1 News

2 DiscussionGroup
  2.1 Quick-Start ................................................................. 3
  2.2 Features ................................................................. 5
  2.3 Examples ............................................................... 7
  2.4 FAQ ................................................................. 23
  2.5 History ............................................................... 28
  2.6 DeepCTR-Torch Models API ........................................... 29
  2.7 DeepCTR-Torch Layers API ........................................... 30
  2.8 deepctr_torch.callbacks module ................................... 47

3 Indices and tables ................................................................. 61

Python Module Index ............................................................... 63

Index ................................................................. 65
DeepCTR-Torch is an Easy-to-use, Modular and Extendible package of deep-learning based CTR models along with lots of core components layer which can be used to build your own custom model easily. It is compatible with PyTorch. You can use any complex model with `model.fit()` and `model.predict()`.

Let's Get Started! (Chinese Introduction)

You can read the latest code at https://github.com/shenweichen/DeepCTR-Torch and DeepCTR for tensorflow version.
CHAPTER 1

News

12/05/2020: Improve compatibility & fix issues. Add History callback (example). Changelog
10/18/2020: Add DCN-M and DCN-Mix. Add EarlyStopping and ModelCheckpoint callbacks (example). Changelog
10/09/2020: Improve the reproducibility & fix some bugs. Changelog
CHAPTER 2

DisscussionGroup

wechat ID: deepctrbot

2.1 Quick-Start

2.1.1 Installation Guide

depetcrt-torch depends on torch>=1.1.0, you can specify to install it through pip.

$ pip install -U deepctr-torch

2.1.2 Getting started: 4 steps to DeepCTR-Torch
Step 1: Import model

```python
import pandas as pd
import torch
from sklearn.metrics import log_loss, roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from deepctr_torch.inputs import SparseFeat, DenseFeat, get_feature_names

data = pd.read_csv('./criteo_sample.txt')
sparse_features = ['C' + str(i) for i in range(1, 27)]
dense_features = ['I' + str(i) for i in range(1, 14)]
data[sparse_features] = data[sparse_features].fillna('-1', )
data[dense_features] = data[dense_features].fillna(0, )
target = ['label']
```

Step 2: Simple preprocessing

Usually there are two simple way to encode the sparse categorical feature for embedding

- **Label Encoding**: map the features to integer value from 0 ~ len(#unique) - 1

```python
for feat in sparse_features:
lbe = LabelEncoder()
data[feat] = lbe.fit_transform(data[feat])
```

- **Hash Encoding**: Currently not supported.

And for dense numerical features, they are usually discretized to buckets, here we use normalization.

```python
mms = MinMaxScaler(feature_range=(0,1))
data[dense_features] = mms.fit_transform(data[dense_features])
```

Step 3: Generate feature columns

For sparse features, we transform them into dense vectors by embedding techniques. For dense numerical features, we concatenate them to the input tensors of fully connected layer.

- **Label Encoding**

```python
fixlen_feature_columns = [SparseFeat(feat, vocabulary_size=data[feat].nunique(), embedding_dim=4) for i,feat in enumerate(sparse_features)] + [DenseFeat(feat, 1, ) for feat in dense_features]
```

- **Feature Hashing on the fly** currently not supported

```python
fixlen_feature_columns = [SparseFeat(feat, vocabulary_size=1e6, embedding_dim=4, use_hash=True, dtype='string') # since the input is string
                         for feat in sparse_features] + [DenseFeat(feat, 1, ) for feat in dense_features]
```
• generate feature columns

dnn_feature_columns = sparse_feature_columns + dense_feature_columns
linear_feature_columns = sparse_feature_columns + dense_feature_columns
feature_names = get_feature_names(linear_feature_columns + dnn_feature_columns)

Step 4: Generate the training samples and train the model

train, test = train_test_split(data, test_size=0.2)
train_model_input = {name:train[name] for name in feature_names}
test_model_input = {name:test[name] for name in feature_names}

device = 'cpu'
use_cuda = True
if use_cuda and torch.cuda.is_available():
    print('cuda ready...')
    device = 'cuda:0'

model = DeepFM(linear_feature_columns,dnn_feature_columns,task='binary',device=device)
model.compile("adam", "binary_crossentropy",
               metrics=['binary_crossentropy'], )

history = model.fit(train_model_input, train[target].values, batch_size=256, epochs=10,
                        verbose=2, validation_split=0.2)
pred_ans = model.predict(test_model_input, batch_size=256)

You can check the full code here.

2.2 Features

2.2.1 Overview

With the great success of deep learning,DNN-based techniques have been widely used in CTR estimation task.

DNN based CTR estimation models consists of the following 4 modules: Input,Embedding, Low-order&High-order Feature Extractor,Prediction

• Input&Embedding

  The data in CTR estimation task usually includes high sparse,high cardinality categorical features and some dense numerical features.

  Since DNN are good at handling dense numerical features,we usually map the sparse categorical features to dense numerical through embedding technique.

  For numerical features,we usually apply discretization or normalization on them.

• Feature Extractor

  Low-order Extractor learns feature interaction through product between vectors.Factorization-Machine and it's variants are widely used to learn the low-order feature interaction.

  High-order Extractor learns feature combination through complex neural network functions like MLP,Cross Net,etc.
2.2.2 Feature Columns

SparseFeat

SparseFeat is a namedtuple with signature `SparseFeat(name, vocabulary_size, embedding_dim, use_hash, dtype, embedding_name, group_name)`

- name: feature name
- vocabulary_size: number of unique feature values for sparse feature or hashing space when use_hash=True
- embedding_dim: embedding dimension
- use_hash: default False. If True the input will be hashed to space of size vocabulary_size.
- dtype: default float32. dtype of input tensor.
- embedding_name: default None. If None, the embedding_name will be same as name.
- group_name: feature group of this feature.

DenseFeat

DenseFeat is a namedtuple with signature `DenseFeat(name, dimension, dtype)`

- name: feature name
- dimension: dimension of dense feature vector.
- dtype: default float32. dtype of input tensor.

VarLenSparseFeat

VarLenSparseFeat is a namedtuple with signature `VarLenSparseFeat(sparsefeat, maxlen, combiner, length_name)`

- sparsefeat: a instance of SparseFeat
- maxlen: maximum length of this feature for all samples
- combiner: pooling method, can be sum, mean or max
- length_name: feature length name, if None, value 0 in feature is for padding.

2.2.3 Models

CCPM (Convolutional Click Prediction Model)

CCPM can extract local-global key features from an input instance with varied elements, which can be implemented for not only single ad impression but also sequential ad impression.
CCPM Model API


PNN (Product-based Neural Network)

PNN concatenates sparse feature embeddings and the product between embedding vectors as the input of MLP.

PNN Model API

**Wide & Deep**

WDL’s deep part concatenates sparse feature embeddings as the input of MLP, the wide part use handcrafted feature as input. The logits of deep part and wide part are added to get the prediction probability.

**WDL Model API**


**DeepFM**

DeepFM can be seen as an improvement of WDL and FNN. Compared with WDL, DeepFM use FM instead of LR in the wide part and use concatenation of embedding vectors as the input of MLP in the deep part. Compared with FNN, the embedding vector of FM and input to MLP are same. And they do not need a FM pretrained vector to initialize, they are learned end2end.

**DeepFM Model API**
MLR (Mixed Logistic Regression/Piece-wise Linear Model)

MLR can be viewed as a combination of $2m$ LR model, $m$ is the piece(region) number. $m$ LR model learns the weight that the sample belong to each region, another $m$ LR model learns sample’s click probability in the region. Finally, the sample’s CTR is a weighted sum of each region’s click probability. Notice the weight is normalized weight.

MLR Model API
NFM (Neural Factorization Machine)

NFM use a bi-interaction pooling layer to learn feature interaction between embedding vectors and compress the result into a single vector which has the same size as a single embedding vector. And then fed it into a MLP. The output logit of MLP and the output logit of linear part are added to get the prediction probability.

NFM Model API

**AFM (Attentional Factorization Machine)**

AFM is a variant of FM, traditional FM sums the inner product of embedding vector uniformly. AFM can be seen as weighted sum of feature interactions. The weight is learned by a small MLP.

**AFM Model API**

**DCN (Deep & Cross Network)**

DCN use a Cross Net to learn both low and high order feature interaction explicitly, and use a MLP to learn feature interaction implicitly. The output of Cross Net and MLP are concatenated. The concatenated vector are fed into one fully connected layer to get the prediction probability.

**DCN Model API**
DCN-Mix (Improved Deep & Cross Network with mix of experts and matrix kernel)

DCN-Mix uses a matrix kernel instead of vector kernel in CrossNet compared with DCN, and it uses mixture of experts to learn feature interactions.

DCN-Mix Model API


DIN (Deep Interest Network)

DIN introduce a attention method to learn from sequence(multi-valued) feature. Traditional method usually use sum/mean pooling on sequence feature. DIN use a local activation unit to get the activation score between candidate item and history items. User’s interest are represented by weighted sum of user behaviors. user's interest vector and other embedding vectors are concatenated and fed into a MLP to get the prediction.

DIN Model API

DIN example

**DIEN (Deep Interest Evolution Network)**

Deep Interest Evolution Network (DIEN) uses interest extractor layer to capture temporal interests from history behavior sequence. At this layer, an auxiliary loss is proposed to supervise interest extracting at each step. As user interests are diverse, especially in the e-commerce system, interest evolving layer is proposed to capture interest evolving process that is relative to the target item. At interest evolving layer, attention mechanism is embedded into the sequential structure novelly, and the effects of relative interests are strengthened during interest evolution.

**DIEN Model API**

**DIEN example**

**xDeepFM**

xDeepFM use a Compressed Interaction Network (CIN) to learn both low and high order feature interaction explicitly, and use a MLP to learn feature interaction implicitly. In each layer of CIN, first compute outer products between $x^k \times x_0$ to get a tensor $Z_{k+1}$, then use a 1DConv to learn feature maps $H_{k+1}$ on this tensor. Finally, apply sum pooling on all the feature maps $H_k$ to get one vector. The vector is used to compute the logit that CIN contributes.

**xDeepFM Model API**

![Diagram of CIN](image)

(a) Outer products along each dimension for feature interactions. The tensor $Z^{k+1}$ is an intermediate result for further learning.

(b) The $k$-th layer of CIN. It compresses the intermediate tensor $Z^{k+1}$ to $H_{k+1}$, embedding vectors (also known as feature maps).

(c) An overview of the CIN architecture.

**Figure 4:** Components and architecture of the Compressed Interaction Network (CIN).
Figure 5: The architecture of xDeepFM.


**AutoInt (Automatic Feature Interaction)**

AutoInt use a interacting layer to model the interactions between different features. Within each interacting layer, each feature is allowed to interact with all the other features and is able to automatically identify relevant features to form meaningful higher-order features via the multi-head attention mechanism. By stacking multiple interacting layers, AutoInt is able to model different orders of feature interactions.

**AutoInt Model API**
Figure 3: The architecture of interacting layer. Combinatorial features are conditioned on attention weights, i.e., $\alpha_m^{(h)}$. 

DeepCTR-Torch Documentation, Release 0.2.4
Figure 1: Overview of our proposed model AutoInt. The details of embedding layer and interacting layer are illustrated in Figure 2 and Figure 3 respectively.


**ONN (Operation-aware Neural Networks for User Response Prediction)**

ONN models second order feature interactions like like FFM and preserves second-order interaction information as much as possible. Further more, deep neural network is used to learn higher-ordered feature interactions.

**ONN Model API**
FiBiNET (Feature Importance and Bilinear feature Interaction NETwork)

Feature Importance and Bilinear feature Interaction NETwork is proposed to dynamically learn the feature importance and fine-grained feature interactions. On the one hand, the FiBiNET can dynamically learn the importance of features via the Squeeze-Excitation network (SENET) mechanism; on the other hand, it is able to effectively learn the feature interactions via bilinear function.

FiBiNET Model API
FiBiNET


2.2.4 Layers

The models of deepctr are modular, so you can use different modules to build your own models.

You can see layers API in Layers

2.3 Examples

2.3.1 Classification: Criteo

The Criteo Display Ads dataset is for the purpose of predicting ads click-through rate. It has 13 integer features and 26 categorical features where each category has a high cardinality.

In this example, we simply normalize the dense feature between 0 and 1, you can try other transformation technique like log normalization or discretization. Then we use SparseFeat and DenseFeat to generate feature columns for sparse features and dense features.
This example shows how to use DeepFM to solve a simple binary classification task. You can get the demo data criteo_sample.txt and run the following codes.

```python
import pandas as pd
import torch
from sklearn.metrics import log_loss, roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from deepctr_torch.inputs import SparseFeat, DenseFeat, get_feature_names
from deepctr_torch.models import *

if __name__ == '__main__':
    data = pd.read_csv('./criteo_sample.txt')

    sparse_features = ['C' + str(i) for i in range(1, 27)]
    dense_features = ['I' + str(i) for i in range(1, 14)]

    data[sparse_features] = data[sparse_features].fillna('-1', )
data[dense_features] = data[dense_features].fillna(0, )
target = ['label']

    # 1. Label Encoding for sparse features, and do simple Transformation for dense features
    for feat in sparse_features:
        lbe = LabelEncoder()
data[feat] = lbe.fit_transform(data[feat])

    mms = MinMaxScaler(feature_range=(0, 1))
data[dense_features] = mms.fit_transform(data[dense_features])

    # 2. Count #unique features for each sparse field, and record dense feature field name
    fixlen_feature_columns = [SparseFeat(feat, data[feat].nunique()) for feat in sparse_features] + [DenseFeat(feat, 1, ) for feat in dense_features]

dnn_feature_columns = fixlen_feature_columns
linear_feature_columns = fixlen_feature_columns

feature_names = get_feature_names(linear_feature_columns + dnn_feature_columns)

# 3. Generate input data for model
train, test = train_test_split(data, test_size=0.2)

train_model_input = {name: train[name] for name in feature_names}
test_model_input = {name: test[name] for name in feature_names}

# 4. Define Model, train, predict and evaluate
device = 'cpu'
use_cuda = True
if use_cuda and torch.cuda.is_available():
    print('cuda ready...')
device = 'cuda:0'
```

(continues on next page)
model = DeepFM(linear_feature_columns=linear_feature_columns, dnn_feature_columns=dnn_feature_columns, task='binary', l2_reg_embedding=1e-5, device=device)

model.compile("adagrad", "binary_crossentropy", metrics=["binary_crossentropy", "auc"], )
model.fit(train_model_input, train[target].values, batch_size=32, epochs=10, verbose=2, validation_split=0.0)

pred_ans = model.predict(test_model_input, 256)
print(""
print("test LogLoss", round(log_loss(test[target].values, pred_ans), 4))
print("test AUC", round(roc_auc_score(test[target].values, pred_ans), 4))

2.3.2 Regression: Movielens

The MovieLens data has been used for personalized tag recommendation, which contains 668,953 tag applications of users on movies. Here is a small fraction of data include only sparse field.

<table>
<thead>
<tr>
<th>movie_id</th>
<th>user_id</th>
<th>gender</th>
<th>age</th>
<th>occupation</th>
<th>zip</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>254181</td>
<td>2944</td>
<td>M</td>
<td>25</td>
<td>20</td>
<td>20009</td>
<td>4</td>
</tr>
<tr>
<td>481546</td>
<td>2208</td>
<td>M</td>
<td>35</td>
<td>3</td>
<td>94109</td>
<td>3</td>
</tr>
<tr>
<td>166949</td>
<td>3629</td>
<td>M</td>
<td>50</td>
<td>19</td>
<td>59457</td>
<td>5</td>
</tr>
<tr>
<td>536371</td>
<td>569</td>
<td>F</td>
<td>18</td>
<td>20</td>
<td>15701-1348</td>
<td>2</td>
</tr>
<tr>
<td>117094</td>
<td>2763</td>
<td>M</td>
<td>35</td>
<td>7</td>
<td>38024</td>
<td>4</td>
</tr>
</tbody>
</table>

This example shows how to use DeepFM to solve a simple binary regression task. You can get the demo data movie-lens_sample.txt and run the following codes.

```python
import pandas as pd
import torch
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

from deepctr_torch.inputs import SparseFeat, get_feature_names
from deepctr_torch.models import DeepFM

if __name__ == '__main__':
    data = pd.read_csv("./movielens_sample.txt")
    sparse_features = ['movie_id', 'user_id',
                       'gender', 'age', 'occupation', 'zip']
    target = ['rating']

    # 1. Label Encoding for sparse features, and do simple Transformation for dense_features
    for feat in sparse_features:
```
lbe = LabelEncoder()
data[feat] = lbe.fit_transform(data[feat])

# 2. count unique features for each sparse field
fixlen_feature_columns = [SparseFeat(feat, data[feat].nunique())
            for feat in sparse_features]
linear_feature_columns = fixlen_feature_columns
dnn_feature_columns = fixlen_feature_columns
feature_names = get_feature_names(linear_feature_columns + dnn_feature_columns)

# 3. generate input data for model
train, test = train_test_split(data, test_size=0.2)
train_model_input = {name: train[name] for name in feature_names}
test_model_input = {name: test[name] for name in feature_names}

# 4. Define Model, train, predict and evaluate
device = 'cpu'
use_cuda = True
if use_cuda and torch.cuda.is_available():
    print('cuda ready...')
    device = 'cuda:0'
model = DeepFM(linear_feature_columns, dnn_feature_columns, task='regression',
              device=device)
model.compile('adam', 'mse', metrics=['mse'], )
history = model.fit(train_model_input, train[target].values, batch_size=256,
                    epochs=10, verbose=2, validation_split=0.2)
pred_ans = model.predict(test_model_input, batch_size=256)
print("test MSE", round(mean_squared_error(test[target].values, pred_ans), 4))

2.3.3 Multi-value Input: Movielens

The MovieLens data has been used for personalized tag recommendation, which contains 668,953 tag applications of users on movies. Here is a small fraction of data include sparse fields and a multivalent field.

<table>
<thead>
<tr>
<th>movie_id</th>
<th>user_id</th>
<th>gender</th>
<th>age</th>
<th>occupation</th>
<th>zip</th>
<th>genres</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Comedy</td>
<td>Drama</td>
</tr>
<tr>
<td>1</td>
<td>169</td>
<td>123</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>118</td>
<td>Action</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
<td>0</td>
<td>2</td>
<td>13</td>
<td>99</td>
<td>Drama</td>
</tr>
<tr>
<td>3</td>
<td>112</td>
<td>21</td>
<td>1</td>
<td>1</td>
<td>18</td>
<td>55</td>
<td>Action</td>
</tr>
<tr>
<td>4</td>
<td>45</td>
<td>187</td>
<td>1</td>
<td>5</td>
<td>19</td>
<td>41</td>
<td>Comedy</td>
</tr>
</tbody>
</table>

There are 2 additional steps to use DeepCTR with sequence feature input.

1. Generate the padded and encoded sequence feature of sequence input feature (value 0 is for padding).
2. Generate config of sequence feature with VarLenSparseFeat

This example shows how to use DeepFM with sequence(multi-value) feature. You can get the demo data movielens_sample.txt and run the following codes.

```python
import numpy as np
import pandas as pd
import torch
from sklearn.preprocessing import LabelEncoder
from tensorflow.python.keras.preprocessing.sequence import pad_sequences
from deepctr_torch.inputs import SparseFeat, VarLenSparseFeat, get_feature_names
from deepctr_torch.models import DeepFM

def split(x):
    key_ans = x.split('|')
    for key in key_ans:
        if key not in key2index:
            # Notice: input value 0 is a special "padding", so we do not use 0 to encode valid feature for sequence input
            key2index[key] = len(key2index) + 1
    return list(map(lambda x: key2index[x], key_ans))

if __name__ == '__main__':
    data = pd.read_csv('./movielens_sample.txt')
    sparse_features = ['movie_id', 'user_id', 'gender', 'age', 'occupation', 'zip',]
    target = ['rating']
    # 1. Label Encoding for sparse features, and process sequence features
    for feat in sparse_features:
        lbe = LabelEncoder()
        data[feat] = lbe.fit_transform(data[feat])
    # preprocess the sequence feature
    key2index = {}
    genres_list = list(map(split, data['genres'].values))
    genres_length = np.array(list(map(len, genres_list)))
    max_len = max(genres_length)
    # Notice: padding='post'
    genres_list = pad_sequences(genres_list, maxlen=max_len, padding='post',)
    # 2. count #unique features for each sparse field and generate feature config for sequence feature
    fixlen_feature_columns = [SparseFeat(feat, data[feat].nunique(), embedding_dim=4) for feat in sparse_features]
    varlen_feature_columns = [VarLenSparseFeat(SparseFeat('genres', vocabulary_size=len(key2index) + 1, embedding_dim=4), maxlen=max_len, combiner='mean')] # Notice: value 0 is for padding for sequence input feature
    linear_feature_columns = fixlen_feature_columns + varlen_feature_columns
    dnn_feature_columns = fixlen_feature_columns + varlen_feature_columns
```

(continues on next page)
feature_names = get_feature_names(linear_feature_columns + dnn_feature_columns)

# 3. generate input data for model
model_input = {name: data[name] for name in sparse_features}  #
model_input["genres"] = genres_list

# 4. Define Model, compile and train
device = 'cpu'
use_cuda = True
if use_cuda and torch.cuda.is_available():
    print('cuda ready...')
    device = 'cuda:0'

model = DeepFM(linear_feature_columns, dnn_feature_columns, task='regression',
               device=device)

model.compile("adam", "mse", metrics=["mse"],)
history = model.fit(model_input, data[target].values, batch_size=256, epochs=10,
                    verbose=2, validation_split=0.2)

2.4 FAQ

2.4.1 1. Save or load weights/models

To save/load weights:

```python
import torch
model = DeepFM()
torch.save(model.state_dict(), 'DeepFM_weights.h5')
model.load_state_dict(torch.load('DeepFM_weights.h5'))
```

To save/load models:

```python
import torch
model = DeepFM()
torch.save(model, 'DeepFM.h5')
model = torch.load('DeepFM.h5')
```

2.4.2 2. Set learning rate and use earlystopping

Here is an example of how to set learning rate and earlystopping:

```python
from torch.optim import Adagrad
from deepctr_torch.models import DeepFM
from deepctr_torch.callbacks import EarlyStopping, ModelCheckpoint
```
2.4.3 3. How to add a long dense feature vector as a input to the model?

```python
from deepctr_torch.models import DeepFM
from deepctr_torch.inputs import DenseFeat, SparseFeat, get_feature_names
import numpy as np

feature_columns = [SparseFeat('user_id', 120,), SparseFeat('item_id', 60,), DenseFeat('pic_vec', 5)]
fixlen_feature_names = get_feature_names(feature_columns)

user_id = np.array([[1], [0], [1]])
item_id = np.array([[30], [20], [10]])
pic_vec = np.array([[0.1, 0.5, 0.4, 0.3, 0.2], [0.1, 0.5, 0.4, 0.3, 0.2], [0.1, 0.5, 0.4, 0.3, 0.2]])
label = np.array([1, 0, 1])

model_input = {'user_id': user_id, 'item_id': item_id, 'pic_vec': pic_vec}

model = DeepFM(feature_columns, feature_columns)
model.compile('adagrad', 'binary_crossentropy')
model.fit(model_input, label)
```

2.4.4 4. How to run the demo with GPU ?

```python
import torch
device = 'cpu'
use_cuda = True
if use_cuda and torch.cuda.is_available():
    print('cuda ready...')
    device = 'cuda:0'

model = DeepFM(..., device=device)
```

2.5 History

- 12/05/2020 : v0.2.4 released. Improve compatibility & fix issues. Add History callback.(example).
- 10/18/2020 : v0.2.3 released. Add DCN-M & DCN-Mix. Add EarlyStopping and ModelCheckpoint callbacks(example).
2.6 DeepCTR-Torch Models API

2.6.1 deepctr_torch.models.basemodel module

Author: Weichen Shen, wcshen1994@163.com

class deepctr_torch.models.basemodel.BaseModel(linear_feature_columns, 
    dnn_feature_columns, 
    l2_reg_linear=1e-05, 
    l2_reg_embedding=1e-05, 
    init_std=0.0001, 
    seed=1024, 
    task='binary', device='cpu')

    compile(optimizer, loss=None, metrics=None)

    Parameters

    • optimizer – String (name of optimizer) or optimizer instance. See [optimizers](https://pytorch.org/docs/stable/optim.html).

    • loss – String (name of objective function) or objective function. See [losses](https://pytorch.org/docs/stable/nn.functional.html#loss-functions).

    • metrics – List of metrics to be evaluated by the model during training and testing. Typically you will use metrics=['accuracy'].

    evaluate(x, y, batch_size=256)

    Parameters

    • x – Numpy array of test data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs).

    • y – Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs).

    • batch_size – Integer or None. Number of samples per evaluation step. If unspecified, batch_size will default to 256.

    Returns Dict contains metric names and metric values.

    fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, initial_epoch=0, validation_split=0.0, validation_data=None, shuffle=True, callbacks=None)

    Parameters
• `x` – Numpy array of training data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.

• `y` – Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs).

• `batch_size` – Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 256.

• `epochs` – Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. Note that in conjunction with `initial_epoch, epochs` is to be understood as “final epoch”. The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.

• `verbose` – Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.

• `initial_epoch` – Integer. Epoch at which to start training (useful for resuming a previous training run).

• `validation_split` – Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the `x` and `y` data provided, before shuffling.

• `validation_data` – tuple `(x_val, y_val)` or tuple `(x_val, y_val, val_sample_weights)` on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. `validation_data` will override `validation_split`.

• `shuffle` – Boolean. Whether to shuffle the order of the batches at the beginning of each epoch.

• `callbacks` – List of `deepctr_torch.callbacks.Callback` instances. List of callbacks to apply during training and validation (if ). See [callbacks](https://tensorflow.google.cn/api_docs/python/tf/keras/callbacks). Now available: `EarlyStopping`, `ModelCheckpoint`.

Returns A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

```python
predict (x, batch_size=256)
```

Parameters

• `x` – The input data, as a Numpy array (or list of Numpy arrays if the model has multiple inputs).

• `batch_size` – Integer. If unspecified, it will default to 256.

Returns Numpy array(s) of predictions.

### 2.6.2 `deepctr_torch.models.ccpm` module

**Author:** Zeng Kai, kk163mail@126.com

class deepctr_torch.models.ccpm.CCPM(linear_feature_columns, dnn_feature_columns, conv_kernel_width=(6, 5), conv_filters=(4, 4), dnn_hidden_units=(256, ), l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_dnn=0, dnn_dropout=0, init_std=0.0001, seed=1024, task='binary', device='cpu', dnn_use_bn=False, dnn_activation='relu')

Instantiates the Convolutional Click Prediction Model architecture.

Parameters

- **linear_feature_columns** – An iterable containing all the features used by linear part of the model.
- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **conv_kernel_width** – list, list of positive integer or empty list, the width of filter in each conv layer.
- **conv_filters** – list, list of positive integer or empty list, the number of filters in each conv layer.
- **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of DNN.
- **l2_reg_linear** – float. L2 regularizer strength applied to linear part
- **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector
- **l2_reg_dnn** – float. L2 regularizer strength applied to DNN
- **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.
- **init_std** – float, to use as the initialize std of embedding vector
- **seed** – integer, to use as random seed.
- **task** – str, "binary" for binary logloss or "regression" for regression loss
- **device** – str, "cpu" or "cuda:0"

Returns

A PyTorch model instance.

forward(X)

Defines the computation performed at every call. Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.6.3 deepctr_torch.models.pnn module

Author: Weichen Shen, wcshen1994@163.com

class deepctr_torch.models.pnn.PNN(dnn_feature_columns,  
dnn_hidden_units=(128,  
128), l2_reg_embedding=1e-05,  l2_reg_dnn=0,  
init_std=0.0001,  seed=1024,  dnn_dropout=0,  
dnn_activation='relu',  use_inner=True,  
use_outter=False,  kernel_type='mat',  task='binary',  
device='cpu')

Instantiates the Product-based Neural Network architecture.

Parameters

- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of deep net
- **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector
- **l2_reg_dnn** – float. L2 regularizer strength applied to DNN
- **init_std** – float, to use as the initialize std of embedding vector
- **seed** – integer, to use as random seed.
- **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.
- **dnn_activation** – Activation function to use in DNN
- **use_inner** – bool, whether use inner-product or not.
- **use_outter** – bool, whether use outter-product or not.
- **kernel_type** – str, kernel_type used in outter-product, can be 'mat', 'vec' or 'num'
- **task** – str, "binary" for binary logloss or "regression" for regression loss
- **device** – str, "cpu" or "cuda:0"

Returns

A PyTorch model instance.

forward(X)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.6.4 deepctr_torch.models.wdl module

Author: Weichen Shen, wcshen1994@163.com

class deepctr_torch.models.wdl.WDL(linear_feature_columns, dnn_feature_columns, dnn_hidden_units=(256, 128), l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_dnn=0, init_std=0.0001, seed=1024, dnn_dropout=0, dnn_activation='relu', dnn_use_bn=False, task='binary', device='cpu')

Instantiates the Wide&Deep Learning architecture.

Parameters

- **linear_feature_columns** – An iterable containing all the features used by linear part of the model.
- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **dnn_hidden_units** – list,list of positive integer or empty list, the layer number and units in each layer of DNN
- **l2_reg_linear** – float. L2 regularizer strength applied to wide part
- **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector
- **l2_reg_dnn** – float. L2 regularizer strength applied to DNN
- **init_std** – float, to use as the initialize std of embedding vector
- **seed** – integer, to use as random seed.
- **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.
- **dnn_activation** – Activation function to use in DNN
- **task** – str, "binary" for binary logloss or "regression" for regression loss
- **device** – str, "cpu" or "cuda:0"

Returns

A PyTorch model instance.

forward(X)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.6.5 deepctr_torch.models.deepfm module

Author: Weichen Shen, wcshen1994@163.com

class deepctr_torch.models.deepfm.DeepFM(linear_feature_columns, dnn_feature_columns, use_fm=True, dnn_hidden_units=(256, 128), l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_dnn=0, init_std=0.0001, seed=1024, dnn_dropout=0, dnn_activation='relu', dnn_use_bn=False, task='binary', device='cpu')

Instantiates the DeepFM Network architecture.

Parameters

- **linear_feature_columns** – An iterable containing all the features used by linear part of the model.
- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **use_fm** – bool, use FM part or not
- **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of DNN
- **l2_reg_linear** – float, L2 regularizer strength applied to linear part
- **l2_reg_embedding** – float, L2 regularizer strength applied to embedding vector
- **l2_reg_dnn** – float, L2 regularizer strength applied to DNN
- **init_std** – float, to use as the initialize std of embedding vector
- **seed** – integer, to use as random seed.
- **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.
- **dnn_activation** – Activation function to use in DNN
- **dnn_use_bn** – bool, Whether use BatchNormalization before activation or not in DNN
- **task** – str, "binary" for binary logloss or "regression" for regression loss
- **device** – str, "cpu" or "cuda:0"

Returns

A PyTorch model instance.

forward(X)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.6.6 deepctr_torch.models.mlr module

Author: Wutong Zhang Weichen Shen, wcshen1994@163.com

class deepctr_torch.models.mlr.MLR(region_feature_columns, base_feature_columns=None, bias_feature_columns=None, region_num=4, l2_reg_linear=1e-05, init_std=0.0001, seed=1024, task='binary', device='cpu')

Instantiates the Mixed Logistic Regression/Piece-wise Linear Model.

Parameters

- **region_feature_columns** – An iterable containing all the features used by region part of the model.
- **base_feature_columns** – An iterable containing all the features used by base part of the model.
- **region_num** – integer > 1, indicate the piece number
- **l2_reg_linear** – float. L2 regularizer strength applied to weight
- **init_std** – float, to use as the initialize std of embedding vector
- **seed** – integer, to use as random seed.
- **task** – str, "binary" for binary logloss or "regression" for regression loss
- **bias_feature_columns** – An iterable containing all the features used by bias part of the model.
- **device** – str, "cpu" or "cuda:0"

Returns

A PyTorch model instance.

forward(X)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

### 2.6.7 deepctr_torch.models.nfm module

**Author:** Weichen Shen, wcshen1994@163.com


class deepctr_torch.models.nfm.NFM(linear_feature_columns, dnn_feature_columns, dnn_hidden_units=(128, 128), l2_reg_embedding=1e-05, l2_reg_linear=1e-05, l2_reg_dnn=0, init_std=0.0001, seed=1024, bi_dropout=0, dnn_dropout=0, dnn_activation='relu', task='binary', device='cpu')

Instantiates the NFM Network architecture.

Parameters

- **linear_feature_columns** – An iterable containing all the features used by linear part of the model.
• **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.

• **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of deep net

• **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector

• **l2_reg_linear** – float. L2 regularizer strength applied to linear part.

• **l2_reg_dnn** – float. L2 regularizer strength applied to DNN

• **init_std** – float, to use as the initialize std of embedding vector

• **seed** – integer, to use as random seed.

• **biout_dropout** – When not **None**, the probability we will drop out the output of BiInteractionPooling Layer.

• **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.

• **dnn_activation** – Activation function to use in deep net

• **task** – str, "binary" for binary logloss or "regression" for regression loss

• **device** – str, "cpu" or "cuda:0"

**Returns** A PyTorch model instance.

**forward** (*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the **Module** instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

### 2.6.8 deepctr_torch.models.afm module

**Author:** Weichen Shen, wcshen1994@163.com


**class** deepctr_torch.models.afm.AFM(*linear_feature_columns, dnn_feature_columns, use_attention=True, attention_factor=8, l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_att=1e-05, afm_dropout=0, init_std=0.0001, seed=1024, task='binary', device='cpu')

Instantiates the Attentional Factorization Machine architecture.

**Parameters**

• **linear_feature_columns** – An iterable containing all the features used by linear part of the model.

• **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.

• **use_attention** – bool, whether use attention or not, if set to **False**, it is the same as standard Factorization Machine
• **attention_factor** – positive integer, units in attention net
• **l2_reg_linear** – float, L2 regularizer strength applied to linear part
• **l2_reg_embedding** – float, L2 regularizer strength applied to embedding vector
• **l2_reg_att** – float, L2 regularizer strength applied to attention net
• **afm_dropout** – float in [0, 1), Fraction of the attention net output units to dropout.
• **init_std** – float, to use as the initialize std of embedding vector
• **seed** – integer, to use as random seed.
• **task** – str, "binary" for binary logloss or "regression" for regression loss
• **device** – str, "cpu" or "cuda:0"

**Returns** A PyTorch model instance.

**forward** *(X)*

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

### 2.6.9 deepctr_torch.models.dcn module

**Author:** chen_kkkk, bgasdo36977@gmail.com
zanshuxun, zanshuxun@aliyun.com

**Reference:**


**class** *deepctr_torch.models.dcn.DCN*(linear_feature_columns, dnn_feature_columns, cross_num=2, cross_parameterization='vector',

- dnn_hidden_units=(128, 128), l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_cross=1e-05, l2_reg_dnn=0, init_std=0.0001, seed=1024,
- dnn_dropout=0, dnn_activation='relu', dnn_use_bn=False, task='binary', device='cpu')

Instantiates the Deep&Cross Network architecture. Including DCN-V (parameterization='vector') and DCN-M (parameterization='matrix').

**Parameters**

• **linear_feature_columns** – An iterable containing all the features used by linear part of the model.

• **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.

• **cross_num** – positive integer, cross layer number
• **cross_parameterization** – str, "vector" or "matrix", how to parameterize the cross network.

• **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of DNN

• **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector

• **l2_reg_cross** – float. L2 regularizer strength applied to cross net

• **l2_reg_dnn** – float. L2 regularizer strength applied to DNN

• **init_std** – float, to use as the initialize std of embedding vector

• **seed** – integer, to use as random seed.

• **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.

• **dnn_use_bn** – bool. Whether use BatchNormalization before activation or not DNN

• **dnn_activation** – Activation function to use in DNN

• **task** – str, "binary" for binary logloss or "regression" for regression loss

• **device** – str, "cpu" or "cuda:0"

**Returns**  A PyTorch model instance.

**forward**(X)

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

### 2.6.10 deepctr_torch.models.dcnmix module

**Author:** chen_kkkk, bgasdo36977@gmail.com

zanshuxun, zanshuxun@aliyun.com

**Reference:**


**class** deepctr_torch.models.dcnmix.DCNMix(linear_feature_columns, dnn_feature_columns, cross_num=2, dnn_hidden_units=(128, 128), l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_cross=1e-05, l2_reg_dnn=0, init_std=0.0001, seed=1024, dnn_dropout=0, low_rank=32, num_experts=4, dnn_activation='relu', dnn_use_bn=False, task='binary', device='cpu')

Instantiates the DCN-Mix model.

**Parameters**
• `linear_feature_columns` – An iterable containing all the features used by linear part of the model.

• `dnn_feature_columns` – An iterable containing all the features used by deep part of the model.

• `cross_num` – positive integer, cross layer number

• `dnn_hidden_units` – list, list of positive integer or empty list, the layer number and units in each layer of DNN

• `l2_reg_embedding` – float. L2 regularizer strength applied to embedding vector

• `l2_reg_cross` – float. L2 regularizer strength applied to cross net

• `l2_reg_dnn` – float. L2 regularizer strength applied to DNN

• `init_std` – float, to use as the initialize std of embedding vector

• `seed` – integer, to use as random seed.

• `dnn_dropout` – float in [0,1), the probability we will drop out a given DNN coordinate.

• `dnn_use_bn` – bool. Whether use BatchNormalization before activation or not DNN

• `dnn_activation` – Activation function to use in DNN

• `low_rank` – Positive integer, dimensionality of low-rank sapce.

• `num_experts` – Positive integer, number of experts.

• `task` – str, "binary" for binary logloss or "regression" for regression loss

• `device` – str, "cpu" or "cuda:0"

Returns A PyTorch model instance.

```python
forward(X)
```
Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

### 2.6.11 deepctr_torch.models.din module

**Author:** Yuef Zhang


```python
class deepctr_torch.models.din.DIN(dnn_feature_columns, history_feature_list, dnn_use_bn=False, dnn_hidden_units=(256, 128), dnn_activation='relu', att_hidden_size=(64, 16), att_activation='Dice', att_weight_normalization=False, l2_reg_dnn=0.0, l2_reg_embedding=1e-06, l2_reg_cross=0.0, l2_reg_embedding=0.0, init_std=0.0001, seed=1024, task='binary', device='cpu')
```

Instantiates the Deep Interest Network architecture.
Parameters

- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **history_feature_list** – list, to indicate sequence sparse field
- **dnn_use_bn** – bool. Whether use BatchNormalization before activation or not in deep net
- **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of deep net
- **dnn_activation** – Activation function to use in deep net
- **att_hidden_size** – list, list of positive integer, the layer number and units in each layer of attention net
- **att_activation** – Activation function to use in attention net
- **att_weight_normalization** – bool. Whether normalize the attention score of local activation unit.
- **l2_reg_dnn** – float. L2 regularizer strength applied to DNN
- **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector
- **dnn_dropout** – float in [0, 1), the probability we will drop out a given DNN coordinate.
- **init_std** – float, to use as the initialize std of embedding vector
- **seed** – integer, to use as random seed.
- **task** – str, "binary" for binary logloss or "regression" for regression loss

Returns

A PyTorch model instance.

```python
forward(X)
```

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

### 2.6.12 `deepctr_torch.models.dien` module

**Author:** Ze Wang, wangze0801@126.com


```python
class deepctr_torch.models.dien.DIEN(dnn_feature_columns, history_feature_list, gru_type='GRU', use_negsampling=False, alpha=1.0, use_bn=False, dnn_hidden_units=(256, 128), dnn_activation='relu', att_hidden_units=(64, 16), att_activation='relu', att_weight_normalization=True, l2_reg_dnn=0, l2_reg_embedding=1e-06, dnn_dropout=0, init_std=0.0001, seed=1024, task='binary', device='cpu')
```
Instantiates the Deep Interest Evolution Network architecture.

**Parameters**

- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **history_feature_list** – list, to indicate sequence sparse field
- **gru_type** – str, can be GRU AIGRU AUGRU AGRU
- **use_negsampling** – bool, whether or not use negative sampling
- **alpha** – float, weight of auxiliary loss
- **use_bn** – bool. Whether use BatchNormalization before activation or not in deep net
- **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of DNN
- **dnn_activation** – Activation function to use in DNN
- **att_hidden_units** – list, list of positive integer, the layer number and units in each layer of attention net
- **att_activation** – Activation function to use in attention net
- **att_weight_normalization** – bool. Whether normalize the attention score of local activation unit.
- **l2_reg_dnn** – float. L2 regularizer strength applied to DNN
- **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector
- **dnn_dropout** – float in [0, 1), the probability we will drop out a given DNN coordinate.
- **init_std** – float, to use as the initialize std of embedding vector
- **seed** – integer, to use as random seed.
- **task** – str, “binary” for binary logloss or “regression” for regression loss
- **device** – str, “cpu” or “cuda:0”

**Returns** A PyTorch model instance.

**forward**(*X*)

 Defines the computation performed at every call.
 Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

```python
class deepctr_torch.models.dien.InterestEvolving(input_size, gru_type='GRU', use_neg=False, init_std=0.001,
att_hidden_size=(64, 16), att_activation='sigmoid',
att_weight_normalization=False)

forward(query, keys, keys_length, mask=None)
query: 2D tensor, [B, H] keys: (masked_interests), 3D tensor, [b, T, H] keys_length: 1D tensor, [B]
outputs: 2D tensor, [B, H]
```
```python
class deepctr_torch.models.dien.InterestExtractor(input_size, use_neg=False, init_std=0.001, device='cpu')

forward(keys, keys_length, neg_keys=None)
keys: 3D tensor, [B, T, H] keys_length: 1D tensor, [B] neg_keys: 3D tensor, [B, T, H]
masked_interests: 2D tensor, [b, H] aux_loss: [1]
```

### 2.6.13 deepctr_torch.models.xdeepfm module

**Author:** Wutong Zhang


```python
class deepctr_torch.models.xdeepfm.xDeepFM(linear_feature_columns, dnn_feature_columns, dnn_hidden_units=(256, 256), cin_layer_size=(256, 128), cin_split_half=True, cin_activation='relu', l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_dnn=0, l2_reg_cin=0, init_std=0.0001, seed=1024, dnn_dropout=0, dnn_activation='relu', dnn_use_bn=False, task='binary', device='cpu')
```

Instantiates the xDeepFM architecture.

**Parameters**

- **linear_feature_columns** – An iterable containing all the features used by linear part of the model.
- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **dnn_hidden_units** – list/list of positive integer or empty list, the layer number and units in each layer of deep net
- **cin_layer_size** – list/list of positive integer or empty list, the feature maps in each hidden layer of Compressed Interaction Network
- **cin_split_half** – bool.if set to True, half of the feature maps in each hidden will connect to output unit
- **cin_activation** – activation function used on feature maps
- **l2_reg_linear** – float. L2 regularizer strength applied to linear part
- **l2_reg_embedding** – L2 regularizer strength applied to embedding vector
- **l2_reg_dnn** – L2 regularizer strength applied to deep net
- **l2_reg_cin** – L2 regularizer strength applied to CIN.
- **init_std** – float,to use as the initialize std of embedding vector
- **seed** – integer ,to use as random seed.
- **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.
- **dnn_activation** – Activation function to use in DNN
- **dnn_use_bn** – bool. Whether use BatchNormalization before activation or not in DNN

2.6. DeepCTR-Torch Models API
• **task** – str, "binary" for binary logloss or "regression" for regression loss

• **device** – str, "cpu" or "cuda:0"

**Returns**  A PyTorch model instance.

**forward**(X)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

### 2.6.14 deepctr_torch.models.autoint module

**Author:** Weichen Shen, wcshen1994@163.com


**class** deepctr_torch.models.autoint.AutoInt(linear_feature_columns, dnn_feature_columns, att_layer_num=3, att_embedding_size=8, att_head_num=2, att_res=True, dnn_hidden_units=(256, 128), dnn_activation='relu', l2_reg_dnn=0, l2_reg_embedding=1e-05, dnn_use_bn=False, dnn_dropout=0, init_std=0.0001, seed=1024, task='binary', device='cpu')

Instantiates the AutoInt Network architecture.

**Parameters**

• **linear_feature_columns** – An iterable containing all the features used by linear part of the model.

• **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.

• **att_layer_num** – int. The InteractingLayer number to be used.

• **att_embedding_size** – int. The embedding size in multi-head self-attention network.

• **att_head_num** – int. The head number in multi-head self-attention network.

• **att_res** – bool. Whether or not use standard residual connections before output.

• **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of DNN

• **dnn_activation** – Activation function to use in DNN

• **l2_reg_dnn** – float. L2 regularizer strength applied to DNN

• **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector

• **dnn_use_bn** – bool. Whether use BatchNormalization before activation or not in DNN

• **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.
• **init_std** – float, to use as the initialize std of embedding vector
• **seed** – integer, to use as random seed.
• **task** – str, "binary" for binary logloss or "regression" for regression loss
• **device** – str, "cpu" or "cuda:0"

**Returns** A PyTorch model instance.

```python
forward(X)
```
Defines the computation performed at every call.
Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

### 2.6.15 deepctr_torch.models.onn module

**Author:** Junyi Huo


```python
class deepctr_torch.models.onn.ONN(linear_feature_columns, dnn_feature_columns, 
dnn_hidden_units=(128, 128), l2_reg_embedding=1e-05,  
l2_reg_linear=1e-05, l2_reg_dnn=0, dnn_dropout=0,  
init_std=0.0001, seed=1024, dnn_use_bn=False,  
dnn_activation='relu', task='binary', device='cpu')
```

Instantiates the Operation-aware Neural Networks architecture.

**Parameters**

• **linear_feature_columns** – An iterable containing all the features used by linear part of the model.
• **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
• **dnn_hidden_units** – list, list of positive integer or empty list, the layer number and units in each layer of deep net
• **l2_reg_embedding** – float, L2 regularizer strength applied to embedding vector
• **l2_reg_linear** – float, L2 regularizer strength applied to linear part.
• **l2_reg_dnn** – float, L2 regularizer strength applied to DNN
• **init_std** – float, to use as the initialize std of embedding vector
• **seed** – integer, to use as random seed.
• **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.
• **use_bn** – bool, whether use bn after ffm out or not
• **reduce_sum** – bool, whether apply reduce_sum on cross vector
• **task** – str, "binary" for binary logloss or "regression" for regression loss
• **device** – str, "cpu" or "cuda:0"
Returns A PyTorch model instance.

forward \( (X) \)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.6.16 deepctr_torch.models.fibinet module

Author: Wutong Zhang


class deepctr_torch.models.fibinet.FiBiNET(linear_feature_columns, dnn_feature_columns, bilinear_type='interaction', reduction_ratio=3, dnn_hidden_units=(128, 128), l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_dnn=0, init_std=0.0001, seed=1024, dnn_dropout=0, dnn_activation='relu', task='binary', device='cpu')

Instantiates the Feature Importance and Bilinear feature Interaction NETwork architecture.

Parameters

- **linear_feature_columns** – An iterable containing all the features used by linear part of the model.
- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **bilinear_type** – str,bilinear function type used in Bilinear Interaction Layer,can be 'all', 'each' or 'interaction'
- **reduction_ratio** – integer in \([1, \infty)\), reduction ratio used in SENET Layer
- **dnn_hidden_units** – list,list of positive integer or empty list, the layer number and units in each layer of DNN
- **l2_reg_linear** – float. L2 regularizer strength applied to wide part
- **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector
- **l2_reg_dnn** – float. L2 regularizer strength applied to DNN
- **init_std** – float,to use as the initialize std of embedding vector
- **seed** – integer ,to use as random seed.
- **dnn_dropout** – float in \([0, 1)\), the probability we will drop out a given DNN coordinate.
- **dnn_activation** – Activation function to use in DNN
- **task** – str, "binary" for binary logloss or "regression" for regression loss
• **device** – str. "cpu" or "cuda:0"

**Returns** A PyTorch model instance.

**forward**(X)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

### 2.7 DeepCTR-Torch Layers API

#### 2.7.1 deepctr_torch.layers.core module

**class** deepctr_torch.layers.core.Conv2dSame(\texttt{in\_channels}, \texttt{out\_channels}, \texttt{kernel\_size}, \texttt{stride}=1, \texttt{padding}=0, \texttt{dilation}=1, \texttt{groups}=1, \texttt{bias}=True)

Tensorflow like ‘SAME’ convolution wrapper for 2D convolutions

**forward**(x)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**class** deepctr_torch.layers.core.DNN(\texttt{inputs\_dim}, \texttt{hidden\_units}, \texttt{activation}='relu', \texttt{l2\_reg}=0, \texttt{dropout\_rate}=0, \texttt{use\_bn}=False, \texttt{init\_std}=0.0001, \texttt{dice\_dim}=3, \texttt{seed}=1024, \texttt{device}='cpu')

The Multi Layer Percetron

**Input shape**

- nD tensor with shape: (\texttt{batch\_size}, ..., \texttt{input\_dim}). The most common situation would be a 2D input with shape (\texttt{batch\_size}, \texttt{input\_dim}).

**Output shape**

- nD tensor with shape: (\texttt{batch\_size}, ..., \texttt{hidden\_size[-1]}). For instance, for a 2D input with shape (\texttt{batch\_size}, \texttt{input\_dim}), the output would have shape (\texttt{batch\_size}, \texttt{hidden\_size[-1]}).

**Arguments**

- \texttt{inputs\_dim}: input feature dimension.
- \texttt{hidden\_units}: list of positive integer, the layer number and units in each layer.
- \texttt{activation}: Activation function to use.
- \texttt{l2\_reg}: float between 0 and 1. L2 regularizer strength applied to the kernel weights matrix.
• **`dropout_rate`**: float in [0,1). Fraction of the units to dropout.
• **`use_bn`**: bool. Whether use BatchNormalization before activation or not.
• **`seed`**: A Python integer to use as random seed.

**forward** *(inputs)*

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note**: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

```python
class deepctr_torch.layers.core.LocalActivationUnit(hidden_units=(64, 32), embedding_dim=4, activation='sigmoid', dropout_rate=0, dice_dim=3, l2_reg=0, use_bn=False)
```

The **LocalActivationUnit** used in DIN with which the representation of user interests varies adaptively given different candidate items.

**Input shape**

- A list of two 3D tensor with shape: 
  
  (batch_size, 1, embedding_size) and 
  
  (batch_size, T, embedding_size)

**Output shape**

- 3D tensor with shape: 
  
  (batch_size, T, 1).

**Arguments**

- **`hidden_units`**: list of positive integer, the attention net layer number and units in each layer.
- **`activation`**: Activation function to use in attention net.
- **`l2_reg`**: float between 0 and 1. L2 regularizer strength applied to the kernel weights matrix of attention net.
- **`dropout_rate`**: float in [0,1). Fraction of the units to dropout in attention net.
- **`use_bn`**: bool. Whether use BatchNormalization before activation or not in attention net.
- **`seed`**: A Python integer to use as random seed.

**References**


**forward** *(query, user_behavior)*

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note**: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.
class deepctr_torch.layers.core.PredictionLayer(task='binary', use_bias=True, **kwargs)

Arguments

- **task**: str, "binary" for binary logloss or "regression" for regression loss
- **use_bias**: bool. Whether add bias term or not.

forward(X)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.7.2 deepctr_torch.layers.interaction module

class deepctr_torch.layers.interaction.AFMLayer(in_features, attention_factor=4, l2_reg_w=0, dropout_rate=0, seed=1024, device='cpu')

Attentional Factorization Machine models pairwise (order-2) feature interactions without linear term and bias.

Input shape
- A list of 3D tensor with shape: (batch_size, 1, embedding_size).

Output shape
- 2D tensor with shape: (batch_size, 1).

Arguments

- **in_features**: Positive integer, dimensionality of input features.
- **attention_factor**: Positive integer, dimensionality of the attention network output space.
- **l2_reg_w**: float between 0 and 1. L2 regularizer strength applied to attention network.
- **dropout_rate**: float between in [0,1). Fraction of the attention net output units to dropout.
- **seed**: A Python integer to use as random seed.

References

- [Attentional Factorization Machines : Learning the Weight of Feature Interactions via Attention Networks](https://arxiv.org/pdf/1708.04617.pdf)

forward(inputs)
Defines the computation performed at every call.
Should be overridden by all subclasses.
Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

```python
class deepctr_torch.layers.interaction.BiInteractionPooling
Bi-Interaction Layer used in Neural FM, compress the pairwise element-wise product of features into one single vector.

Input shape
• A 3D tensor with shape: (batch_size, field_size, embedding_size).

Output shape
• 3D tensor with shape: (batch_size, 1, embedding_size).

References
```

```python
def forward(inputs)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.
```

```python
class deepctr_torch.layers.interaction.BilinearInteraction (field_size, embedding_size, bilinear_type='interaction', seed=1024, device='cpu')
BilinearInteraction Layer used in FiBiNET.

Input shape
• A list of 3D tensor with shape: (batch_size, field_size, embedding_size).

Output shape
• 3D tensor with shape: (batch_size, field_size, embedding_size).

Arguments
• field_size : Positive integer, number of feature groups.
• str : String, types of bilinear functions used in this layer.
• seed : A Python integer to use as random seed.

References
• [FiBiNET: Combining Feature Importance and Bilinear feature Interaction for Click-Through Rate Prediction Tongwen](https://arxiv.org/pdf/1905.09433.pdf)
```
forward(inputs)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.interaction.CIN(field_size, layer_size=(128, 128), activation='relu', split_half=True, l2_reg=1e-05, seed=1024, device='cpu')

Compressed Interaction Network used in xDeepFM. Input shape
• 3D tensor with shape: (batch_size, field_size, embedding_size).

Output shape
• 2D tensor with shape: (batch_size, featuremap_num) featuremap_num = sum(self.layer_size[:-1]) // 2 + self.layer_size[-1] if split_half=True else sum(layer_size).

Arguments
• filed_size: Positive integer, number of feature groups.
• layer_size: list of int. Feature maps in each layer.
• activation: activation function name used on feature maps.
• split_half: bool. If set to False, half of the feature maps in each hidden will connect to output unit.
• seed: A Python integer to use as random seed.

References

forward(inputs)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.interaction.ConvLayer(field_size, conv_kernel_width, conv_filters, device='cpu')

Conv Layer used in CCPM.

Input shape
• A list of N 3D tensor with shape: (batch_size, 1, filed_size, embedding_size).

Output shape
• A list of N 3D tensor with shape: (batch_size, last_filters, pooling_size, embedding_size).

Arguments

• field_size : Positive integer, number of feature groups.
• conv_kernel_width: list. list of positive integer or empty list, the width of filter in each conv layer.
• conv_filters: list. list of positive integer or empty list, the number of filters in each conv layer.

Reference:


forward(inputs)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.interaction.CrossNet(in_features, layer_num=2, parameterization='vector', seed=1024, device='cpu')

The Cross Network part of Deep & Cross Network model, which leans both low and high degree cross feature.

Input shape

• 2D tensor with shape: (batch_size, units).

Output shape

• 2D tensor with shape: (batch_size, units).

Arguments

• in_features: Positive integer, dimensionality of input features.
• input_feature_num: Positive integer, shape(Input tensor)[-1]
• layer_num: Positive integer, the cross layer number
• parameterization: string, "vector" or "matrix", way to parameterize the cross network.
• l2_reg: float between 0 and 1. L2 regularizer strength applied to the kernel weights matrix
• seed: A Python integer to use as random seed.

References

forward(inputs)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.interaction.CrossNetMix(in_features, low_rank=32, num_experts=4, layer_num=2, device='cpu')

The Cross Network part of DCN-Mix model, which improves DCN-M by: 1 add MOE to learn feature interactions in different subspaces 2 add nonlinear transformations in low-dimensional space

Input shape
• 2D tensor with shape: (batch_size, units).

Output shape
• 2D tensor with shape: (batch_size, units).

Arguments
• in_features : Positive integer, dimensionality of input features.
• low_rank : Positive integer, dimensionality of low-rank space.
• num_experts : Positive integer, number of experts.
• layer_num : Positive integer, the cross layer number
• device: str, e.g. "cpu" or "cuda:0"

References

forward(inputs)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.interaction.FM
Factorization Machine models pairwise (order-2) feature interactions without linear term and bias.

Input shape
• 3D tensor with shape: (batch_size, field_size, embedding_size).

Output shape
• 2D tensor with shape: (batch_size, 1).

References
• [Factorization Machines](https://www.csie.ntu.edu.tw/~b97053/paper/Rendle2010FM.pdf)
forward(inputs)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.interaction.InnerProductLayer(reduce_sum=True, device='cpu')
InnerProduct Layer used in PNN that compute the element-wise product or inner product between feature vectors.

Input shape
• a list of 3D tensor with shape: (batch_size,1,embedding_size).

Output shape
• 3D tensor with shape: (batch_size, N*(N-1)/2 ,1) if use reduce_sum. or 3D tensor with shape: (batch_size, N*(N-1)/2, embedding_size) if not use reduce_sum.

Arguments
• reduce_sum: bool. Whether return inner product or element-wise product

References

forward(inputs)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.interaction.InteractingLayer(in_features, att_embedding_size=8, head_num=2, use_res=True, seed=1024, device='cpu')
A Layer used in AutoInt that model the correlations between different feature fields by multi-head self-attention mechanism. Input shape
• A 3D tensor with shape: (batch_size,field_size,embedding_size).

Output shape
• 3D tensor with shape: (batch_size,field_size,att_embedding_size * head_num).
Arguments

- **in_features**: Positive integer, dimensionality of input features.
- **att_embedding_size**: int. The embedding size in multi-head self-attention network.
- **head_num**: int. The head number in multi-head self-attention network.
- **use_res**: bool. Whether or not use standard residual connections before output.
- **seed**: A Python integer to use as random seed.

References


```python
class deepctr_torch.layers.interaction.OutterProductLayer(field_size, embedding_size, kernel_type='mat', seed=1024, device='cpu')
```

OutterProduct Layer used in PNN. This implemenation is adapted from code that the author of the paper published on [https://github.com/Atomu2014/product-nets](https://github.com/Atomu2014/product-nets).

Input shape

- A list of N 3D tensor with shape: (batch_size, 1, embedding_size).

Output shape

- 2D tensor with shape: (batch_size, \(N \times (N-1)/2\)).

Arguments

- **field_size**: Positive integer, number of feature groups.
- **kernel_type**: str. The kernel weight matrix type to use, can be mat, vec or num
- **seed**: A Python integer to use as random seed.

References


```python
forward(inputs)
```

Defines the computation performed at every call.

Should be overridden by all subclasses.
class deepctr_torch.layers.interaction.SENETLayer(
    filed_size,    reduction_ratio=3,
    seed=1024, device='cpu')

SENETLayer used in FiBiNET.

Input shape

• A list of 3D tensor with shape: (batch_size, filed_size, embedding_size).

Output shape

• A list of 3D tensor with shape: (batch_size, filed_size, embedding_size).

Arguments

• filed_size : Positive integer, number of feature groups.
• reduction_ratio : Positive integer, dimensionality of the 
  attention network output space.
• seed : A Python integer to use as random seed.

References

• [FiBiNET: Combining Feature Importance and Bilinear feature Interaction for Click-Through Rate Prediction]
  Tongwen](https://arxiv.org/pdf/1905.09433.pdf)

forward(inputs)

Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the
Module instance afterwards instead of this since the former takes care of running the registered hooks
while the latter silently ignores them.

2.7.3 deepctr_torch.layers.sequence module

class deepctr_torch.layers.sequence.AGRUCell(
    input_size, hidden_size, bias=True)

Attention based GRU (AGRU)

Reference: - Deep Interest Evolution Network for Click-Through Rate Prediction[J]. arXiv preprint

forward(input, hx, att_score)

Defines the computation performed at every call.
Should be overridden by all subclasses.
class deepctr_torch.layers.sequence.AUGRUCell(input_size, hidden_size, bias=True)
Effect of GRU with attentional update gate (AUGRU)

Reference: - Deep Interest Evolution Network for Click-Through Rate Prediction[J]. arXiv preprint

forward(input, hx, att_score)
Defines the computation performed at every call.
Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the
Module instance afterwards instead of this since the former takes care of running the registered hooks
while the latter silently ignores them.

class deepctr_torch.layers.sequence.AttentionSequencePoolingLayer(att_hidden_units=(80, 40),
att_activation='sigmoid', weight_normalization=False, return_score=False, supports_masking=False, embedding_dim=4, **kwargs)
The Attentional sequence pooling operation used in DIN & DIEN.

Arguments
- att_hidden_units: list of positive integer, the attention net layer number and units in each layer.
- att_activation: Activation function to use in attention net.
- weight_normalization: bool. Whether normalize the attention score of local activation unit.
- supports_masking: If True, the input need to support masking.

References

forward(query, keys, keys_length, mask=None)

Input shape
- A list of three tensor: [query, keys, keys_length]
- query is a 3D tensor with shape: (batch_size, 1, embedding_size)
- keys is a 3D tensor with shape: (batch_size, T, embedding_size)
- keys_length is a 2D tensor with shape: (batch_size, 1)
Output shape

• 3D tensor with shape: (batch_size, 1, embedding_size).

class deepctr_torch.layers.sequence.DynamicGRU(input_size, hidden_size, bias=True, gru_type='AGRU')

forward(input, att_scores=None, hx=None)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.sequence.KMaxPooling(k, axis, device='cpu')

K Max pooling that selects the k biggest value along the specific axis.

Input shape

• nD tensor with shape: (batch_size, ..., input_dim).

Output shape

• nD tensor with shape: (batch_size, ..., output_dim).

Arguments

• k: positive integer, number of top elements to look for along the axis dimension.
• axis: positive integer, the dimension to look for elements.

forward(input)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class deepctr_torch.layers.sequence.SequencePoolingLayer(mode='mean', supports_masking=False, device='cpu')

The SequencePoolingLayer is used to apply pooling operation(sum,mean,max) on variable-length sequence feature/multi-value feature.

Input shape

• A list of two tensor [seq_value,seq_len]
• seq_value is a 3D tensor with shape: (batch_size, T, embedding_size)
• seq_len is a 2D tensor with shape: (batch_size, 1),indicate valid length of each sequence.

Output shape

• 3D tensor with shape: (batch_size, 1, embedding_size).

Arguments
• **mode**: str. Pooling operation to be used, can be sum, mean or max.

```python
forward(seq_value_len_list)
```

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

## 2.8 deepctr_torch.callbacks module

```python
class deepctr_torch.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False, save_weights_only=False, mode='auto', save_freq='epoch', **kwargs)
```

Save the model after every epoch.

*filepath* can contain named formatting options, which will be filled the value of *epoch* and keys in *logs* (passed in *on_epoch_end*).

For example: if *filepath* is *weights.{epoch:02d}-{val_loss:.2f}.hdf5*, then the model checkpoints will be saved with the epoch number and the validation loss in the filename.

**Arguments:**
- *filepath*: string, path to save the model file.
- *monitor*: quantity to monitor.
- *verbose*: verbosity mode, 0 or 1.
- *save_best_only*: if *save_best_only=True*, the latest best model according to the quantity monitored will not be overwritten.
- *mode*: one of {auto, min, max}. If *save_best_only=True*, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. For *val_acc*, this should be *max*, for *val_loss* this should be *min*, etc. In *auto* mode, the direction is automatically inferred from the name of the monitored quantity.
- *save_weights_only*: if True, then only the model’s weights will be saved

- period: Interval (number of epochs) between checkpoints.

```python
on_epoch_end(epoch, logs=None)
```

Called at the end of an epoch.

Subclasses should override for any actions to run. This function should only be called during TRAIN mode.

**Arguments:**
- *epoch*: integer, index of epoch.
- *logs*: dict, metric results for this training epoch, and for the validation epoch if validation is performed. Validation result keys are prefixed with *val_*. 

2.8. deepctr_torch.callbacks module 59
CHAPTER 3

Indices and tables

- genindex
- modindex
- search
Python Module Index

d
deeptorch.callbacks, 59
deeptorch.layers.core, 47
deeptorch.layers.interaction, 49
deeptorch.layers.sequence, 56
deeptorch.models.afm, 37
deeptorch.models.autoint, 44
deeptorch.models.basemodel, 30
deeptorch.models.ccpm, 31
deeptorch.models.dcn, 38
deeptorch.models.dcnmix, 39
deeptorch.models.deepfm, 34
deeptorch.models.dien, 41
deeptorch.models.din, 40
deeptorch.models.fibinet, 46
deeptorch.models.mlrlr, 35
deeptorch.models.nfm, 36
deeptorch.models.onn, 45
deeptorch.models.pnn, 32
deeptorch.models.wdl, 33
deeptorch.models.xdeepfm, 43
Index

A
AFM (class in deepctr_torch.models.afm), 37
AFMLayer (class in deepctr_torch.layers.interaction), 49
AGRUCell (class in deepctr_torch.layers.sequence), 56
AttentionSequencePoolingLayer (class in deepctr_torch.layers.sequence), 57
AUGRUCell (class in deepctr_torch.layers.sequence), 57
AutoInt (class in deepctr_torch.models.autoint), 44

B
BaseModel (class in deepctr_torch.models.basemodel), 30
BiInteractionPooling (class in deepctr_torch.layers.interaction), 50
BilinearInteraction (class in deepctr_torch.layers.interaction), 50

C
CCPM (class in deepctr_torch.models.ccpm), 31
CIN (class in deepctr_torch.layers.interaction), 51
compile() (deepctr_torch.models.basemodel.BaseModel method), 30
Conv2dSame (class in deepctr_torch.layers.core), 47
ConvLayer (class in deepctr_torch.layers.interaction), 51
CrossNet (class in deepctr_torch.layers.interaction), 52
CrossNetMix (class in deepctr_torch.layers.interaction), 53

D
DCN (class in deepctr_torch.models.dcn), 38
DCNMix (class in deepctr_torch.models.dcnmix), 39
depctr_torch.callbacks (module), 59
depctr_torch.layers.core (module), 47
depctr_torch.layers.interaction (module), 49
depctr_torch.layers.sequence (module), 56
depctr_torch.models.afm (module), 37
depctr_torch.models.autoint (module), 44
depctr_torch.models.basemodel (module), 30
depctr_torch.models.ccpm (module), 31
depctr_torch.models.dcn (module), 38
depctr_torch.models.dcnmix (module), 39
depctr_torch.models.deepfm (module), 34
depctr_torch.models.dien (module), 41
depctr_torch.models.din (module), 40
depctr_torch.models.fibinet (module), 46
depctr_torch.models.mlr (module), 35
depctr_torch.models.nfm (module), 36
depctr_torch.models.onn (module), 45
depctr_torch.models.pnn (module), 32
depctr_torch.models.wdl (module), 33
depctr_torch.models.xdeepfm (module), 43
DeepFM (class in deepctr_torch.models.deepfm), 34
DIEN (class in deepctr_torch.models.dien), 41
DIN (class in deepctr_torch.models.din), 40
DNN (class in deepctr_torch.layers.core), 47
DynamicGRU (class in deepctr_torch.layers.sequence), 58

evaluate() (deepctr_torch.models.basemodel.BaseModel method), 30

F
FiBiNET (class in deepctr_torch.models.fibinet), 46
fit() (deepctr_torch.models.basemodel.BaseModel method), 30
FM (class in deepctr_torch.layers.interaction), 53
forward() (deepctr_torch.layers.core.Conv2dSame method), 47
forward() (deepctr_torch.layers.core.DNN method), 48
forward() (deepctr_torch.layers.core.LocalActivationUnit method), 48
forward() (deepctr_torch.layers.core.PredictionLayer
method), 49
forward() (deepctr_torch.layers.interaction.AFMLayer
forward() (deepctr_torch.models.dien.InterestExtractor
method), 49
forward() (deepctr_torch.layers.interaction.BiInteractionPooling
method), 41
forward() (deepctr_torch.models.fibinet.FiBiNET
method), 47
forward() (deepctr_torch.layers.interaction.BilinearInteraction
method), 49
forward() (deepctr_torch.layers.interaction.CIN
method), 50
forward() (deepctr_torch.layers.interaction.ConvLayer
method), 50
forward() (deepctr_torch.layers.interaction.CrossNet
method), 52
forward() (deepctr_torch.layers.interaction.CrossNetMix
method), 52
forward() (deepctr_torch.layers.interaction.FM
method), 53
forward() (deepctr_torch.layers.interaction.InnerProductLayer
method), 54
forward() (deepctr_torch.layers.interaction.InteractingLayer
method), 54
forward() (deepctr_torch.layers.interaction.OutterProductLayer
method), 55
forward() (deepctr_torch.layers.interaction.SENETLayer
method), 56
forward() (deepctr_torch.layers.sequence.AGRUCel
method), 56
forward() (deepctr_torch.models.autoint.AutoInt
method), 45
forward() (deepctr_torch.models.ccpm.CCPM
method), 32
forward() (deepctr_torch.models.dcn.DCN
method), 39
forward() (deepctr_torch.models.dcnmix.DCNMix
method), 40
forward() (deepctr_torch.models.deepfm.DeepFM
method), 35
forward() (deepctr_torch.models.dien.DIEN
method), 42
forward() (deepctr_torch.models.dien.InterestEvolution
method), 42
forward() (deepctr_torch.models.dien.InterestExtractor
method), 43
forward() (deepctr_torch.models.mlr.MLR
method), 36
forward() (deepctr_torch.models.nfm.NFM
method), 36
forward() (deepctr_torch.models.onn.ONN
method), 46
forward() (deepctr_torch.models.pnn.PNN
method), 33
forward() (deepctr_torch.models.xdeepfm.xDeepFM
method), 44
forward() (deepctr_torch.models.wdl.WDL
method), 34
forward() (deepctr_torch.models.xdeepfm.xDeepFM
method), 44
predict() (deepctr_torch.models.basemodel.BaseModel method), 31

PredictionLayer (class in deepctr_torch.layers.core), 48

SENETLayer (class in deepctr_torch.layers.interaction), 56

SequencePoolingLayer (class in deepctr_torch.layers.sequence), 58

WDL (class in deepctr_torch.models.wdl), 33

xDeepFM (class in deepctr_torch.models.xdeepfm), 43