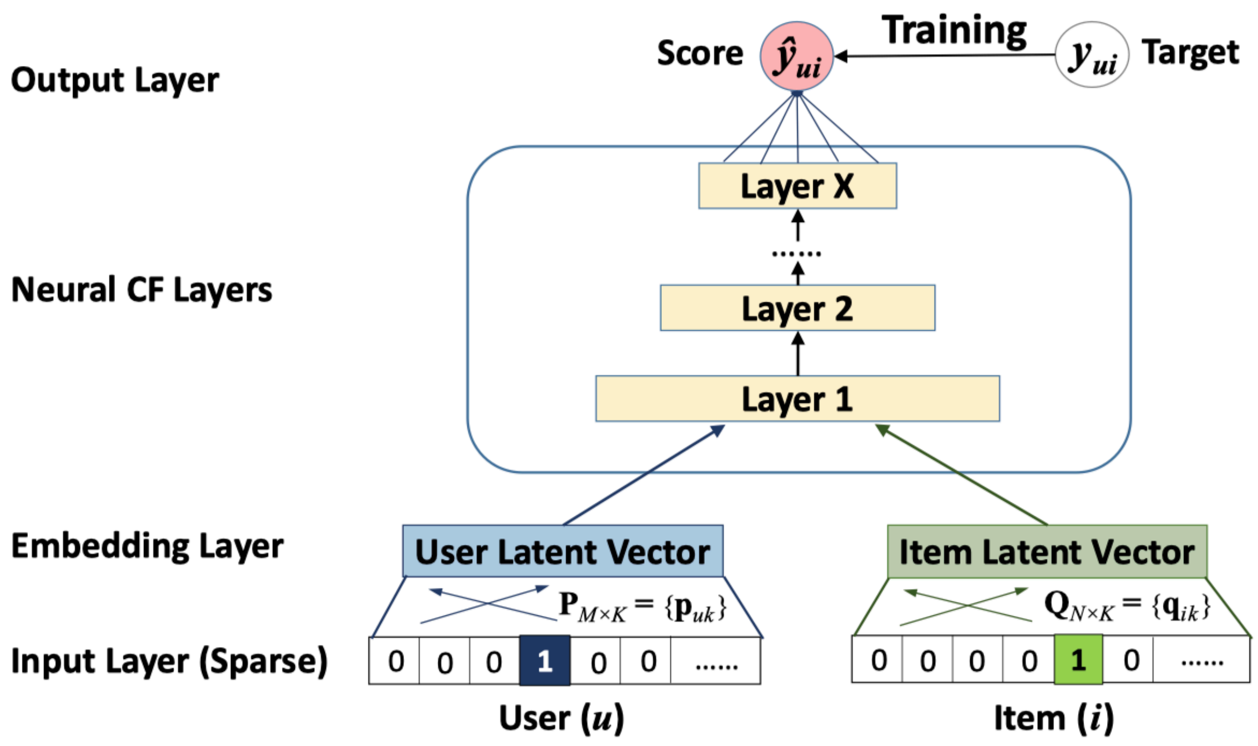


Figure 2: Neural collaborative filtering framework

1. User embedding layers
   - (num_user, latent_dim)
   - User x → x^{th} row of embedding matrix
2. Item embedding layers
   - (num_item, latent_dim)
   - Item y → y^{th} row of embedding matrix
3. 3-layered MLP
4. Predict scalar value

How to improve?

Generalized Matrix Factorization

- Learn good p_u and q_i embeddings to explain interactions

$$\hat{y}_{ui} = f(u, i|\mathbf{p}_u, \mathbf{q}_i) = \mathbf{p}_u^T \mathbf{q}_i = \sum_{k=1}^{K} p_{uk} q_{ik},$$

p_u = latent vector for a user
q_i = latent vector for an item
$\hat{y}_{ui}$= inner products of p, q

Neural Collaborative Filtering

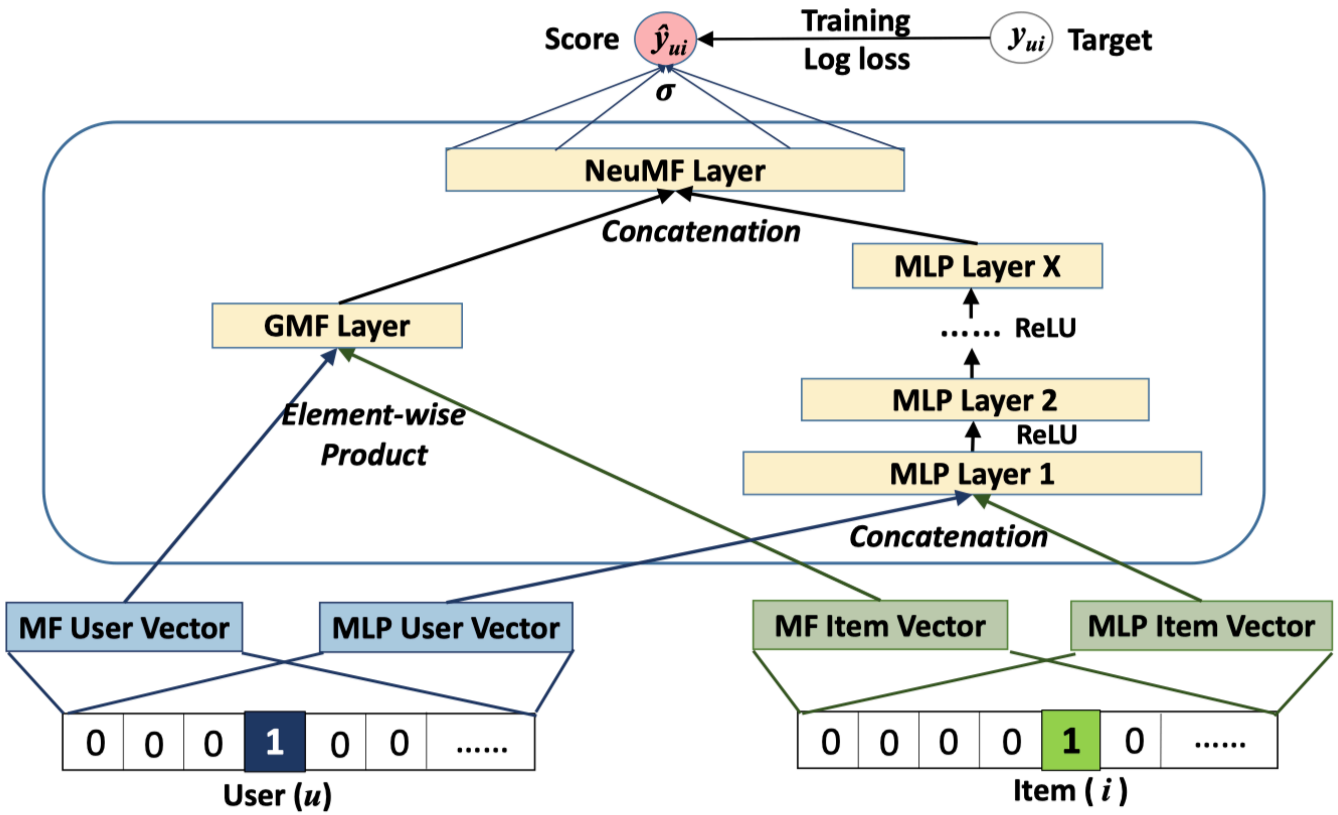


Figure 3: Neural matrix factorization model

1. GMF module
2. MLP module
3. Combine two modules
4. Predict scalar value

Agenda

1. Neural Collaborative Filtering (NCF)
   - Movie Ratings Prediction

2. Implementing NCF with PyTorch

3. Coding tutorial

Step 1:
Prepare Inputs

- Convert dataset into tensor
- Initialize DataLoader class

```
# convert input to torch tensors
train_user = torch.tensor(train_data['user'].values,
                          device=device)
train_item = torch.tensor(train_data['item'].values,
                          device=device)
train_rating = torch.tensor(train_data['rating'].values,
                            device=device,
                            dtype=torch.float)

# convert tensors to dataloader
train_dataset = data.TensorDataset(train_user,
                                   train_item,
                                   train_rating)
train_loader = data.DataLoader(train_dataset,
                               batch_size=batch_size,
                               shuffle=True)
```

Step 2:
MLP Module

- Initialize user embeddings
- Initialize item embeddings

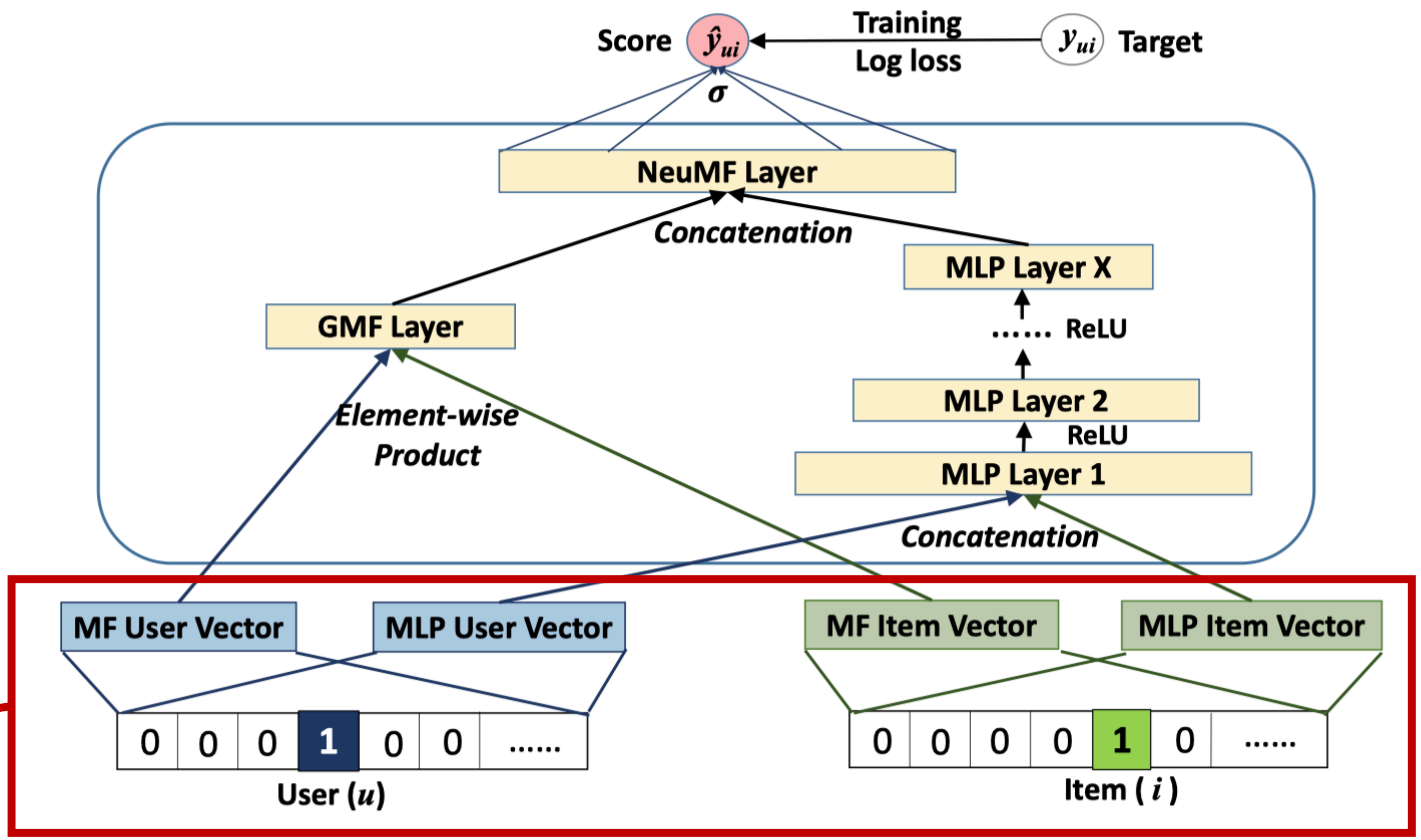


Figure 3: Neural matrix factorization model

```
# multilayer perceptron sub-module
class MLP(nn.Module):
  def __init__(self, user_num, item_num, output_dim):
    super(MLP, self).__init__()
    self.user_emb = nn.Embedding(user_num, output_dim * 4)
    self.item_emb = nn.Embedding(item_num, output_dim * 4)
```

Step 2: MLP Module

- Initialize linear layers
- 1st layer → 128
- 2nd layer → 64
- 3rd layer → 32

```
# multilayer perceptron sub-module
class MLP(nn.Module):
    def __init__(self, user_num, item_num, output_dim):
        super(MLP, self).__init__()
        self.user_emb = nn.Embedding(user_num, output_dim * 4)
        self.item_emb = nn.Embedding(item_num, output_dim * 4)

        self.linear_1 = nn.Linear(output_dim * 8, output_dim * 4)
        self.linear_2 = nn.Linear(output_dim * 4, output_dim * 2)
        self.linear_3 = nn.Linear(output_dim * 2, output_dim)
        self.dropout = nn.Dropout()
        self.relu = nn.ReLU()
```

Step 2:
MLP Module

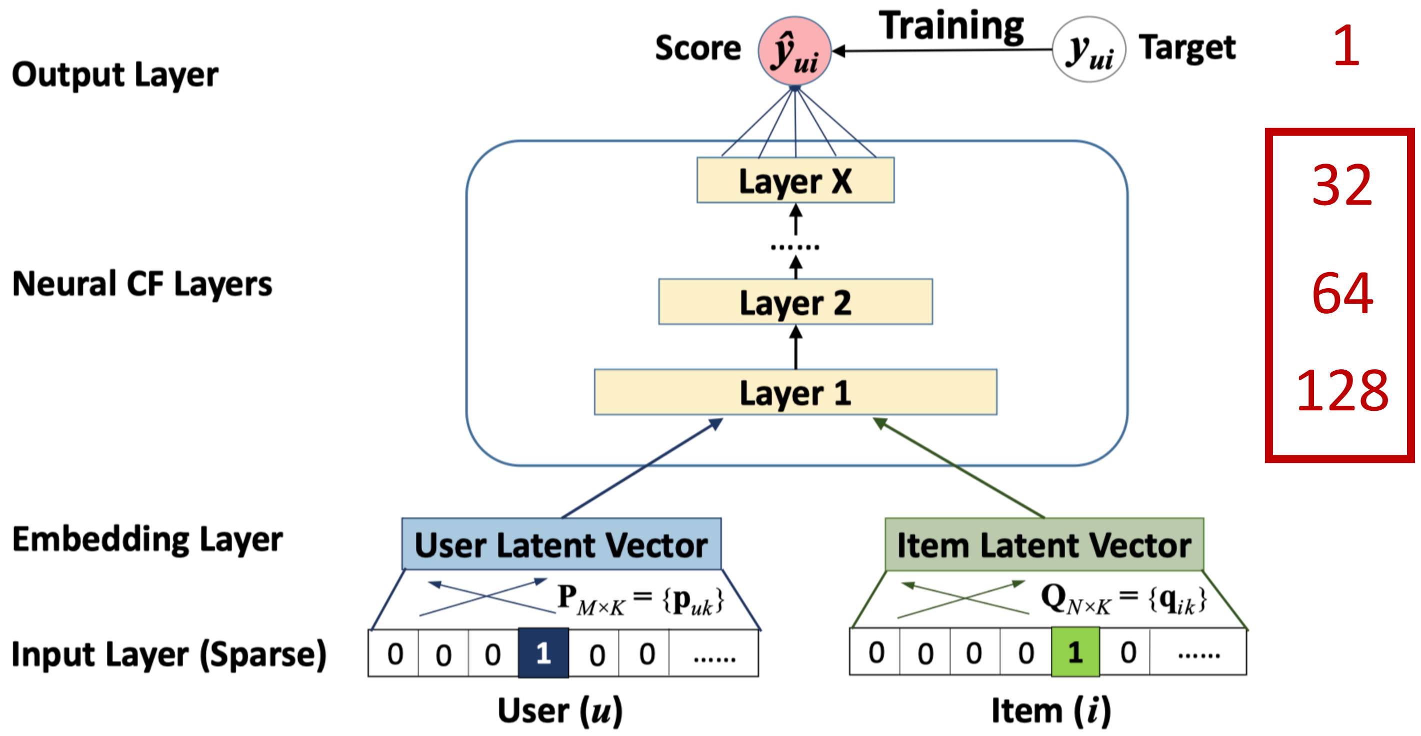


Figure 2: Neural collaborative filtering framework

Step 2: MLP Module

- Represent user-item interaction
- Three matrix multiplications

```python
def forward(self, user, item):
    user_emb = self.user_emb(user)
    item_emb = self.item_emb(item)
    concat = torch.cat((user_emb, item_emb), -1)

    x = self.linear_1(concat)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.linear_2(x)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.linear_3(x)
    x = self.relu(x)
    return x
```

Step 3: GMF Module

- Initialize user embeddings
- Initialize item embeddings

```python
# generalized matrix factorization sub-module
class GMF(nn.Module):
    def __init__(self, user_num, item_num, output_dim):
        super(GMF, self).__init__()
        self.user_emb = nn.Embedding(user_num, output_dim)
        self.item_emb = nn.Embedding(item_num, output_dim)
        self._init_weight_()

    def forward(self, user, item):
        user_emb = self.user_emb(user)
        item_emb = self.item_emb(item)
        output = user_emb * item_emb
        return output
```

$$\hat{y}_{ui} = f(u, i|\mathbf{p}_u, \mathbf{q}_i) = \mathbf{p}_u^T \mathbf{q}_i$$

Step 3:
GMF Module

- Initialize user embeddings
- Initialize item embeddings

```python
# generalized matrix factorization sub-module
class GMF(nn.Module):
    def __init__(self, user_num, item_num, output_dim):
        super(GMF, self).__init__()
        self.user_emb = nn.Embedding(user_num, output_dim)
        self.item_emb = nn.Embedding(item_num, output_dim)
        self._init_weight_()

    def forward(self, user, item):
        user_emb = self.user_emb(user)
        item_emb = self.item_emb(item)
        output = user_emb * item_emb
        return output
```

$$\hat{y}_{ui} = f(u, i | \boxed{\mathbf{p}_u, \mathbf{q}_i}) = \mathbf{p}_u^T \mathbf{q}_i$$

Step 3: GMF Module

- Initialize user embeddings
- Initialize item embeddings
- Output is product of two latent vectors

```python
# generalized matrix factorization sub-module
class GMF(nn.Module):
    def __init__(self, user_num, item_num, output_dim):
        super(GMF, self).__init__()
        self.user_emb = nn.Embedding(user_num, output_dim)
        self.item_emb = nn.Embedding(item_num, output_dim)
        self._init_weight_()

    def forward(self, user, item):
        user_emb = self.user_emb(user)
        item_emb = self.item_emb(item)
        output = user_emb * item_emb
        return output
```

$$\hat{y}_{ui} = f(u, i | \mathbf{p}_u, \mathbf{q}_i) = \boxed{\mathbf{p}_u^T \mathbf{q}_i}$$

Step 4:
NCF Module

- Concatenate outputs from two modules
- Predict scalar values for each input

```
# full neural collaborative filtering module
class NCF(nn.Module):
  def __init__(self, user_num, item_num, output_dim):
    super(NCF, self).__init__()
    self.GMF_layers = GMF(user_num=user_num,
                          item_num=item_num,
                          output_dim=output_dim)
    self.MLP_layers = MLP(user_num=user_num,
                          item_num=item_num,
                          output_dim=output_dim)
    self.predict_layer = nn.Linear(output_dim * 2, 1)
    self.dropout = nn.Dropout()

  def forward(self, user, item):
    # GMF layers
    output_GMF = self.GMF_layers(user, item)

    # MLP layers
    output_MLP = self.MLP_layers(user, item)

    # merge outputs
    concat = torch.cat((output_GMF, output_MLP), -1)
    concat = self.dropout(concat)
    prediction = self.predict_layer(concat).view(-1)
    return prediction
```

Step 4: NCF Module

- Concatenate outputs from two modules
- Predict scalar values for each input

```
# full neural collaborative filtering module
class NCF(nn.Module):
  def __init__(self, user_num, item_num, output_dim):
    super(NCF, self).__init__()
    self.GMF_layers = GMF(user_num=user_num,
                          item_num=item_num,
                          output_dim=output_dim)
    self.MLP_layers = MLP(user_num=user_num,
                          item_num=item_num,
                          output_dim=output_dim)
    self.predict_layer = nn.Linear(output_dim * 2, 1)
    self.dropout = nn.Dropout()

  def forward(self, user, item):
    # GMF layers
    output_GMF = self.GMF_layers(user, item)

    # MLP layers
    output_MLP = self.MLP_layers(user, item)

    # merge outputs
    concat = torch.cat((output_GMF, output_MLP), -1)
    concat = self.dropout(concat)
    prediction = self.predict_layer(concat).view(-1)
    return prediction
```

Step 5:
Training Loop

- Initialize model
- Initialize loss and optimizer
- Compute MSE loss and update params

```
model = NCF(user_num, item_num, output_dim)
model = model.to(device)
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

for epoch in range(1, 20):
  # training loop
  start_time = time.time()
  model.train()
  for user, item, label in tqdm(train_loader, total=len(tra
    user = user.to(device)
    item = item.to(device)
    label = label.float().to(device)
    model.zero_grad()
    prediction = model(user, item)
    loss = loss_function(prediction, label)
    loss.backward()
    optimizer.step()
```

Step 6: Evaluation

- Remember the set model to eval mode
- Compute desired metrics such as MAP

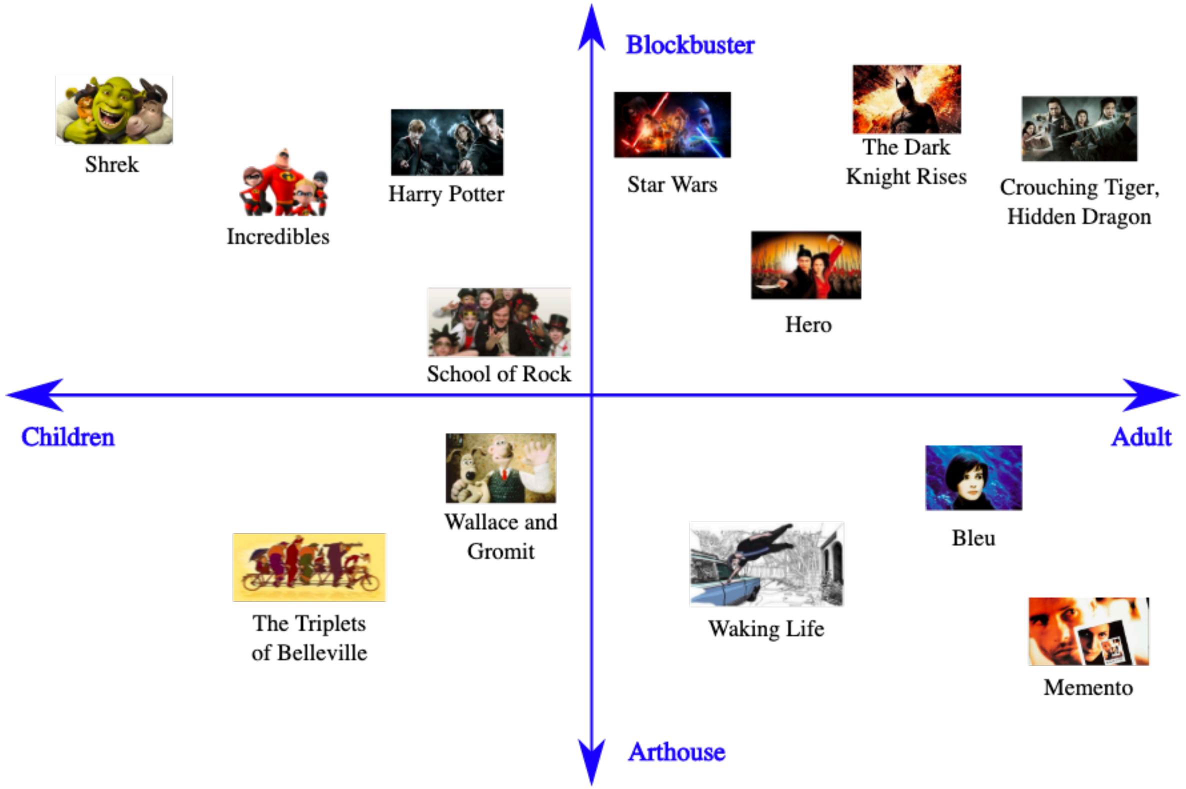
```
# eval loop
test_mae = []
model.eval()
for user, item, label in test_loader:
    user = user.to(device)
    item = item.to(device)
    prediction = model(user, item)
    label = label.float().detach().cpu().numpy()
    prediction = prediction.float().detach().cpu().numpy()
    MAE = mean_absolute_error(y_pred=prediction, y_true=label)
    test_mae.append(MAE)
```

Recommendation

- Using NCF, we trained a model to directly predict ratings of (user, item) pairs
- Predict ratings of unobserved items
  - Can be used as a ranking function
  - High rating can be potentially similar items

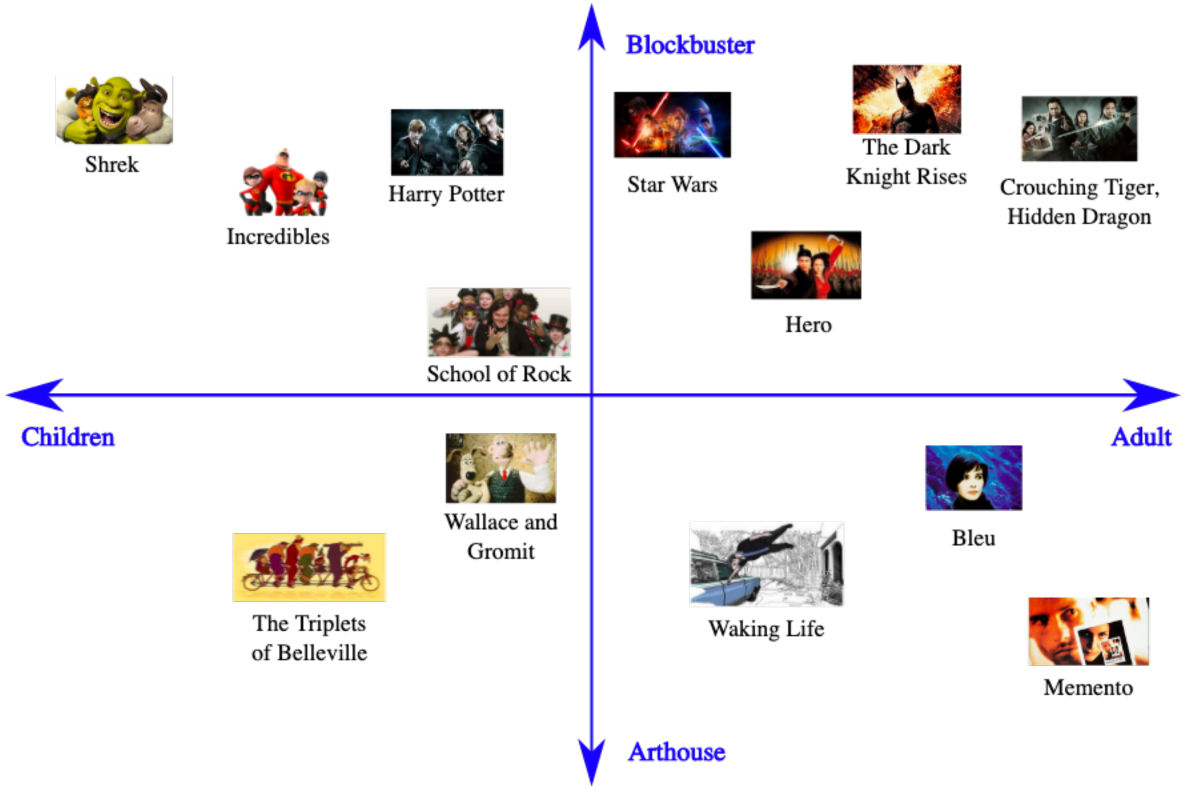
Item Similarity

- Projected one-hot item vectors into dense embeddings
- Contain notion of similarity
- Dimensionality reduction algorithms can be applied for visualization



Obtaining Weights of PyTorch Layer

- model.parameters()
- model.layer.weight
- Convert to numpy array and apply T-SNE or PCA



Coding Exercise

- https://github.com/boslbi92/pytorch_tutorial
- Implement class simpleCF inside practice.py
- Solution is available at solution.py