DeepCTR Documentation

Release 0.8.0

Weichen Shen

Sep 11, 2020

Home:

1 News **3**

2 DisscussionGroup **5**
- 2.1 Quick-Start . . . 7
- 2.2 Features . . . 10
- 2.3 Examples . . . 30
- 2.4 FAQ . . . 38
- 2.5 History . . . 41
- 2.6 DeepCTR Models API . . . 42
- 2.7 DeepCTR Estimators API . . . 63
- 2.8 DeepCTR Layers API . . . 75

3 Indices and tables **99**

Python Module Index **101**

Index **103**

DeepCTR is a **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 easily build custom models.You can use any complex model with `model.fit()` and `model.predict()`.

- Provide `tf.keras.Model` like interface for **quick experiment**. example
- Provide `tensorflow estimator` interface for **large scale data** and **distributed training**. example
- It is compatible with both `tf 1.x` and `tf 2.x`.

Let's Get Started! (Chinese Introduction)

You can read the latest code and related projects

- DeepCTR: https://github.com/shenweichen/DeepCTR
- DeepMatch: https://github.com/shenweichen/DeepMatch
- DeepCTR-Torch: https://github.com/shenweichen/DeepCTR-Torch

CHAPTER 1

News

06/27/2020 : Support `Tensorflow Estimator` for large scale data and distributed training.Support different initializers for different embedding weights and loading pretrained embeddings.Add new model `FwFM`. Changelog

05/17/2020 : Fix numerical instability in `LayerNormalization`. Changelog

03/15/2020 : Add FLEN () and `FieldWiseBiInteraction`. Changelog

CHAPTER 2

DisscussionGroup

wechat ID: **deepctrbot**



浅梦的学习笔记

微信扫描二维码，关注我的公众号

2.1 Quick-Start

2.1.1 Installation Guide

Now `deepctr` is available for python `2.7`and `3.5, 3.6, 3.7`.`deepctr` depends on tensorflow, you can specify to install the cpu version or gpu version through `pip`.

CPU version

```
$ pip install deepctr[cpu]
```

GPU version

```
$ pip install deepctr[gpu]
```

2.1.2 Getting started: 4 steps to DeepCTR

Step 1: Import model

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.model_selection import train_test_split
from deepctr.models import DeepFM
from deepctr.feature_column 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 we have two methods to encode the sparse categorical feature for embedding

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

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

- Hash Encoding: map the features to a fix range,like 0 ~ 9999.We have 2 methods to do that:
  - Do feature hashing before training

```
for feat in sparse_features:
    lbe = HashEncoder()
    data[feat] = lbe.transform(data[feat])
```

- Do feature hashing on the fly in training process

  We can do feature hashing by setting `use_hash=True` in `SparseFeat` or `VarlenSparseFeat` in Step3.

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

```
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.

And for varlen(multi-valued) sparse features,you can use VarlenSparseFeat. Visit examples of using `VarlenSparseFeat`

- Label Encoding

```
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

```
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 = fixlen_feature_columns
linear_feature_columns = fixlen_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].values for name in feature_names}
test_model_input = {name:test[name].values for name in feature_names}


model = DeepFM(linear_feature_columns,dnn_feature_columns,task='binary')
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.

2.1.3 Getting started: 4 steps to DeepCTR Estimator with TFRecord

Step 1: Import model

```
import tensorflow as tf

from deepctr.estimator.inputs import input_fn_tfrecord
from deepctr.estimator.models import DeepFMEstimator
```

Step 2: Generate feature columns for linear part and dnn part

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

dnn_feature_columns = []
linear_feature_columns = []

for i, feat in enumerate(sparse_features):
    dnn_feature_columns.append(tf.feature_column.embedding_column(
        tf.feature_column.categorical_column_with_identity(feat, 1000), 4))
    linear_feature_columns.append(tf.feature_column.categorical_column_with_
↪identity(feat, 1000))
for feat in dense_features:
    dnn_feature_columns.append(tf.feature_column.numeric_column(feat))
    linear_feature_columns.append(tf.feature_column.numeric_column(feat))
```

Step 3: Generate the training samples with TFRecord format

```
feature_description = {k: tf.FixedLenFeature(dtype=tf.int64, shape=1) for k in sparse_
↪features}
feature_description.update(
    {k: tf.FixedLenFeature(dtype=tf.float32, shape=1) for k in dense_features})
feature_description['label'] = tf.FixedLenFeature(dtype=tf.float32, shape=1)

train_model_input = input_fn_tfrecord('./criteo_sample.tr.tfrecords', feature_
↪description, 'label', batch_size=256,
                                      num_epochs=1, shuffle_factor=10)
test_model_input = input_fn_tfrecord('./criteo_sample.te.tfrecords', feature_
↪description, 'label',
                                     batch_size=2 ** 14, num_epochs=1, shuffle_
↪factor=0)
```

Step 4: Train and evaluate the model

```
model = DeepFMEstimator(linear_feature_columns, dnn_feature_columns, task='binary')

model.train(train_model_input)
eval_result = model.evaluate(test_model_input)

print(eval_result)
```

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 prediction task.

DNN based CTR prediction models usually have 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, embeddings_initializer, embedding_name, group_name, trainable)`

- name : feature name
- vocabulary_size : number of unique feature values for sprase feature or hashing space when `use_hash=True`
- embedding_dim : embedding dimension
- use_hash : defualt `False`.If `True` the input will be hashed to space of size `vocabulary_size`.
- dtype : default `float32`.dtype of input tensor.
- embeddings_initializer : initializer for the `embeddings` matrix.
- embedding_name : default `None`. If None, the embedding_name will be same as `name`.
- group_name : feature group of this feature.
- trainable: default `True`.Whether or not the embedding is trainable.

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, weight_name,weight_norm)`

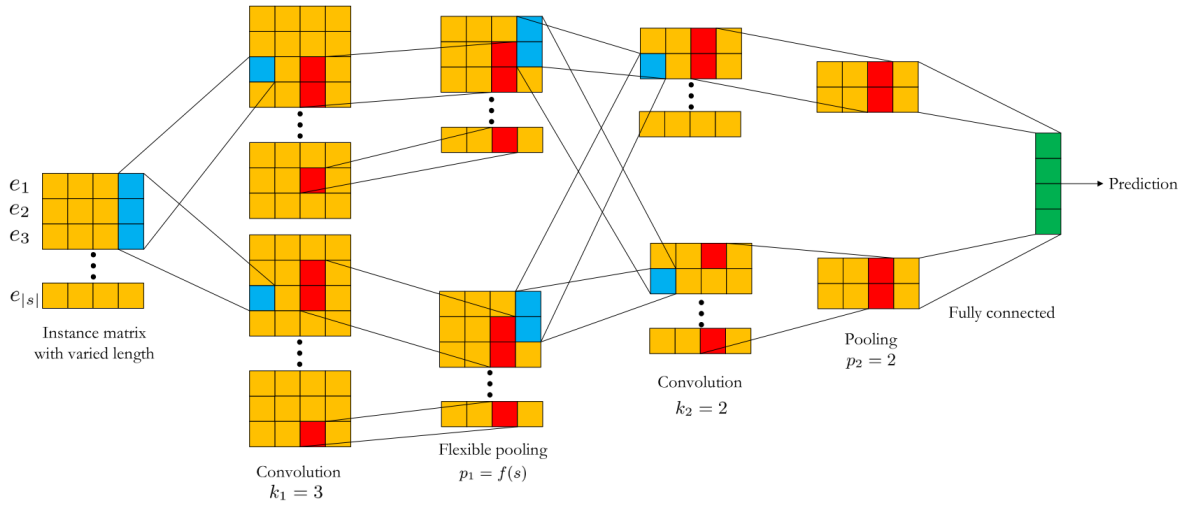
- sparsefeat : a instance of `SparseFeat`
- maxlen : maximum length of this feature for all samples
- combiner : pooling method,can be `sum`,`mean` or `max`
- length_name : feature length name,if `None`, value 0 in feature is for padding.
- weight_name : default `None`. If not None, the sequence feature will be multiplyed by the feature whose name is `weight_name`.
- weight_norm : default `True`. Whether normalize the weight score or not.

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 CCPM Estimator API



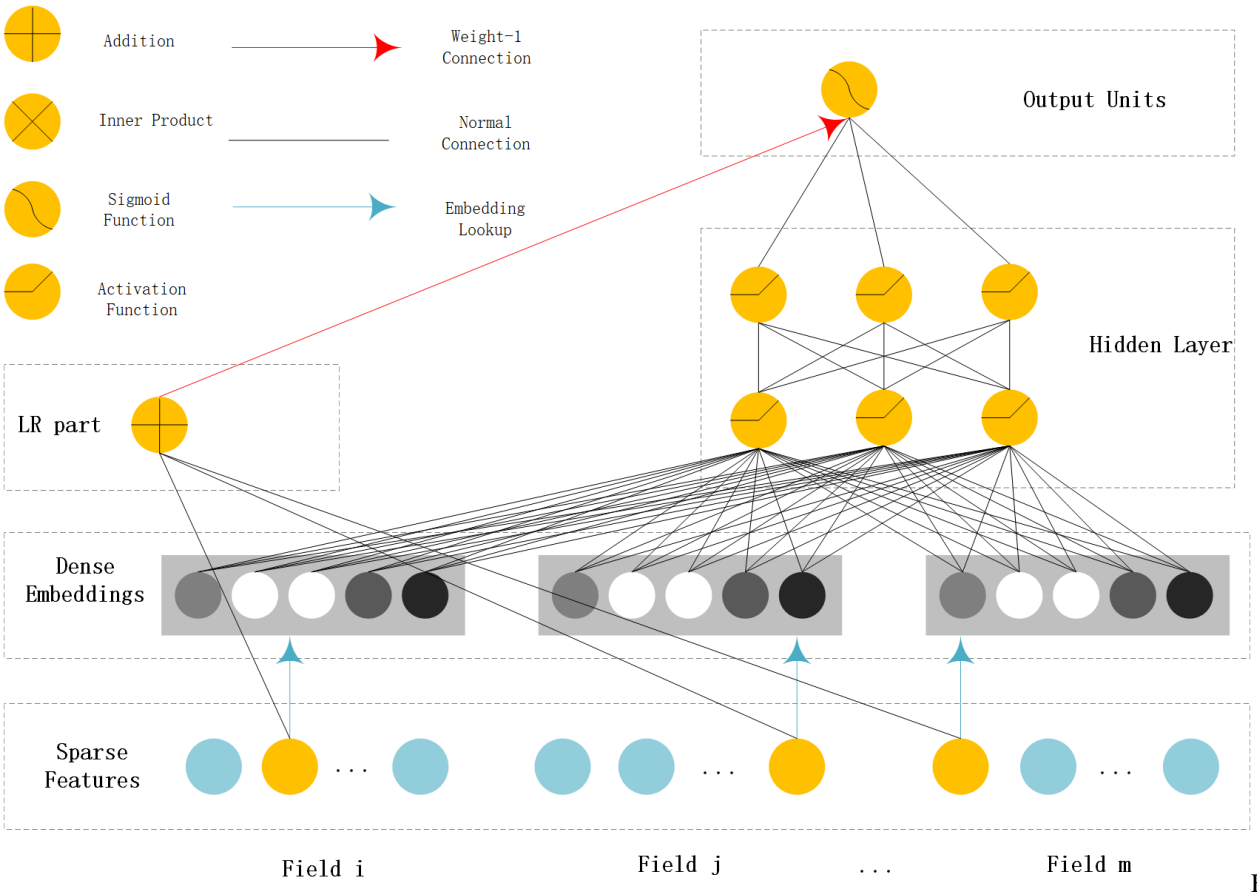
CCPM

Liu Q, Yu F, Wu S, et al. A convolutional click prediction model[C]//Proceedings of the 24th ACM International on Conference on Information and Knowledge Management. ACM, 2015: 1743-1746.

FNN (Factorization-supported Neural Network)

According to the paper,FNN learn embedding vectors of categorical data via pre-trained FM. It use FM's latent vector to initialiaze the embedding vectors.During the training stage,it concatenates the embedding vectors and feeds them into a MLP(MultiLayer Perceptron).

FNN Model API FNN Estimator API



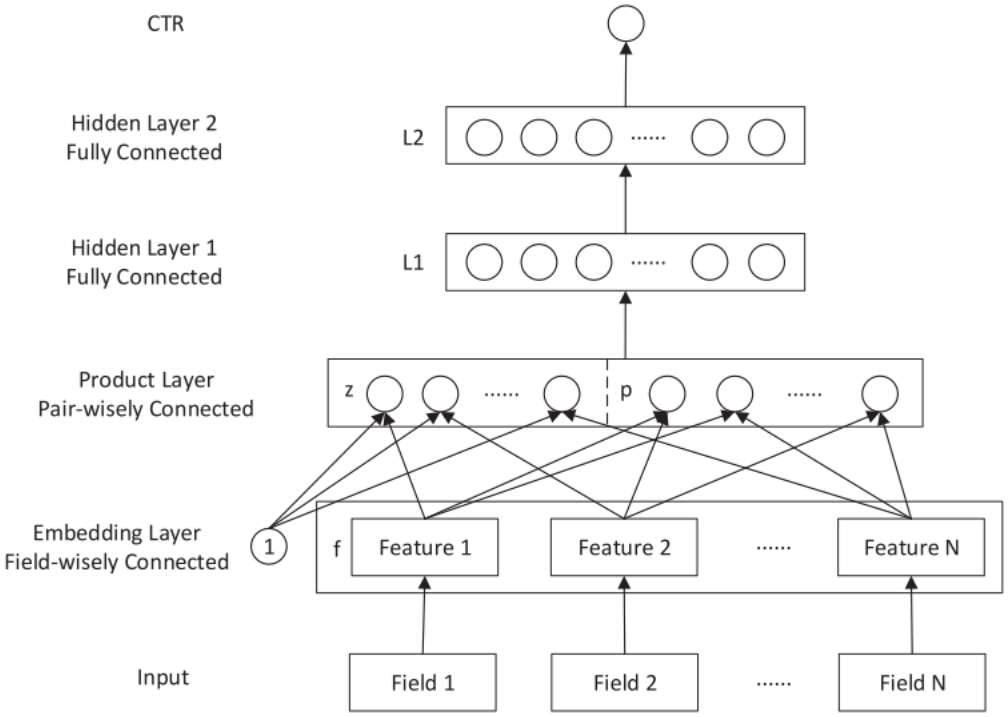
FNN

Zhang W, Du T, Wang J. Deep learning over multi-field categorical data[C]//European conference on information retrieval. Springer, Cham, 2016: 45-57.

PNN (Product-based Neural Network)

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

PNN Model API PNN Estimator API



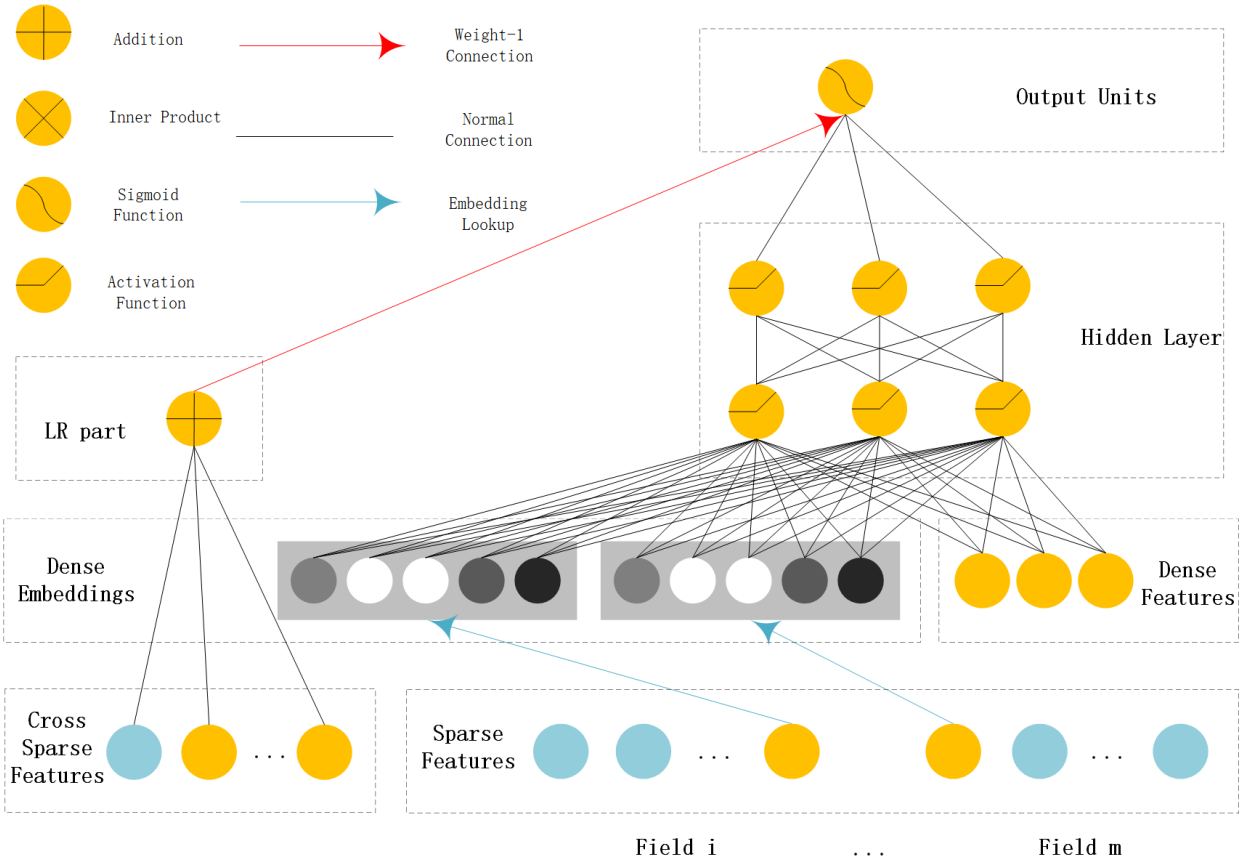
PNN

Qu Y, Cai H, Ren K, et al. Product-based neural networks for user response prediction[C]//Data Mining (ICDM), 2016 IEEE 16th International Conference on. IEEE, 2016: 1149-1154.

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 WDL Estimator API



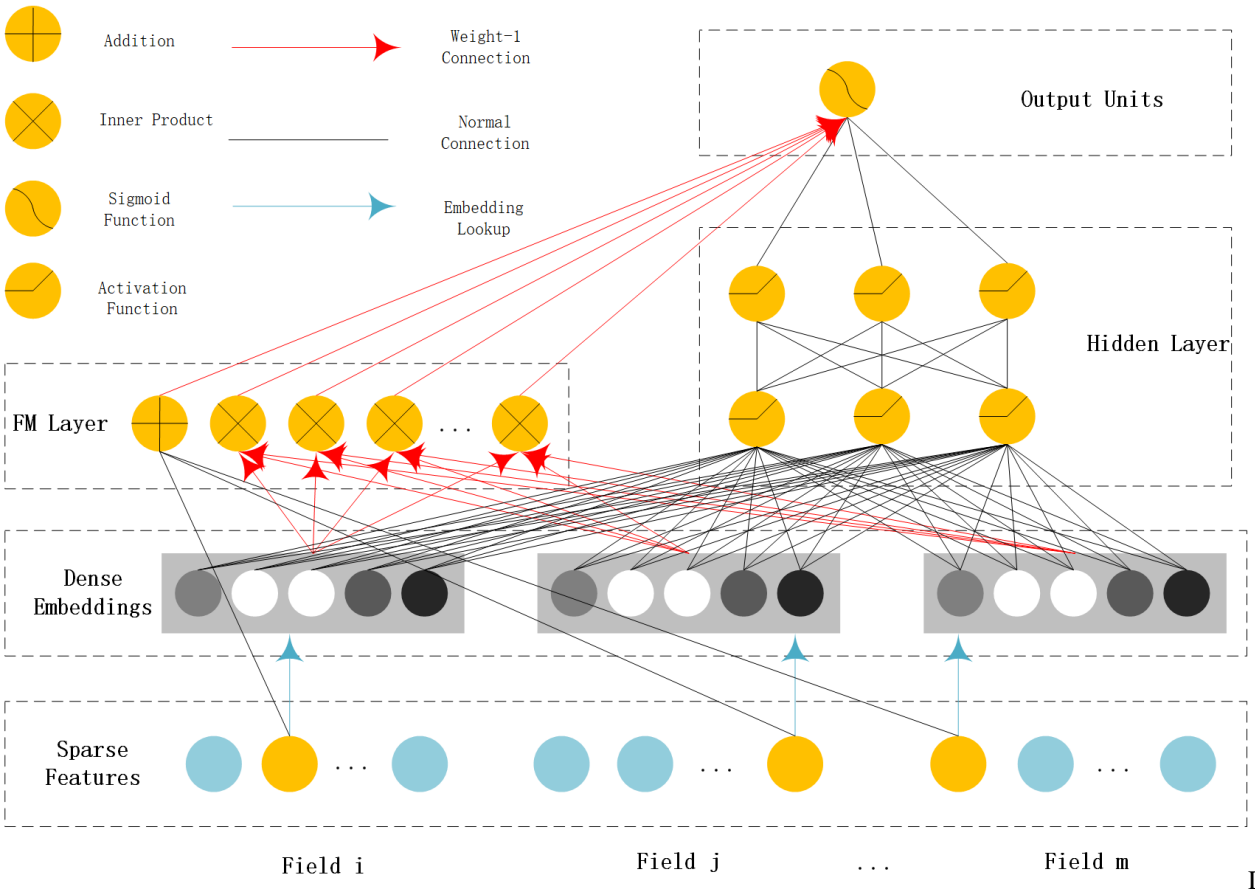
WDL

Cheng H T, Koc L, Harmsen J, et al. Wide & deep learning for recommender systems[C]//Proceedings of the 1st Workshop on Deep Learning for Recommender Systems. ACM, 2016: 7-10.

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 initialiaze,they are learned end2end.

DeepFM Model API DeepFM Estimator API



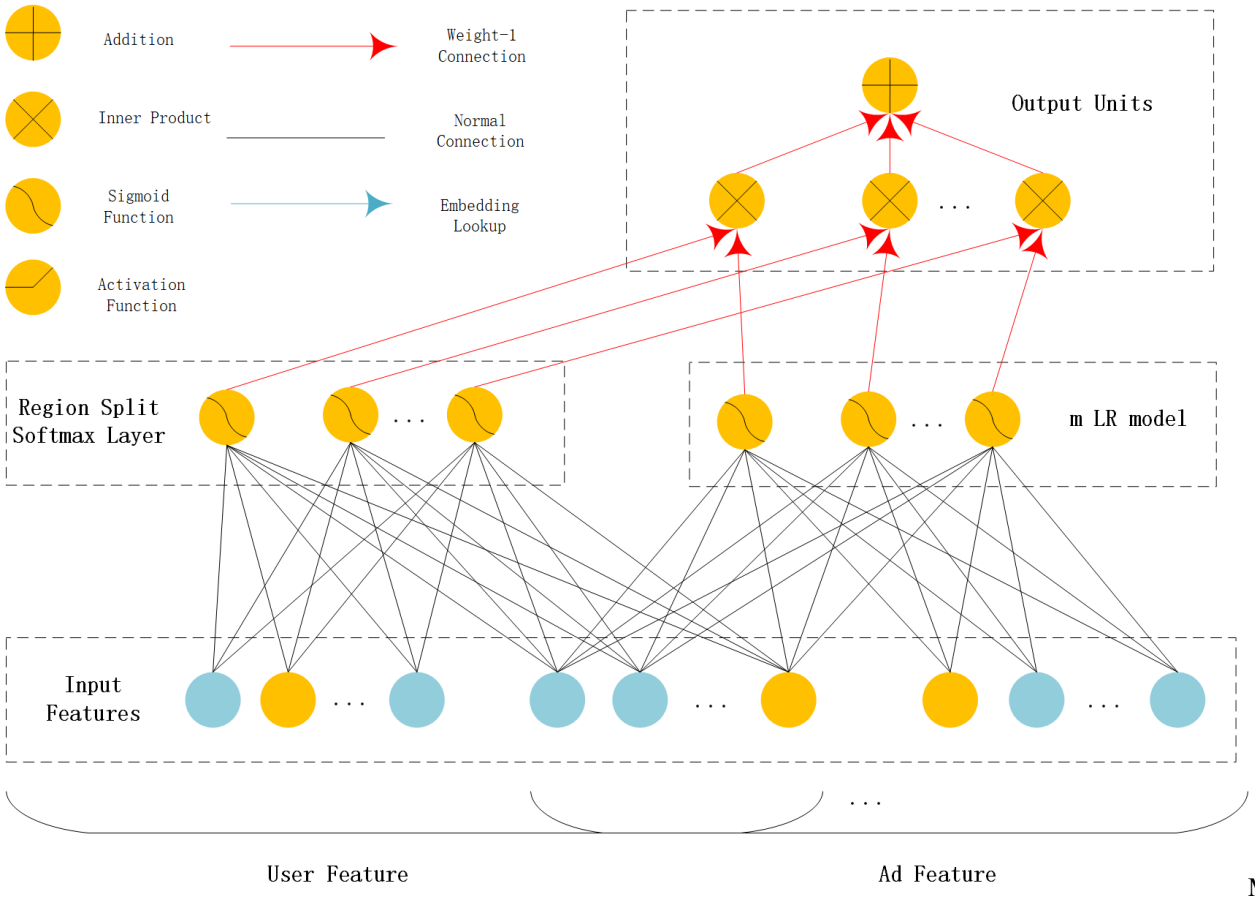
DeepFM

Guo H, Tang R, Ye Y, et al. Deepfm: a factorization-machine based neural network for ctr prediction[J]. arXiv preprint arXiv:1703.04247, 2017.

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 learn 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



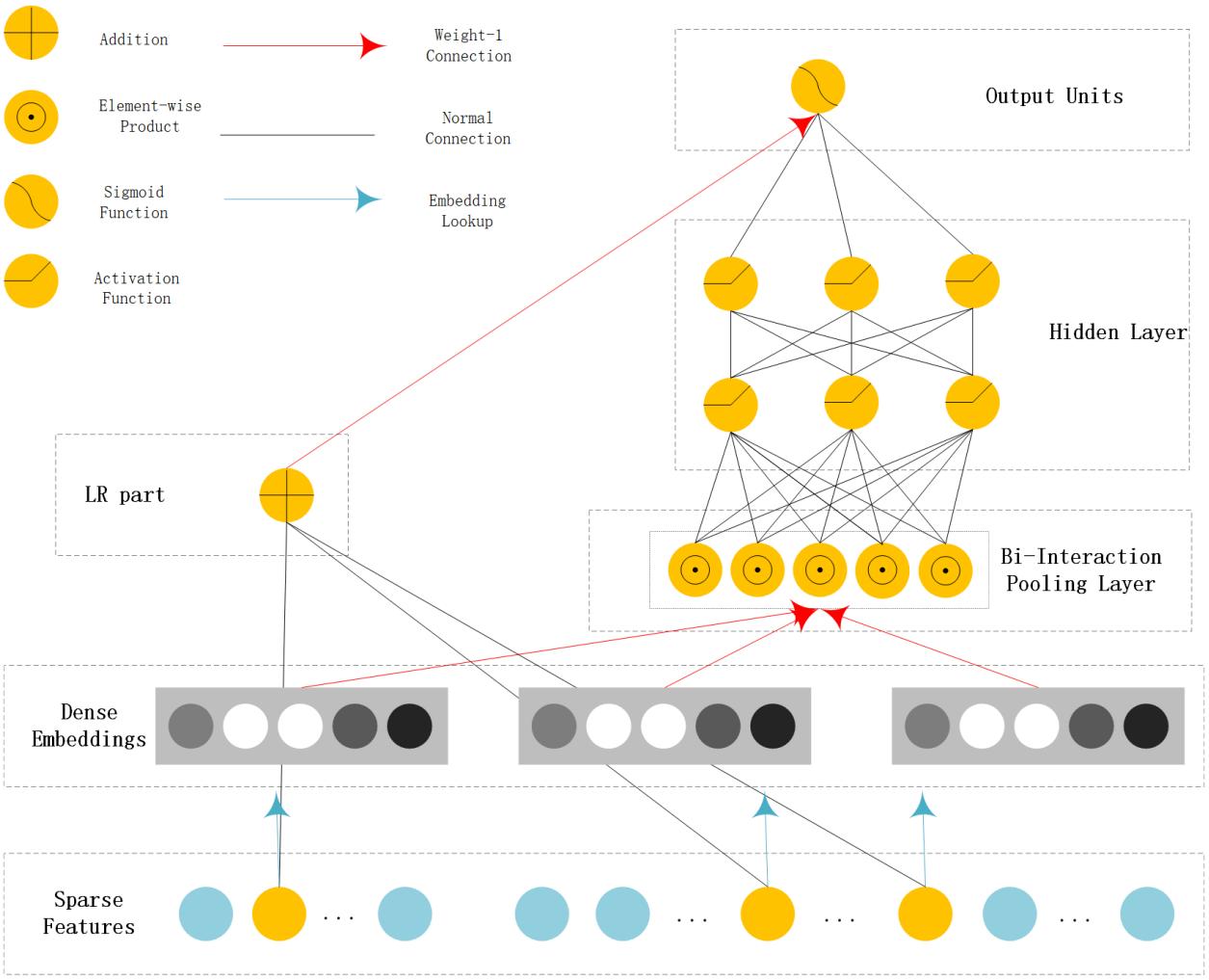
MLR

Gai K, Zhu X, Li H, et al. Learning Piece-wise Linear Models from Large Scale Data for Ad Click Prediction[J]. arXiv preprint arXiv:1704.05194, 2017.

NFM (Neural Factorization Machine)

NFM use a bi-interaction pooling layer to learn feature interaction between embedding vectors and compress the result into a singe 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 NFM Estimator API



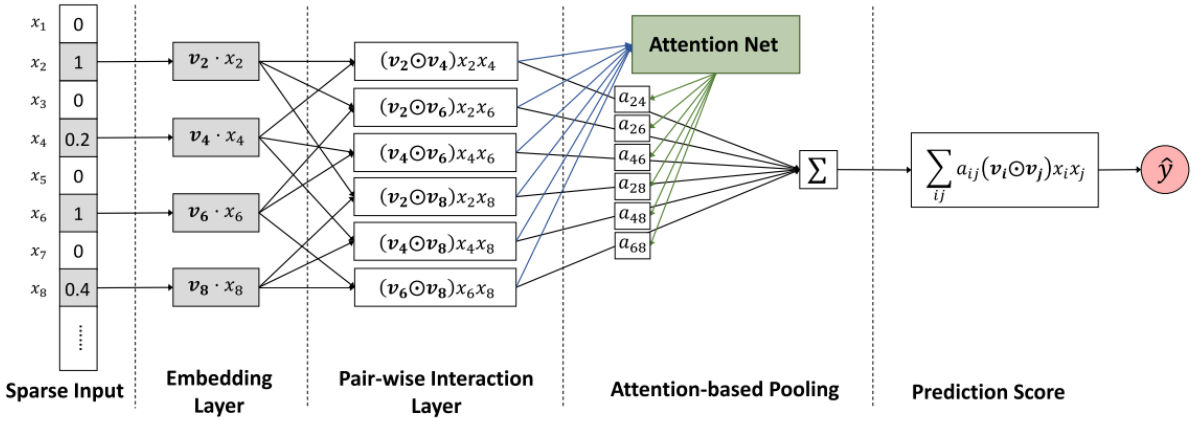
NFM

He X, Chua T S. Neural factorization machines for sparse predictive analytics[C]//Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval. ACM, 2017: 355-364.

AFM (Attentional Factorization Machine)

AFM is a variant of FM,tradional 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 AFM Estimator API



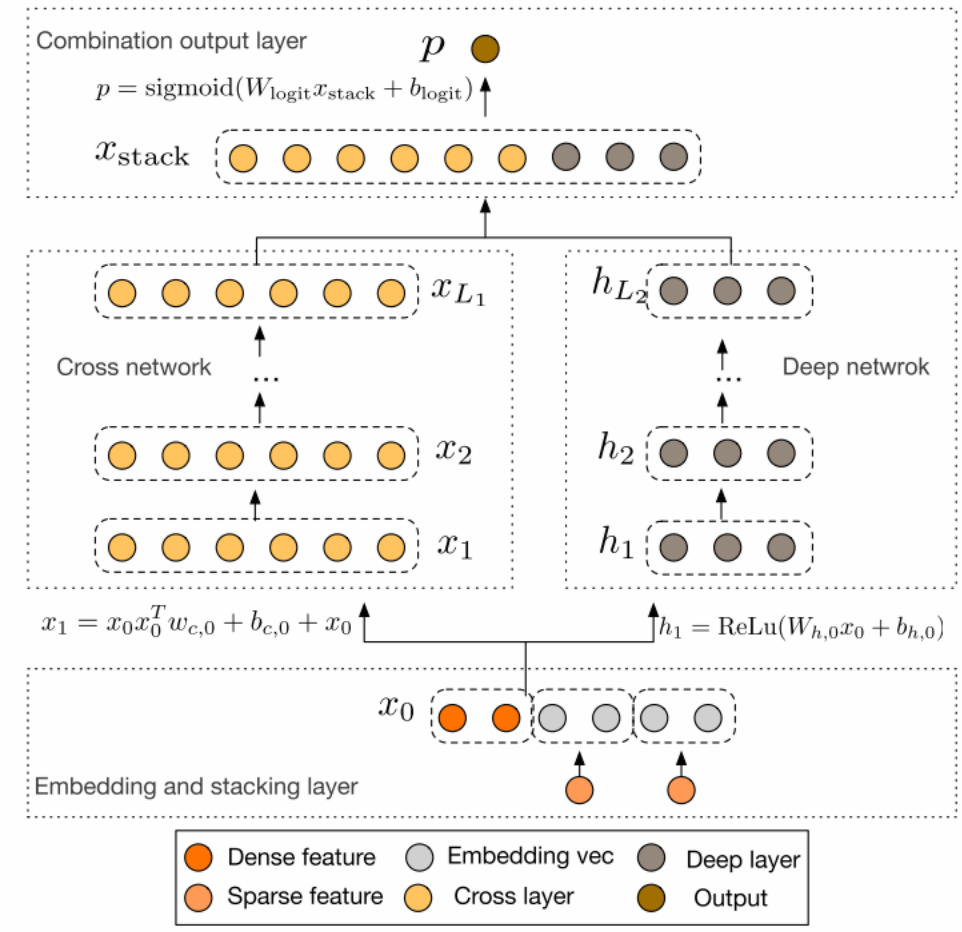
AFM

Xiao J, Ye H, He X, et al. Attentional factorization machines: Learning the weight of feature interactions via attention networks[J]. arXiv preprint arXiv:1708.04617, 2017.

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 feed into one fully connected layer to get the prediction probability.

DCN Model API DCN Estimator API



DCN

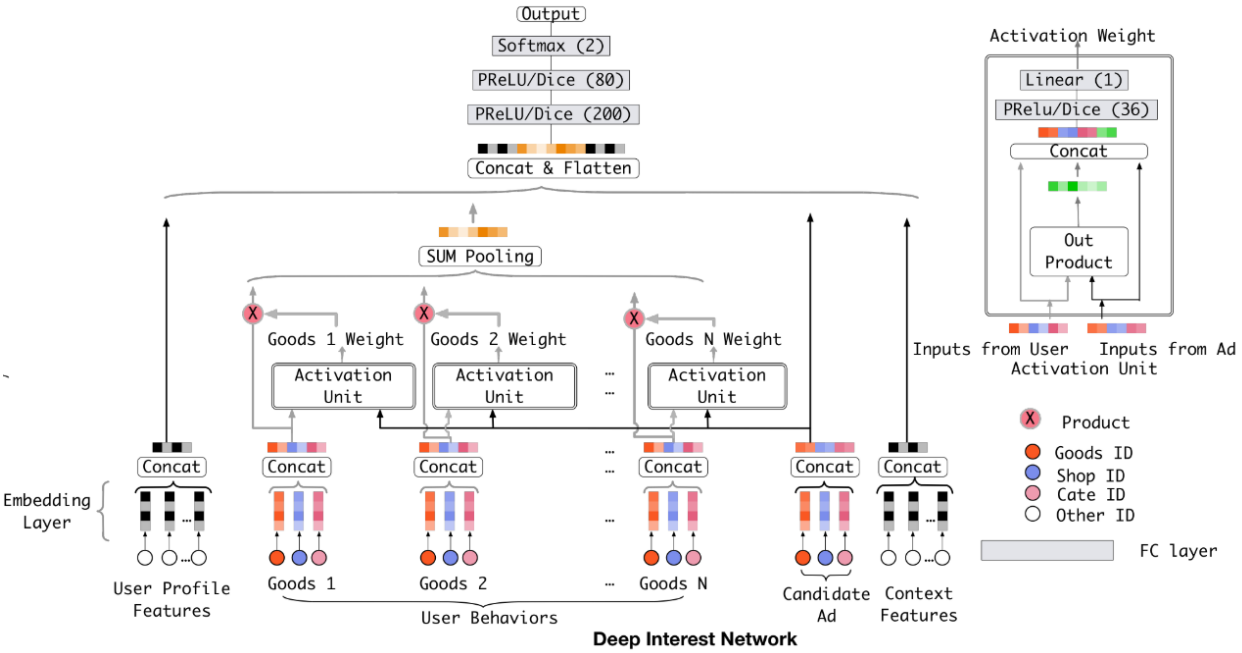
Wang R, Fu B, Fu G, et al. Deep & cross network for ad click predictions[C]//Proceedings of the ADKDD'17. ACM, 2017: 12.

DIN (Deep Interest Network)

DIN introduce a attention method to learn from sequence(multi-valued) feature. Tradional 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



DIN

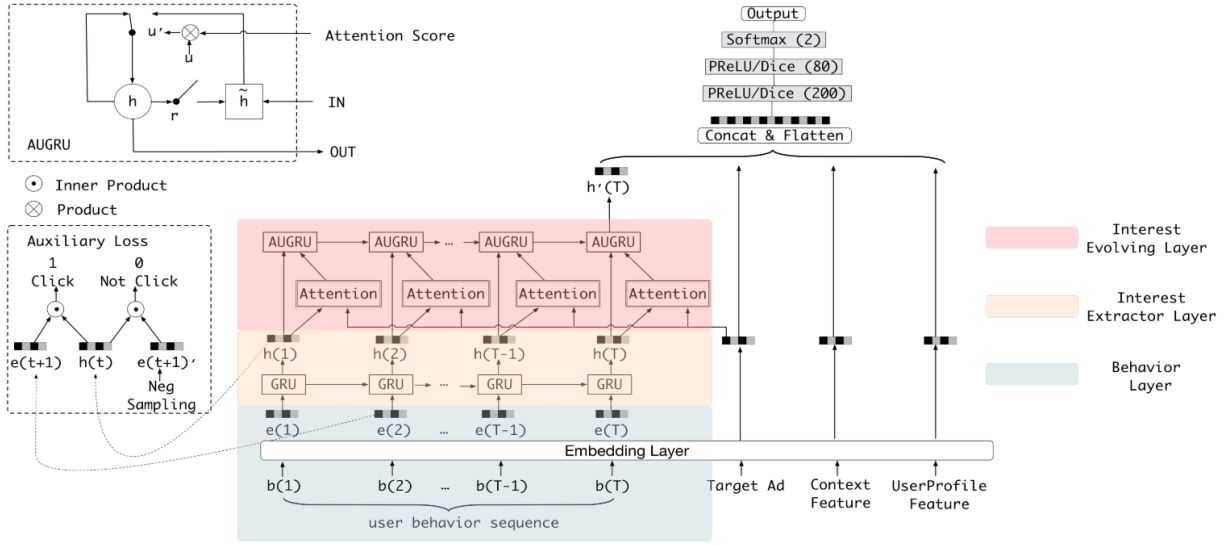
Zhou G, Zhu X, Song C, et al. Deep interest network for click-through rate prediction[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. ACM, 2018: 1059-1068.

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



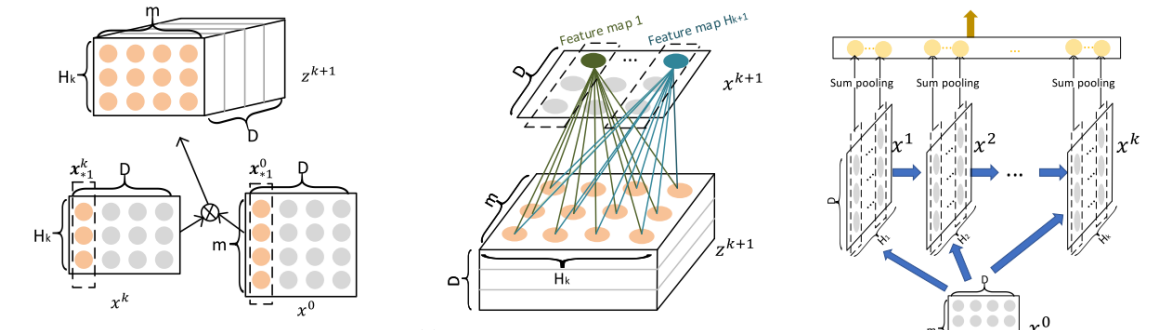
DIEN

Zhou G, Mou N, Fan Y, et al. Deep Interest Evolution Network for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1809.03672, 2018.

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 and 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 xDeepFM Estimator API



(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 (aslo known as *feature maps*).

(c) An overview of the CIN architecture.

Figure 4: Components and architecture of the Compressed Interaction Network (CIN).

CIN

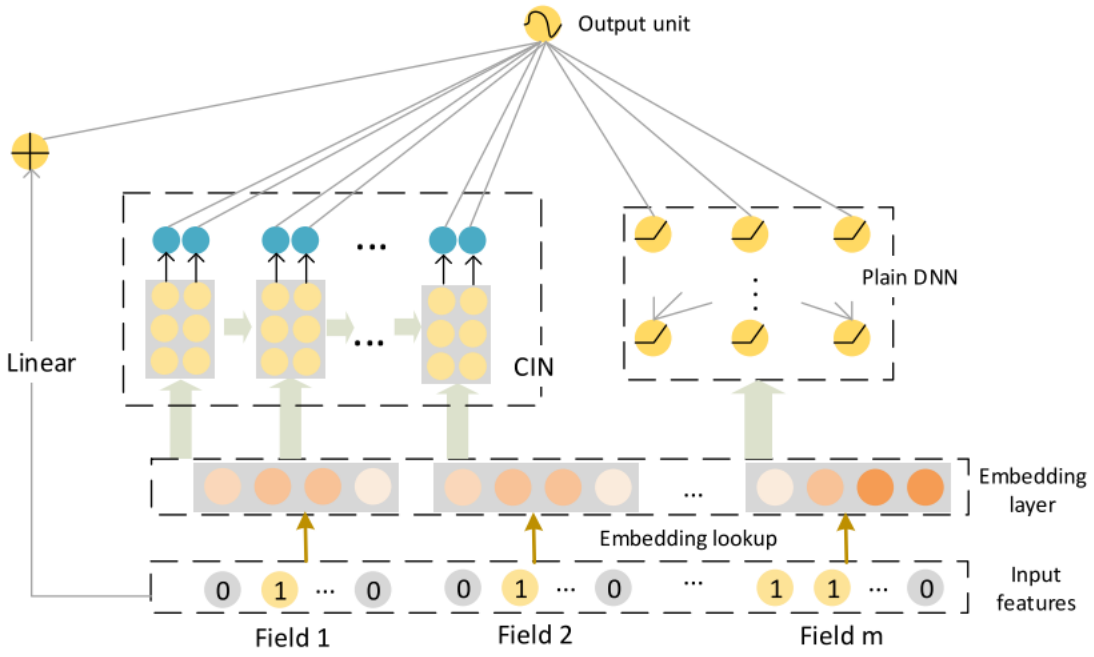


Figure 5: The architecture of xDeepFM.

xDeepFM

Lian J, Zhou X, Zhang F, et al. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems[J]. arXiv preprint arXiv:1803.05170, 2018.

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 AutoInt Estimator API

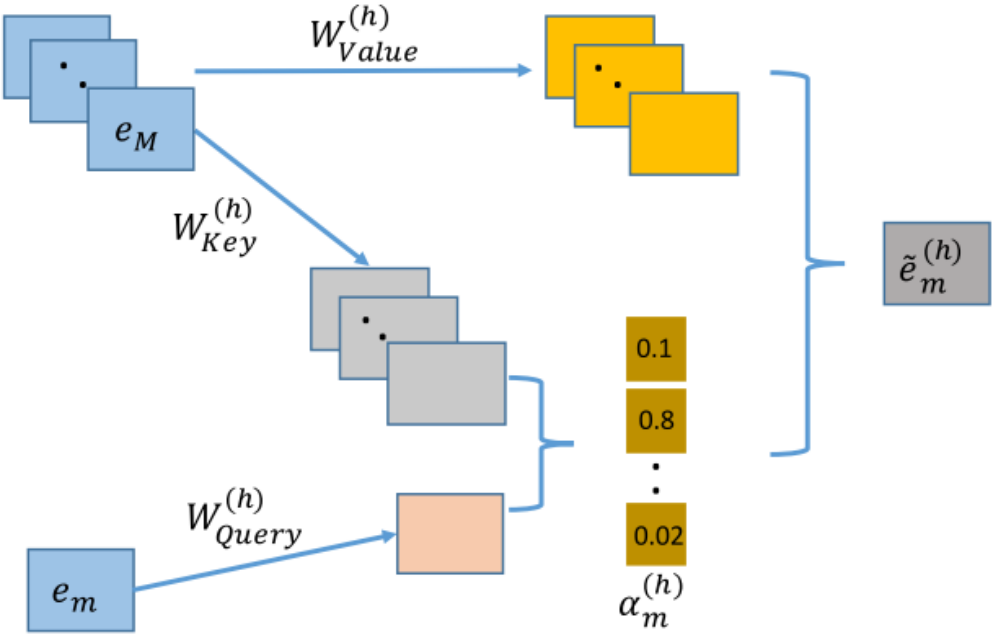


Figure 3: The architecture of interacting layer. Combinatorial features are conditioned on attention weights, i.e., $\alpha_m^{(h)}$.

InteractingLayer

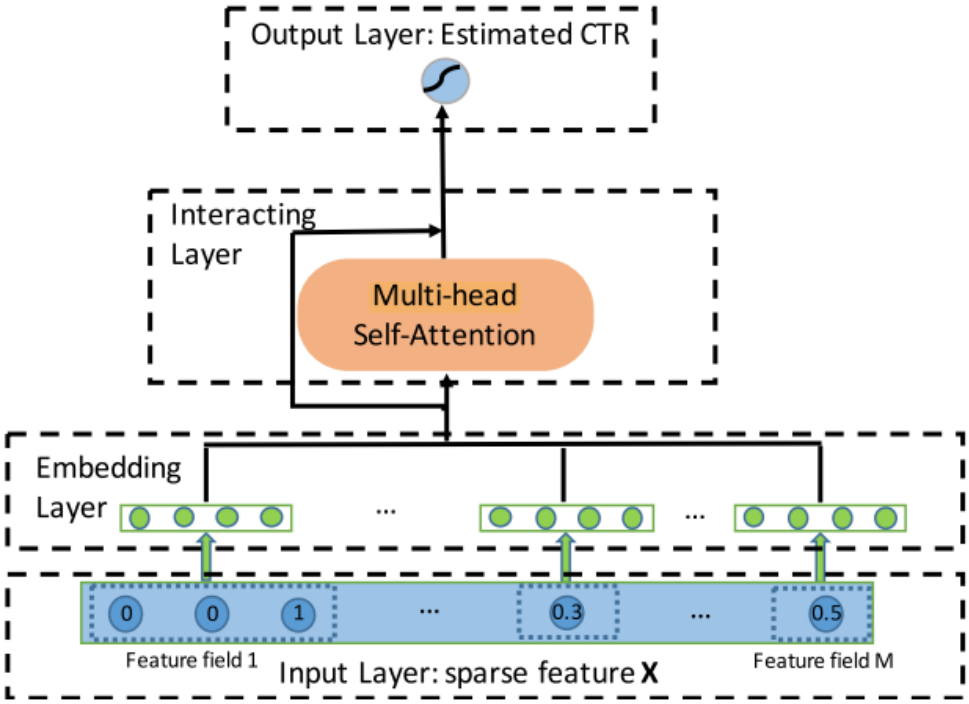


Figure 1: Overview of our proposed model AutoInt. The details of embedding layer and interacting layer are illustrated in Figure 2 and Figure 3 respectively.

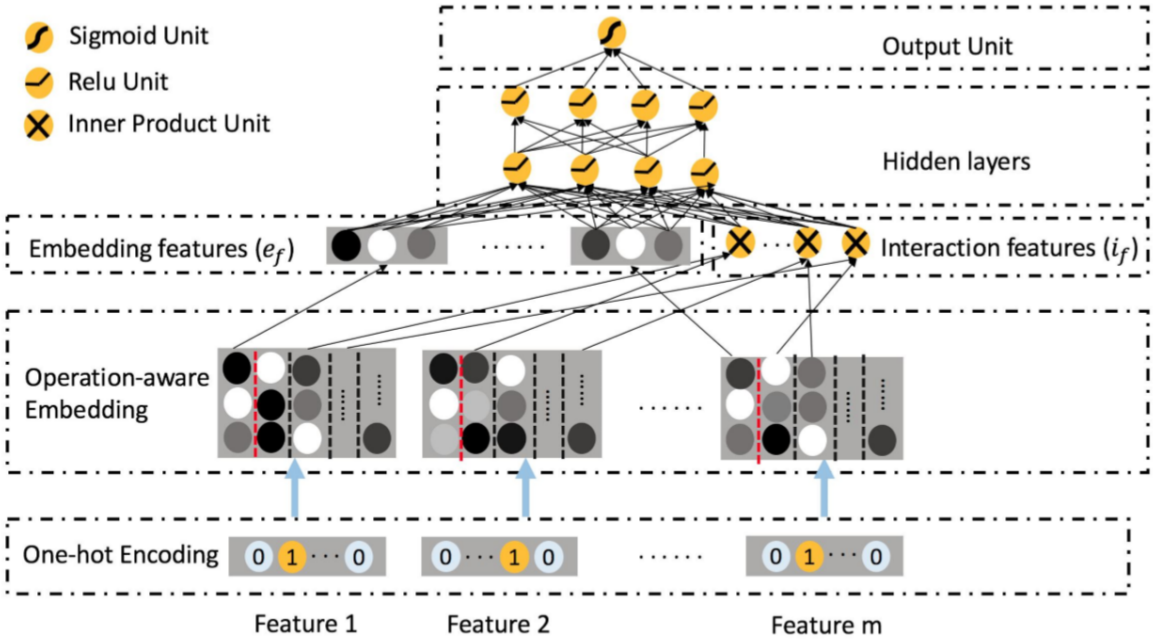
AutoInt

Song W, Shi C, Xiao Z, et al. AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks[J]. arXiv preprint arXiv:1810.11921, 2018.

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



ONN

Yang Y, Xu B, Shen F, et al. Operation-aware Neural Networks for User Response Prediction[J]. arXiv preprint arXiv:1904.12579, 2019.

FGCNN(Feature Generation by Convolutional Neural Network)

FGCNN models with two components: Feature Generation and Deep Classifier. Feature Generation leverages the strength of CNN to generate local patterns and recombine them to generate new features. Deep Classifier adopts the structure of IPNN to learn interactions from the augmented feature space.

FGCNN Model API

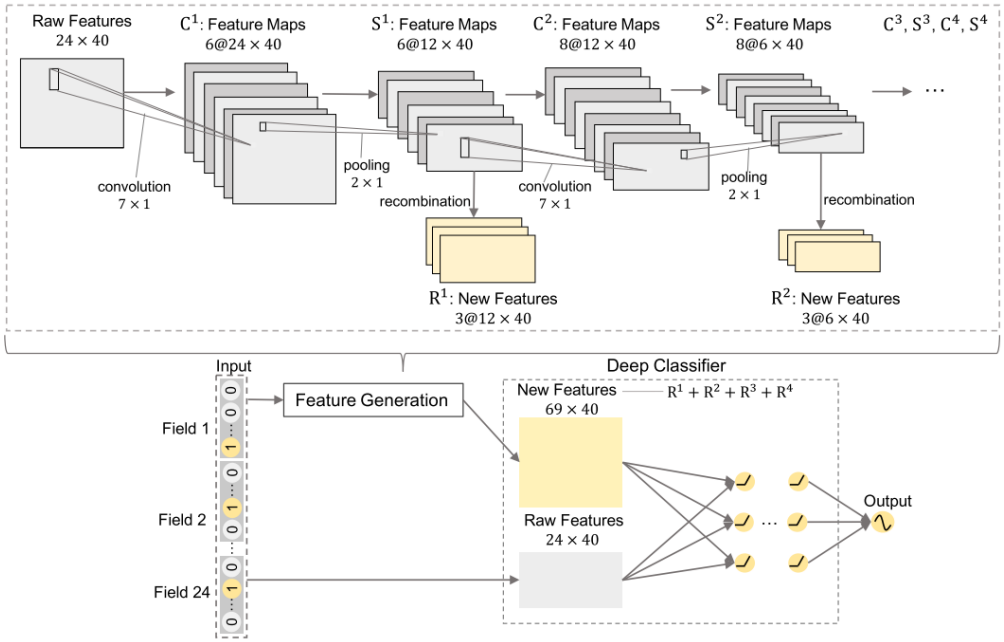


Figure 2: An overview of Feature Generation by Convolutional Neural Network Model (The hyper-parameters in the figure are the best setting of FGCNN on Avazu Dataset)

FGCNN

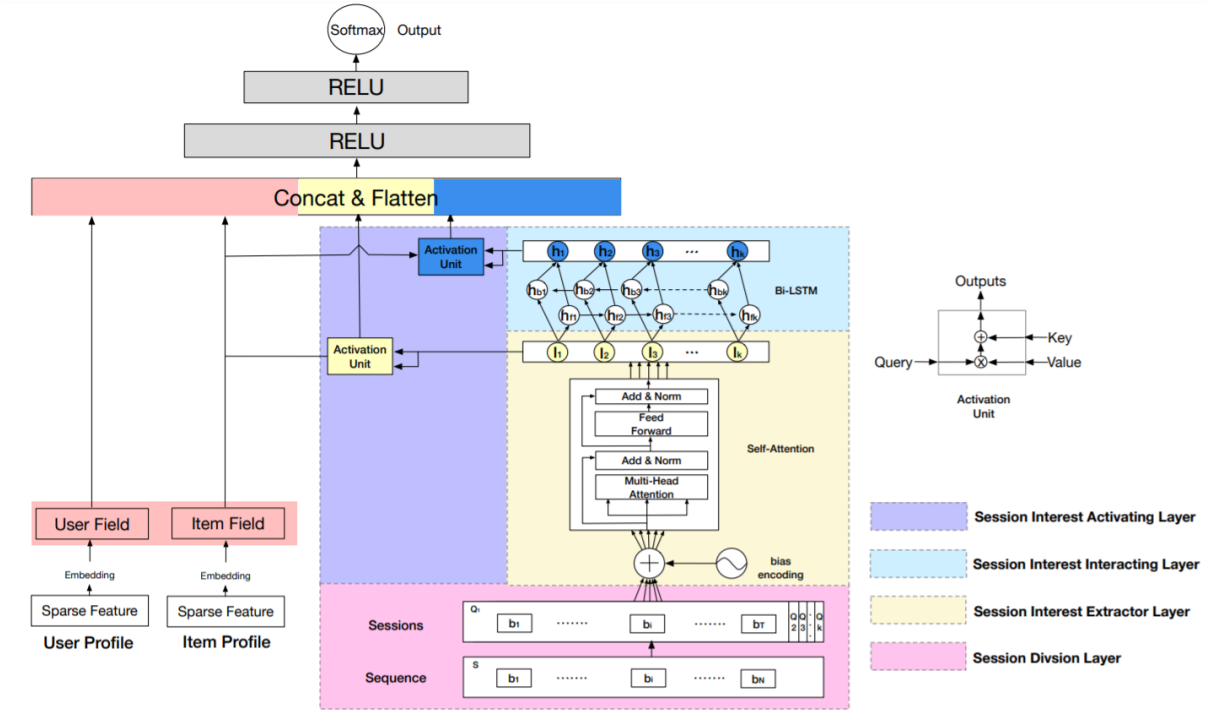
Liu B, Tang R, Chen Y, et al. Feature Generation by Convolutional Neural Network for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1904.04447, 2019.

DSIN(Deep Session Interest Network)

Deep Session Interest Network (DSIN) extracts users' multiple historical sessions in their behavior sequences. First it uses self-attention mechanism with bias encoding to extract users' interests in each session. Then apply Bi-LSTM to model how users' interests evolve and interact among sessions. Finally, local activation unit is used to adaptively learn the influences of various session interests on the target item.

DSIN Model API

DSIN example



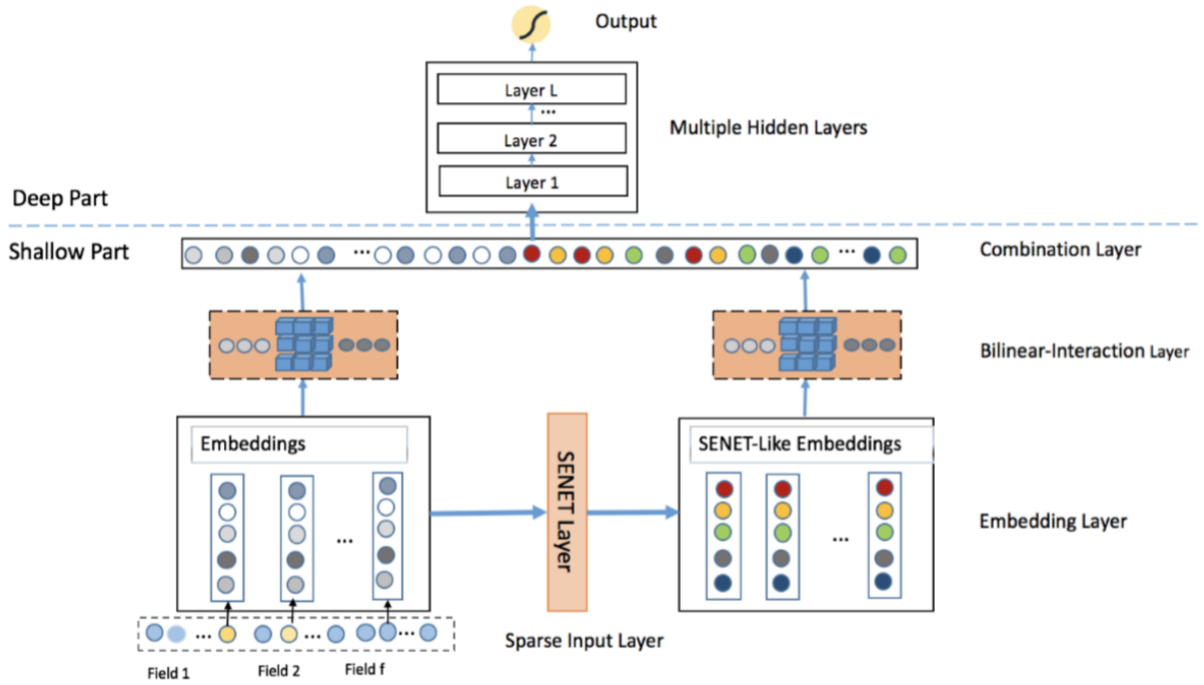
DSIN

Feng Y, Lv F, Shen W, et al. Deep Session Interest Network for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1905.06482, 2019.

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 Estimator API



FiBiNET

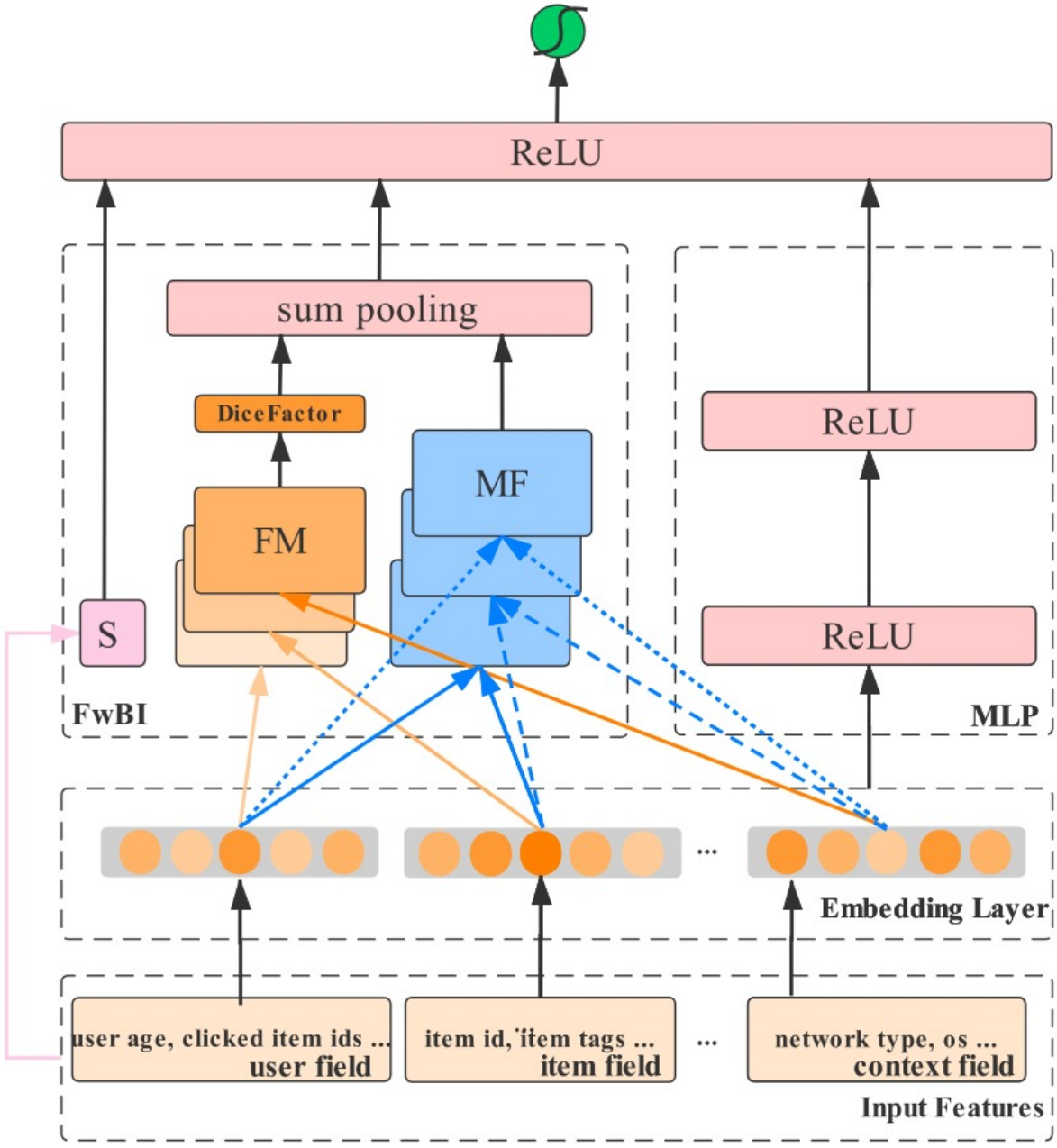
Huang T, Zhang Z, Zhang J. FiBiNET: Combining Feature Importance and Bilinear feature Interaction for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1905.09433, 2019.

FLEN(Field-Leveraged Embedding Network)

A large-scale CTR prediction model with efficient usage of field information to alleviate gradient coupling problem.

FLEN Model API

FLEN example



FLEN

Chen W, Zhan L, Ci Y, Lin C. FLEN: Leveraging Field for Scalable CTR Prediction[J]. arXiv preprint arXiv:1911.04690, 2019.

2.2.4 Layers

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

The module is a class that inherits from `tf.keras.layers.Layer`,it has the same attributes and methods as keras Layers like `tf.keras.layers.Dense()` etc

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.

	label	I1	I2	I3	I4	I5	I6	I7	I8	I9	...	C17	C18	C19	C20	C21	C22	C23	C24
0	0	NaN	3	260.0	NaN	17668.0	NaN	NaN	33.0	NaN	...	e5ba7672	87c6f83c	NaN	NaN	0429f84b	NaN	3a171ecb	c0d61a5c
1	0	NaN	-1	19.0	35.0	30251.0	247.0	1.0	35.0	160.0	...	d4bb7bd8	6fc84bfb	NaN	NaN	5155d8a3	NaN	be7c41b4	ded4aac9
2	0	0.0	0	2.0	12.0	2013.0	164.0	6.0	35.0	523.0	...	e5ba7672	675c9258	NaN	NaN	2e01979f	NaN	bcdee96c	6d5d1302
3	0	NaN	13	1.0	4.0	16836.0	200.0	5.0	4.0	29.0	...	e5ba7672	52e44668	NaN	NaN	e587c466	NaN	32c7478e	3b183c5c
4	0	0.0	0	104.0	27.0	1990.0	142.0	4.0	32.0	37.0	...	e5ba7672	25c88e42	21ddcdc9	b1252a9d	0e8585d2	NaN	32c7478e	0d4a6d1a

image

In this example,we simply normailize 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.

```
import pandas as pd
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.models import *
from deepctr.feature_column import SparseFeat, DenseFeat, get_feature_names

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

(continues on next page)

(continued from previous page)

```
    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
    model = DeepFM(linear_feature_columns, dnn_feature_columns, task='binary')
    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)
    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 Classification: Criteo with feature hashing on the fly

This example shows how to use `DeepFM` to solve a simple binary classification task using feature hashing. You can get the demo data criteo_sample.txt and run the following codes.

```
import pandas as pd
from sklearn.metrics import log_loss, roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

from deepctr.models import DeepFM
from deepctr.feature_column import SparseFeat, DenseFeat,get_feature_names

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.do simple Transformation for dense features
    mms = MinMaxScaler(feature_range=(0, 1))
    data[dense_features] = mms.fit_transform(data[dense_features])

    # 2.set hashing space for each sparse field,and record dense feature field name

    fixlen_feature_columns = [SparseFeat(feat, vocabulary_size=1000,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]
```

(continues on next page)

(continued from previous page)

```
    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
    model = DeepFM(linear_feature_columns,dnn_feature_columns, task='binary')
    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)
    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))
```

2.3.3 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.

	movie_id	user_id	gender	age	occupation	zip	rating
254181	2944	1545	M	25	20	20009	4
481546	2208	2962	M	35	3	94109	3
166949	3629	1062	M	50	19	59457	5
536371	569	3308	F	18	20	15701-1348	2
117094	2763	754	M	35	7	38024	4

image

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

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

from deepctr.models import DeepFM
from deepctr.feature_column import SparseFeat,get_feature_names

if __name__ == "__main__":

    data = pd.read_csv("./movielens_sample.txt")
```

(continues on next page)

(continued from previous page)

```
    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(),embedding_dim=4)
                              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].values for name in feature_names}
    test_model_input = {name:test[name].values for name in feature_names}

    # 4.Define Model,train,predict and evaluate
    model = DeepFM(linear_feature_columns, dnn_feature_columns, task='regression')
    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))
```

2.3.4 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.

	movie_id	user_id	gender	age	occupation	zip	genres	rating
0	12	107	0	2	4	35	Comedy|Drama	4
1	169	123	1	1	4	118	Action|Thriller	3
2	6	12	0	2	13	99	Drama|Romance	4
3	112	21	1	1	18	55	Action|Adventure	3
4	45	187	1	5	19	41	Comedy|Drama	5

image

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

1. Generate the paded 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.

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from tensorflow.python.keras.preprocessing.sequence import pad_sequences

from deepctr.models import DeepFM
from deepctr.feature_column import SparseFeat, VarLenSparseFeat,get_feature_names


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]

    use_weighted_sequence = False
    if use_weighted_sequence:
        varlen_feature_columns = [VarLenSparseFeat(SparseFeat('genres',vocabulary_
↪size=len(
            key2index) + 1,embedding_dim=4), maxlen= max_len, combiner='mean',weight_
↪name='genres_weight')]  # Notice : value 0 is for padding for sequence input feature
    else:
        varlen_feature_columns = [VarLenSparseFeat(SparseFeat('genres',vocabulary_
↪size= len(
            key2index) + 1,embedding_dim=4), maxlen=max_len, combiner='mean',weight_
↪name=None)]  # Notice : value 0 is for padding for sequence input feature
```

(continues on next page)

(continued from previous page)

```
    linear_feature_columns = fixlen_feature_columns + varlen_feature_columns
    dnn_feature_columns = fixlen_feature_columns + varlen_feature_columns

    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
    model_input["genres_weight"] =  np.random.randn(data.shape[0],max_len,1)


    # 4.Define Model,compile and train
    model = DeepFM(linear_feature_columns,dnn_feature_columns,task='regression')

    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, )
```

2.3.5 Multi-value Input : Movielens with feature hashing on the fly

```
import numpy as np
import pandas as pd
from tensorflow.python.keras.preprocessing.sequence import pad_sequences

from deepctr.feature_column import SparseFeat, VarLenSparseFeat,get_feature_names
from deepctr.models import DeepFM

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

    data[sparse_features] = data[sparse_features].astype(str)
    target = ['rating']

    # 1.Use hashing encoding on the fly for sparse features,and process sequence␣
↪features

    genres_list = list(map(lambda x: x.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',␣
↪dtype=str, value=0)

    # 2.set hashing space for each sparse field and generate feature config for␣
↪sequence feature

    fixlen_feature_columns = [SparseFeat(feat, data[feat].nunique() * 5, embedding_
↪dim=4, use_hash=True, dtype='string')
```

(continues on next page)

(continued from previous page)

```
                              for feat in sparse_features]
    varlen_feature_columns = [
        VarLenSparseFeat(SparseFeat('genres', vocabulary_size=100, embedding_dim=4,␣
↪use_hash=True, dtype="string"),
                         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
    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 feature_names}
    model_input['genres'] = genres_list

    # 4.Define Model,compile and train
    model = DeepFM(linear_feature_columns, dnn_feature_columns, task='regression')

    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, )
```

2.3.6 Estimator with TFRecord: Classification Criteo

This example shows how to use `DeepFMEstimator` to solve a simple binary classification task. You can get the demo data criteo_sample.tr.tfrecords and criteo_sample.te.tfrecords and run the following codes.

```
import tensorflow as tf

from deepctr.estimator import DeepFMEstimator
from deepctr.estimator.inputs import input_fn_tfrecord

if __name__ == "__main__":

    # 1.generate feature_column for linear part and dnn part

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

    dnn_feature_columns = []
    linear_feature_columns = []

    for i, feat in enumerate(sparse_features):
        dnn_feature_columns.append(tf.feature_column.embedding_column(
            tf.feature_column.categorical_column_with_identity(feat, 1000), 4))
        linear_feature_columns.append(tf.feature_column.categorical_column_with_
↪identity(feat, 1000))
    for feat in dense_features:
        dnn_feature_columns.append(tf.feature_column.numeric_column(feat))
        linear_feature_columns.append(tf.feature_column.numeric_column(feat))

    # 2.generate input data for model

    feature_description = {k: tf.FixedLenFeature(dtype=tf.int64, shape=1) for k in␣
↪sparse_features}
```

(continues on next page)

(continued from previous page)

```
    feature_description.update(
        {k: tf.FixedLenFeature(dtype=tf.float32, shape=1) for k in dense_features})
    feature_description['label'] = tf.FixedLenFeature(dtype=tf.float32, shape=1)

    train_model_input = input_fn_tfrecord('./criteo_sample.tr.tfrecords', feature_
↪description, 'label', batch_size=256,
                                          num_epochs=1, shuffle_factor=10)
    test_model_input = input_fn_tfrecord('./criteo_sample.te.tfrecords', feature_
↪description, 'label',
                                         batch_size=2 ** 14, num_epochs=1, shuffle_
↪factor=0)

    # 3.Define Model,train,predict and evaluate
    model = DeepFMEstimator(linear_feature_columns, dnn_feature_columns, task='binary
↪')

    model.train(train_model_input)
    eval_result = model.evaluate(test_model_input)

    print(eval_result)
```

2.3.7 Estimator with Pandas DataFrame: Classification Criteo

This example shows how to use `DeepFMEstimator` to solve a simple binary classification task. You can get the demo data criteo_sample.txt and run the following codes.

```
import pandas as pd
import tensorflow as tf
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.estimator import DeepFMEstimator
from deepctr.estimator.inputs import input_fn_pandas

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
```

(continues on next page)

(continued from previous page)

```
    dnn_feature_columns = []
    linear_feature_columns = []

    for i, feat in enumerate(sparse_features):
        dnn_feature_columns.append(tf.feature_column.embedding_column(
            tf.feature_column.categorical_column_with_identity(feat, data[feat].
↪nunique()), 4))
        linear_feature_columns.append(tf.feature_column.categorical_column_with_
↪identity(feat, data[feat].nunique()))
    for feat in dense_features:
        dnn_feature_columns.append(tf.feature_column.numeric_column(feat))
        linear_feature_columns.append(tf.feature_column.numeric_column(feat))

    # 3.generate input data for model

    train, test = train_test_split(data, test_size=0.2)

    # Not setting default value for continuous feature. filled with mean.

    train_model_input = input_fn_pandas(train, sparse_features + dense_features,
↪'label', shuffle=True)
    test_model_input = input_fn_pandas(test, sparse_features + dense_features, None,␣
↪shuffle=False)

    # 4.Define Model,train,predict and evaluate
    model = DeepFMEstimator(linear_feature_columns, dnn_feature_columns, task='binary
↪')

    model.train(train_model_input)
    pred_ans_iter = model.predict(test_model_input)
    pred_ans = list(map(lambda x: x['pred'], pred_ans_iter))
    #
    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))
```

2.4 FAQ

2.4.1 1. Save or load weights/models

To save/load weights,you can write codes just like any other keras models.

```
model = DeepFM()
model.save_weights('DeepFM_w.h5')
model.load_weights('DeepFM_w.h5')
```

To save/load models,just a little different.

```
from tensorflow.python.keras.models import  save_model,load_model
model = DeepFM()
save_model(model, 'DeepFM.h5')# save_model, same as before
```

(continues on next page)

(continued from previous page)

```
from deepctr.layers import custom_objects
model = load_model('DeepFM.h5',custom_objects)# load_model,just add a parameter
```

2.4.2 2. Set learning rate and use earlystopping

You can use any models in DeepCTR like a keras model object. Here is a example of how to set learning rate and earlystopping:

```
import deepctr
from tensorflow.python.keras.optimizers import Adam,Adagrad
from tensorflow.python.keras.callbacks import EarlyStopping

model = deepctr.models.DeepFM(linear_feature_columns,dnn_feature_columns)
model.compile(Adagrad(0.1024),'binary_crossentropy',metrics=['binary_crossentropy'])

es = EarlyStopping(monitor='val_binary_crossentropy')
history = model.fit(model_input, data[target].values,batch_size=256, epochs=10,␣
↪verbose=2, validation_split=0.2,callbacks=[es] )
```

2.4.3 3. Get the attentional weights of feature interactions in AFM

First,make sure that you have install the latest version of deepctr.

Then,use the following code,the `attentional_weights[:,i,0]` is the `feature_interactions[i]`'s attentional weight of all samples.

```
import itertools
import deepctr
from deepctr.models import AFM
from deepctr.feature_column import get_feature_names
from tensorflow.python.keras.models import Model
from tensorflow.python.keras.layers import Lambda

model = AFM(linear_feature_columns,dnn_feature_columns)
model.fit(model_input,target)

afmlayer = model.layers[-3]
afm_weight_model = Model(model.input,outputs=Lambda(lambda x:afmlayer.normalized_att_
↪score)(model.input))
attentional_weights = afm_weight_model.predict(model_input,batch_size=4096)

feature_names = get_feature_names(dnn_feature_columns)
feature_interactions = list(itertools.combinations(feature_names ,2))
```

2.4.4 4. How to extract the embedding vectors in deepfm?

```
feature_columns = [SparseFeat('user_id',120,),SparseFeat('item_id',60,),SparseFeat(
↪'cate_id',60,)]

def get_embedding_weights(dnn_feature_columns,model):
    embedding_dict = {}
    for fc in dnn_feature_columns:
        if hasattr(fc,'embedding_name'):
            if fc.embedding_name is not None:
                name = fc.embedding_name
            else:
                name = fc.name
            embedding_dict[name] = model.get_layer("sparse_emb_"+name).get_
↪weights()[0]
    return embedding_dict

embedding_dict = get_embedding_weights(feature_columns,model)

user_id_emb = embedding_dict['user_id']
item_id_emb = embedding_dict['item_id']
```

2.4.5 5. How to add a long dense feature vector as a input to the model?

```
from deepctr.models import DeepFM
from deepctr.feature_column import SparseFeat, DenseFeat,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)
```

2.4.6 6. How to use pretrained weights to initialize embedding weights and frozen embedding weights?

Use `tf.initializers.identity()` to set the `embeddings_initializer` of `SparseFeat`,and set `trainable=False` to frozen embedding weights.

```
import numpy as np
import tensorflow as tf
from deepctr.models import DeepFM
```

(continues on next page)

(continued from previous page)

```
from deepctr.feature_column import SparseFeat,get_feature_names

pretrained_item_weights = np.random.randn(60,4)
pretrained_weights_initializer = tf.initializers.identity(pretrained_item_weights)

feature_columns = [SparseFeat('user_id',120,),SparseFeat('item_id',60,embedding_dim=4,
↪embeddings_initializer=pretrained_weights_initializer,trainable=False)]
fixlen_feature_names = get_feature_names(feature_columns)

user_id = np.array([[1],[0],[1]])
item_id = np.array([[30],[20],[10]])
label = np.array([1,0,1])

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

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

2.4.7 7. How to run the demo with GPU ?

just install deepctr with

```
$ pip install deepctr[gpu]
```

2.4.8 8. How to run the demo with multiple GPUs

you can use multiple gpus with tensorflow version higher than `1.4`,see run_classification_criteo_multi_gpu.py

2.5 History

- 06/27/2020 : v0.8.0 released.
  - Support `Tensorflow Estimator` for large scale data and distributed training. example: Estimator with TFRecord
  - Support different initializers for different embedding weights and loading pretrained embeddings. example
  - Add new model `FwFM`.
- 05/17/2020 : v0.7.5 released.Fix numerical instability in `LayerNormalization`.
- 03/15/2020 : v0.7.4 released.Add FLEN and `FieldWiseBiInteraction`.
- 03/04/2020 : v0.7.3 released.Fix the inconsistency of prediction results when the model is loaded with trained weights.
- 02/08/2020 : v0.7.2 released.Fix some bugs.
- 01/28/2020 : v0.7.1 released.Simplify VarLenSparseFeat,support setting weight_normalization.Fix problem of embedding size of `SparseFeat` in `linear_feature_columns`.
- 11/24/2019 : v0.7.0 released.Refactor feature columns.Different features can use different `embedding_dim` and group-wise interaction is available by setting `group_name`.

- 11/06/2019 : v0.6.3 released.Add `WeightedSequenceLayer` and support weighted sequence feature input.
- 10/03/2019 : v0.6.2 released.Simplify the input logic.
- 09/08/2019 : v0.6.1 released.Fix bugs in `CCPM` and `DynamicGRU`.
- 08/02/2019 : v0.6.0 released.Now DeepCTR is compatible with tensorflow `1.14` and `2.0.0`.
- 07/21/2019 : v0.5.2 released.Refactor `Linear` Layer.
- 07/10/2019 : v0.5.1 released.Add FiBiNET.
- 06/30/2019 : v0.5.0 released.Refactor inputs module.
- 05/19/2019 : v0.4.1 released.Add DSIN.
- 05/04/2019 : v0.4.0 released.Support feature hashing on the fly and python2.7.
- 04/27/2019 : v0.3.4 released.Add FGCNN and `FGCNNLayer`.
- 04/21/2019 : v0.3.3 released.Add CCPM.
- 03/30/2019 : v0.3.2 released.Add DIEN and ONN Model.
- 02/17/2019 : v0.3.1 released.Refactor layers ,add `BiLSTM` and `Transformer`.
- 01/24/2019 : v0.2.3 released.Use a new feature config generation method and fix bugs.
- 01/01/2019 : v0.2.2 released.Add sequence(multi-value) input support for `AFM`,`AutoInt`,`DCN`,`DeepFM`, `FNN`,`NFM`,`PNN`,`xDeepFM` models.
- 12/27/2018 : v0.2.1 released.Add AutoInt Model.
- 12/22/2018 : v0.2.0 released.Add xDeepFM and automatic check for new version.
- 12/19/2018 : v0.1.6 released.Now DeepCTR is compatible with tensorflow from `1.4-1.12` except for `1.7` and `1.8`.
- 11/29/2018 : v0.1.4 released.Add FAQ in docs
- 11/24/2018 : DeepCTR first version v0.1.0 is released on PyPi

2.6 DeepCTR Models API

2.6.1 Methods

compile

```
compile(optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_
↪mode=None, weighted_metrics=None, target_tensors=None)
```

Configures the model for training.

Arguments

- **optimizer**: String (name of optimizer) or optimizer instance. See optimizers.
- **loss**: String (name of objective function) or objective function. See losses. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses.

- **metrics**: List of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a': 'accuracy'}`.
- **loss_weights**: Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the weighted sum of all individual losses, weighted by the `loss_weights` coefficients. If a list, it is expected to have a 1:1 mapping to the model's outputs. If a tensor, it is expected to map output names (strings) to scalar coefficients.
- **sample_weight_mode**: If you need to do timestep-wise sample weighting (2D weights), set this to `"temporal"`. `None` defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a dictionary or a list of modes.
- **weighted_metrics**: List of metrics to be evaluated and weighted by sample_weight or class_weight during training and testing.
- **target_tensors**: By default, Keras will create placeholders for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensors (in turn, Keras will not expect external Numpy data for these targets at training time), you can specify them via the `target_tensors` argument. It can be a single tensor (for a single-output model), a list of tensors, or a dict mapping output names to target tensors.

Raises

- **ValueError**: In case of invalid arguments for `optimizer`, `loss`, `metrics` or `sample_weight_mode`.

fit

```
fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_
↪split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_
↪weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None,␣
↪validation_freq=1)
```

Trains the model for a given number of epochs (iterations on a dataset).

Arguments

- **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. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **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). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.
- **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.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during training and validation (if). See callbacks.

- **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 training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **sample_weight**: Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).
- **steps_per_epoch**: Integer or `None`. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default `None` is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined. validation_steps: Only relevant if `steps_per_epoch` is specified. Total number of steps (batches of samples) to validate before stopping.
- **validation_freq**: Only relevant if validation data is provided. Integer or list/tuple/set. If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a list, tuple, or set, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.

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).

Raises

- **RuntimeError**: If the model was never compiled. ValueError: In case of mismatch between the provided input data and what the model expects.

evaluate

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None,␣
↪callbacks=None)
```

Returns the loss value & metrics values for the model in test mode. Computation is done in batches.

Arguments

- **x**: Numpy array of test 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. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).

- **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). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per evaluation step. If unspecified, `batch_size` will default to 32.
- **verbose**: 0 or 1. Verbosity mode. 0 = silent, 1 = progress bar.
- **sample_weight**: Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **steps**: Integer or `None`. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `None`.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during evaluation. See callbacks.

Returns

- Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

predict

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None)
```

Generates output predictions for the input samples.

Computation is done in batches.

Arguments

- **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 32.
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of None.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during prediction. See callbacks.

Returns

- Numpy array(s) of predictions.

Raises

- **ValueError**: In case of mismatch between the provided input data and the model's expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.

train_on_batch

```
train_on_batch(x, y, sample_weight=None, class_weight=None)
```

Runs a single gradient update on a single batch of data.

Arguments

- **x**: Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

Returns

- Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

test_on_batch

```
test_on_batch(x, y, sample_weight=None)
```

Test the model on a single batch of samples.

Arguments

- **x**: Numpy array of test data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.

Returns

- Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

predict_on_batch

```
predict_on_batch(x)
```

Returns predictions for a single batch of samples.

Arguments

- **x**: Input samples, as a Numpy array.

Returns

- Numpy array(s) of predictions.

fit_generator

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None,␣
↪validation_data=None, validation_steps=None, validation_freq=1, class_weight=None,␣
↪max_queue_size=10, workers=1, use_multiprocessing=False, shuffle=True, initial_
↪epoch=0)
```

Trains the model on data generated batch-by-batch by a Python generator (or an instance of `Sequence`). The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU. The use of `tf.keras.utils.Sequence` guarantees the ordering and guarantees the single use of every input per epoch when using `use_multiprocessing=True`.

Arguments

- **generator**: A generator or an instance of `Sequence` (`tf.keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing. The output of the generator must be either

  a tuple `(inputs, targets)` or a tuple `(inputs, targets, sample_weights)`.This tuple (a single output of the generator) makes a single batch. Therefore, all arrays in this tuple must have the same length (equal to the size of this batch). Different batches may have different sizes. For example, the last batch of the epoch is commonly smaller than the others, if the size of the dataset is not divisible by the batch size. The generator is expected to loop over its data indefinitely. An epoch finishes when `steps_per_epoch` batches have been seen by the model.

- **steps_per_epoch**: Integer. Total number of steps (batches of samples) to yield from `generator` before declaring one epoch finished and starting the next epoch. It should typically be equal to `ceil(num_samples / batch_size)` Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.

- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire data provided, as defined by `steps_per_epoch`. 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.

- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.

- **validation_data**: This can be either a generator or a `Sequence` object for the validation data tuple `(x_val, y_val)` 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_steps**: Only relevant if `validation_data` is a generator. Total number of steps (batches of samples) to yield from `validation_data` generator before stopping at the end of every epoch. It should typically be equal to the number of samples of your validation dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(validation_data)` as a number of steps.

- **validation_freq**: Only relevant if validation data is provided. Integer or `collections.Container` instance (e.g. list, tuple, etc.). If an integer, specifies how many training epochs to run before a new validation

run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a Container, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.

- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **max_queue_size**: Integer. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.
- **shuffle**: Boolean. Whether to shuffle the order of the batches at the beginning of each epoch. Only used with instances of `Sequence` (`tf.keras.utils.Sequence`). Has no effect when `steps_per_epoch` is not `None`. initial_epoch: Integer. Epoch at which to start training (useful for resuming a previous training run).

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).

Raises

- **ValueError**: In case the generator yields data in an invalid format.

Example

```
def generate_arrays_from_file(path):
    while True:
        with open(path) as f:
            for line in f:
                # create numpy arrays of input data
                # and labels, from each line in the file
                x1, x2, y = process_line(line)
                yield ({'input_1': x1, 'input_2': x2}, {'output': y})

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
                    steps_per_epoch=10000, epochs=10)
```

evaluate_generator

```
evaluate_generator(generator, steps=None, callbacks=None, max_queue_size=10,␣
↪workers=1, use_multiprocessing=False, verbose=0)
```

Evaluates the model on a data generator. The generator should return the same kind of data as accepted by `test_on_batch`.

Arguments

- **generator**: Generator yielding tuples (inputs, targets) or (inputs, targets, sample_weights) or an instance of Sequence (tf.keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.

- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **max_queue_size**: maximum size for the generator queue
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

- Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

predict_generator

```
predict_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1,
↪ use_multiprocessing=False, verbose=0)
```

Generates predictions for the input samples from a data generator. The generator should return the same kind of data as accepted by `predict_on_batch`.

Arguments

- **generator**: Generator yielding batches of input samples or an instance of Sequence (`tf.keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **max_queue_size**: Maximum size for the generator queue.
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: If `True`, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

- Numpy array(s) of predictions.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

get_layer

```
get_layer(name=None, index=None)
```

Retrieves a layer based on either its name (unique) or index. If `name` and `index` are both provided, `index` will take precedence. Indices are based on order of horizontal graph traversal (bottom-up).

Arguments

- **name**: String, name of layer.
- **index**: Integer, index of layer.

Returns

- A layer instance.

Raises

- **ValueError**: In case of invalid layer name or index.

2.6.2 deepctr.models.ccpm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Liu Q, Yu F, Wu S, et al. A convolutional click prediction model[C]//Proceedings of the 24th ACM International on Conference on Information and Knowledge Management. ACM, 2015: 1743-1746. (http://ir.ia.ac.cn/bitstream/173211/12337/1/A%20Convolutional%20Click%20Prediction%20Model.pdf)

deepctr.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*, *seed=1024*, *task='binary'*)

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

Returns A Keras model instance.

2.6.3 deepctr.models.fnn module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Zhang W, Du T, Wang J. Deep learning over multi-field categorical data[C]//European conference on information retrieval. Springer, Cham, 2016: 45-57.(https://arxiv.org/pdf/1601.02376.pdf)

`deepctr.models.fnn.`**`FNN`**(*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*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *task='binary'*)

Instantiates the Factorization-supported Neural 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 weight
- **`l2_reg_dnn`** – float . L2 regularizer strength applied to DNN
- **`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

Returns A Keras model instance.

2.6.4 deepctr.models.pnn module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Qu Y, Cai H, Ren K, et al. Product-based neural networks for user response prediction[C]//Data Mining (ICDM), 2016 IEEE 16th International Conference on. IEEE, 2016: 1149-1154.(https://arxiv.org/pdf/1611.00144.pdf)

`deepctr.models.pnn.`**`PNN`**(*dnn_feature_columns*, *dnn_hidden_units=(128, 128)*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *use_inner=True*, *use_outter=False*, *kernel_type='mat'*, *task='binary'*)

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
- **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

Returns A Keras model instance.

2.6.5 deepctr.models.wdl module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Cheng H T, Koc L, Harmsen J, et al. Wide & deep learning for recommender systems[C]//Proceedings of the 1st Workshop on Deep Learning for Recommender Systems. ACM, 2016: 7-10.(https://arxiv.org/pdf/1606.07792.pdf)

`deepctr.models.wdl.`**WDL**(*linear_feature_columns*, *dnn_feature_columns*, *dnn_hidden_units=(128, 128)*, *l2_reg_linear=1e-05*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *task='binary'*)

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
- **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

Returns A Keras model instance.

2.6.6 deepctr.models.deepfm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Guo H, Tang R, Ye Y, et al. Deepfm: a factorization-machine based neural network for ctr prediction[J]. arXiv preprint arXiv:1703.04247, 2017.(https://arxiv.org/abs/1703.04247)

deepctr.models.deepfm.**DeepFM**(*linear_feature_columns, dnn_feature_columns, fm_group=['default_group'], dnn_hidden_units=(128, 128), l2_reg_linear=1e-05, l2_reg_embedding=1e-05, l2_reg_dnn=0, seed=1024, dnn_dropout=0, dnn_activation='relu', dnn_use_bn=False, task='binary'*)

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.
- **fm_group** – list, group_name of features that will be used to do feature interactions.
- **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
- **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

Returns A Keras model instance.

2.6.7 deepctr.models.mlr module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Gai K, Zhu X, Li H, et al. Learning Piece-wise Linear Models from Large Scale Data for Ad Click Prediction[J]. arXiv preprint arXiv:1704.05194, 2017.(https://arxiv.org/abs/1704.05194)

deepctr.models.mlr.**MLR**(*region_feature_columns, base_feature_columns=None, region_num=4, l2_reg_linear=1e-05, seed=1024, task='binary', bias_feature_columns=None*)

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
- **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.

Returns A Keras model instance.

2.6.8 deepctr.models.nfm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] He X, Chua T S. Neural factorization machines for sparse predictive analytics[C]//Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval. ACM, 2017: 355-364. (https://arxiv.org/abs/1708.05027)

deepctr.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*, *seed=1024*, *bi_dropout=0*, *dnn_dropout=0*, *dnn_activation='relu'*, *task='binary'*)

Instantiates the Neural 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.
- **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
- **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

Returns A Keras model instance.

2.6.9 deepctr.models.afm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Xiao J, Ye H, He X, et al. Attentional factorization machines: Learning the weight of feature interactions via attention networks[J]. arXiv preprint arXiv:1708.04617, 2017. (https://arxiv.org/abs/1708.04617)

deepctr.models.afm.**AFM**(*linear_feature_columns*, *dnn_feature_columns*, *fm_group='default_group'*, *use_attention=True*, *attention_factor=8*, *l2_reg_linear=1e-05*, *l2_reg_embedding=1e-05*, *l2_reg_att=1e-05*, *afm_dropout=0*, *seed=1024*, *task='binary'*)

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.
- **fm_group** – list, group_name of features that will be used to do feature interactions.
- **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.
- **seed** – integer ,to use as random seed.
- **task** – str, `"binary"` for binary logloss or `"regression"` for regression loss

Returns A Keras model instance.

2.6.10 deepctr.models.dcn module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Wang R, Fu B, Fu G, et al. Deep & cross network for ad click predictions[C]//Proceedings of the ADKDD'17. ACM, 2017: 12. (https://arxiv.org/abs/1708.05123)

deepctr.models.dcn.**DCN**(*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*, *seed=1024*, *dnn_dropout=0*, *dnn_use_bn=False*, *dnn_activation='relu'*, *task='binary'*)

Instantiates the Deep&Cross 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.
- **cross_num** – positive integet,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
- **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

Returns A Keras model instance.

2.6.11 deepctr.models.din module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Zhou G, Zhu X, Song C, et al. Deep interest network for click-through rate prediction[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. ACM, 2018: 1059-1068. (https://arxiv.org/pdf/1706.06978.pdf)

deepctr.models.din.**DIN**(*dnn_feature_columns*, *history_feature_list*, *dnn_use_bn=False*, *dnn_hidden_units=(200, 80)*, *dnn_activation='relu'*, *att_hidden_size=(80, 40)*, *att_activation='dice'*, *att_weight_normalization=False*, *l2_reg_dnn=0*, *l2_reg_embedding=1e-06*, *dnn_dropout=0*, *seed=1024*, *task='binary'*)

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.
- **seed** – integer ,to use as random seed.
- **task** – str, "binary" for binary logloss or "regression" for regression loss

Returns A Keras model instance.

2.6.12 deepctr.models.dien module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Zhou G, Mou N, Fan Y, et al. Deep Interest Evolution Network for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1809.03672, 2018. (https://arxiv.org/pdf/1809.03672.pdf)

`deepctr.models.dien.`**`DIEN`**(*dnn_feature_columns*, *history_feature_list*, *gru_type='GRU'*, *use_negsampling=False*, *alpha=1.0*, *use_bn=False*, *dnn_hidden_units=(200, 80)*, *dnn_activation='relu'*, *att_hidden_units=(64, 16)*, *att_activation='dice'*, *att_weight_normalization=True*, *l2_reg_dnn=0*, *l2_reg_embedding=1e-06*, *dnn_dropout=0*, *seed=1024*, *task='binary'*)

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 negtive 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

Returns A Keras model instance.

2.6.13 deepctr.models.dsin module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Feng Y, Lv F, Shen W, et al. Deep Session Interest Network for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1905.06482, 2019.(https://arxiv.org/abs/1905.06482)

deepctr.models.dsin.**DSIN**(*dnn_feature_columns*, *sess_feature_list*, *sess_max_count=5*, *bias_encoding=False*, *att_embedding_size=1*, *att_head_num=8*, *dnn_hidden_units=(200, 80)*, *dnn_activation='sigmoid'*, *dnn_dropout=0*, *dnn_use_bn=False*, *l2_reg_dnn=0*, *l2_reg_embedding=1e-06*, *seed=1024*, *task='binary'*)

Instantiates the Deep Session Interest Network architecture.

Parameters

- **dnn_feature_columns** – An iterable containing all the features used by deep part of the model.
- **sess_feature_list** – list,to indicate sequence sparse field
- **sess_max_count** – positive int, to indicate the max number of sessions
- **sess_len_max** – positive int, to indicate the max length of each session
- **bias_encoding** – bool. Whether use bias encoding or postional encoding
- **att_embedding_size** – positive int, the embedding size of each attention head
- **att_head_num** – positive int, the number of attention head
- **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
- **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 in deep net
- **l2_reg_dnn** – float. L2 regularizer strength applied to DNN
- **l2_reg_embedding** – float. L2 regularizer strength applied to embedding vector
- **seed** – integer ,to use as random seed.
- **task** – str, "binary" for binary logloss or "regression" for regression loss

Returns A Keras model instance.

2.6.14 deepctr.models.xdeepfm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Lian J, Zhou X, Zhang F, et al. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems[J]. arXiv preprint arXiv:1803.05170, 2018.(https://arxiv.org/pdf/1803.05170.pdf)

deepctr.models.xdeepfm.**xDeepFM**(*linear_feature_columns*, *dnn_feature_columns*, *dnn_hidden_units=(256, 256)*, *cin_layer_size=(128, 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*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *dnn_use_bn=False*, *task='binary'*)

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.
- **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

Returns A Keras model instance.

2.6.15 deepctr.models.autoint module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Song W, Shi C, Xiao Z, et al. AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks[J]. arXiv preprint arXiv:1810.11921, 2018.(https://arxiv.org/abs/1810.11921)

`deepctr.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, 256)*, *dnn_activation='relu'*, *l2_reg_linear=1e-05*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *dnn_use_bn=False*, *dnn_dropout=0*, *seed=1024*, *task='binary'*)

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_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_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.
- **seed** – integer ,to use as random seed.
- **task** – str, "binary" for binary logloss or "regression" for regression loss

Returns A Keras model instance.

2.6.16 deepctr.models.onn module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Yang Y, Xu B, Shen F, et al. Operation-aware Neural Networks for User Response Prediction[J]. arXiv preprint arXiv:1904.12579, 2019. https://arxiv.org/pdf/1904.12579

deepctr.models.onn.**ONN**(*linear_feature_columns*, *dnn_feature_columns*, *embedding_size=4*, *dnn_hidden_units=(128, 128)*, *l2_reg_embedding=1e-05*, *l2_reg_linear=1e-05*, *l2_reg_dnn=0*, *dnn_dropout=0*, *seed=1024*, *use_bn=True*, *reduce_sum=False*, *task='binary'*)

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.
- **embedding_size** – positive integer,sparse feature embedding_size
- **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
- **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

Returns A Keras model instance.

2.6.17 deepctr.models.fgcnn module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Liu B, Tang R, Chen Y, et al. Feature Generation by Convolutional Neural Network for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1904.04447, 2019. (https://arxiv.org/pdf/1904.04447)

deepctr.models.fgcnn.**FGCNN**(*linear_feature_columns*, *dnn_feature_columns*, *conv_kernel_width=(7, 7, 7, 7)*, *conv_filters=(14, 16, 18, 20)*, *new_maps=(3, 3, 3, 3)*, *pooling_width=(2, 2, 2, 2)*, *dnn_hidden_units=(128,)*, *l2_reg_linear=1e-05*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *dnn_dropout=0*, *seed=1024*, *task='binary'*)

Instantiates the Feature Generation by Convolutional Neural 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.
- **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.
- **new_maps** – list, list of positive integer or empty list, the feature maps of generated features.
- **pooling_width** – list, list of positive integer or empty list,the width of pooling layer.
- **dnn_hidden_units** – list,list of positive integer or empty list, the layer number and units in each layer of deep 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_dnn** – float. L2 regularizer strength applied to DNN
- **dnn_dropout** – float in [0,1), the probability we will drop out a given DNN coordinate.
- **seed** – integer ,to use as random seed.
- **task** – str, "binary" for binary logloss or "regression" for regression loss

Returns A Keras model instance.

2.6.18 deepctr.models.fibinet module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Huang T, Zhang Z, Zhang J. FiBiNET: Combining Feature Importance and Bilinear feature Interaction for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1905.09433, 2019.

deepctr.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*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *task='binary'*)

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,inf), 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
- **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

Returns A Keras model instance.

2.6.19 deepctr.models.flen module

Author: Tingyi Tan,5636374@qq.com

Reference: [1] Chen W, Zhan L, Ci Y, Lin C. FLEN: Leveraging Field for Scalable CTR Prediction . arXiv preprint arXiv:1911.04690, 2019.(https://arxiv.org/pdf/1911.04690)

deepctr.models.flen.**FLEN**(*linear_feature_columns*, *dnn_feature_columns*, *dnn_hidden_units=(128, 128)*, *l2_reg_linear=1e-05*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *seed=1024*, *dnn_dropout=0.0*, *dnn_activation='relu'*, *dnn_use_bn=False*, *task='binary'*)

Instantiates the FLEN 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_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
- **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

Returns A Keras model instance.

2.7 DeepCTR Estimators API

2.7.1 deepctr.estimator.models.ccpm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Liu Q, Yu F, Wu S, et al. A convolutional click prediction model[C]//Proceedings of the 24th ACM International on Conference on Information and Knowledge Management. ACM, 2015: 1743-1746. (http://ir.ia.ac.cn/bitstream/173211/12337/1/A%20Convolutional%20Click%20Prediction%20Model.pdf)

`deepctr.estimator.models.ccpm.`**`CCPMEstimator`**(*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*, *seed=1024*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

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
- **task** – str, "binary" for binary logloss or "regression" for regression loss
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.2 deepctr.estimator.models.fnn module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Zhang W, Du T, Wang J. Deep learning over multi-field categorical data[C]//European conference on information retrieval. Springer, Cham, 2016: 45-57.(https://arxiv.org/pdf/1601.02376.pdf)

deepctr.estimator.models.fnn.**FNNEstimator**(*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*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

Instantiates the Factorization-supported Neural 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 weight
- **l2_reg_dnn** – float . L2 regularizer strength applied to DNN
- **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
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.3 deepctr.estimator.models.pnn module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Qu Y, Cai H, Ren K, et al. Product-based neural networks for user response prediction[C]//Data Mining (ICDM), 2016 IEEE 16th International Conference on. IEEE, 2016: 1149-1154.(https://arxiv.org/pdf/1611.00144.pdf)

deepctr.estimator.models.pnn.**PNNEstimator**(*dnn_feature_columns*, *dnn_hidden_units=(128, 128)*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *use_inner=True*, *use_outter=False*, *kernel_type='mat'*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

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
- **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

- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.4 deepctr.estimator.models.wdl module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Cheng H T, Koc L, Harmsen J, et al. Wide & deep learning for recommender systems[C]//Proceedings of the 1st Workshop on Deep Learning for Recommender Systems. ACM, 2016: 7-10.(https://arxiv.org/pdf/1606.07792.pdf)

deepctr.estimator.models.wdl.**WDLEstimator**(*linear_feature_columns*, *dnn_feature_columns*, *dnn_hidden_units=(128, 128)*, *l2_reg_linear=1e-05*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

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
- **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

- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.5 deepctr.estimator.models.deepfm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Guo H, Tang R, Ye Y, et al. Deepfm: a factorization-machine based neural network for ctr prediction[J]. arXiv preprint arXiv:1703.04247, 2017.(https://arxiv.org/abs/1703.04247)

deepctr.estimator.models.deepfm.**DeepFMEstimator**(*linear_feature_columns*, *dnn_feature_columns*, *dnn_hidden_units=(128, 128)*, *l2_reg_linear=1e-05*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *dnn_use_bn=False*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

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.
- **fm_group** – list, group_name of features that will be used to do feature interactions.
- **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
- **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
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.6 deepctr.estimator.models.nfm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] He X, Chua T S. Neural factorization machines for sparse predictive analytics[C]//Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval. ACM, 2017: 355-364. (https://arxiv.org/abs/1708.05027)

deepctr.estimator.models.nfm.**NFMEstimator**(*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*, *seed=1024*, *bi_dropout=0*, *dnn_dropout=0*, *dnn_activation='relu'*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

Instantiates the Neural 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.
- **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
- **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
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.7 deepctr.estimator.models.afm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Xiao J, Ye H, He X, et al. Attentional factorization machines: Learning the weight of feature interactions via attention networks[J]. arXiv preprint arXiv:1708.04617, 2017. (https://arxiv.org/abs/1708.04617)

deepctr.estimator.models.afm.**AFMEstimator**(*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*, *seed=1024*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

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.
- **seed** – integer ,to use as random seed.
- **task** – str, "binary" for binary logloss or "regression" for regression loss

- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.8 deepctr.estimator.models.dcn module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Wang R, Fu B, Fu G, et al. Deep & cross network for ad click predictions[C]//Proceedings of the ADKDD'17. ACM, 2017: 12. (https://arxiv.org/abs/1708.05123)

deepctr.estimator.models.dcn.**DCNEstimator**(*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*, *seed=1024*, *dnn_dropout=0*, *dnn_use_bn=False*, *dnn_activation='relu'*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

Instantiates the Deep&Cross 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.
- **cross_num** – positive integet,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
- **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
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.9 deepctr.estimator.models.xdeepfm module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Lian J, Zhou X, Zhang F, et al. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems[J]. arXiv preprint arXiv:1803.05170, 2018.(https://arxiv.org/pdf/1803.05170.pdf)

deepctr.estimator.models.xdeepfm.**xDeepFMEstimator**(*linear_feature_columns*, *dnn_feature_columns*, *dnn_hidden_units=(256, 256)*, *cin_layer_size=(128, 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*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *dnn_use_bn=False*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

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.
- **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
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.10 deepctr.estimator.models.autoint module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Song W, Shi C, Xiao Z, et al. AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks[J]. arXiv preprint arXiv:1810.11921, 2018.(https://arxiv.org/abs/1810.11921)

`deepctr.estimator.models.autoint.`**`AutoIntEstimator`**(*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, 256)*, *dnn_activation='relu'*, *l2_reg_linear=1e-05*, *l2_reg_embedding=1e-05*, *l2_reg_dnn=0*, *dnn_use_bn=False*, *dnn_dropout=0*, *seed=1024*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

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_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_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.
- **seed** – integer ,to use as random seed.
- **task** – str, `"binary"` for binary logloss or `"regression"` for regression loss
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.7.11 deepctr.estimator.models.fibinet module

Author: Weichen Shen,wcshen1994@163.com

Reference: [1] Huang T, Zhang Z, Zhang J. FiBiNET: Combining Feature Importance and Bilinear feature Interaction for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1905.09433, 2019.

deepctr.estimator.models.fibinet.**FiBiNETEstimator**(*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*, *seed=1024*, *dnn_dropout=0*, *dnn_activation='relu'*, *task='binary'*, *model_dir=None*, *config=None*, *linear_optimizer='Ftrl'*, *dnn_optimizer='Adagrad'*, *training_chief_hooks=None*)

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,inf), 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
- **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
- **model_dir** – Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
- **config** – tf.RunConfig object to configure the runtime settings.
- **linear_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the linear part of the model. Defaults to FTRL optimizer.
- **dnn_optimizer** – An instance of *tf.Optimizer* used to apply gradients to the deep part of the model. Defaults to Adagrad optimizer.
- **training_chief_hooks** – Iterable of *tf.train.SessionRunHook* objects to run on the chief worker during training.

Returns A Tensorflow Estimator instance.

2.8 DeepCTR Layers API

2.8.1 deepctr.layers.core module

Author: Weichen Shen,wcshen1994@163.com

class deepctr.layers.core.**DNN**(*hidden_units*, *activation='relu'*, *l2_reg=0*, *dropout_rate=0*, *use_bn=False*, *seed=1024*, ***kwargs*)

The Multi Layer Percetron

Input shape

- nD tensor with shape: `(batch_size, ..., input_dim)`. The most common situation would be a 2D input with shape `(batch_size, input_dim)`.

Output shape

- nD tensor with shape: `(batch_size, ..., hidden_size[-1])`. For instance, for a 2D input with shape `(batch_size, input_dim)`, the output would have shape `(batch_size, hidden_size[-1])`.

Arguments

- **hidden_units**:list of positive integer, the layer number and units in each layer.
- **activation**: Activation function to use.
- **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.

build(*input_shape*)

Creates the variables of the layer.

call(*inputs*, *training=None*, ***kwargs*)

This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.core.**LocalActivationUnit**(*hidden_units=(64, 32), activation='sigmoid', l2_reg=0, dropout_rate=0, use_bn=False, seed=1024, **kwargs*)

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

- [Zhou G, Zhu X, Song C, et al. Deep interest network for click-through rate prediction[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. ACM, 2018: 1059-1068.](https://arxiv.org/pdf/1706.06978.pdf)

build(*input_shape*)

Creates the variables of the layer.

call(*inputs*, *training=None*, ***kwargs*)

This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_mask(*inputs*, *mask*)

Computes an output mask tensor.

Arguments: inputs: Tensor or list of tensors. mask: Tensor or list of tensors.

Returns:

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.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.

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

2.8.2 deepctr.layers.interaction module

Authors: Weichen Shen,wcshen1994@163.com, Harshit Pande

class deepctr.layers.interaction.**AFMLayer**(*attention_factor=4*, *l2_reg_w=0*, *dropout_rate=0*, *seed=1024*, ***kwargs*)
Attentonal 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

- **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)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, *training=None*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**BiInteractionPooling**(***kwargs*)
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

- [He X, Chua T S. Neural factorization machines for sparse predictive analytics[C]//Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval. ACM, 2017: 355-364.](http://arxiv.org/abs/1708.05027)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

class deepctr.layers.interaction.**BilinearInteraction**(*bilinear_type='interaction'*, *seed=1024*, ***kwargs*)
BilinearInteraction Layer used in FiBiNET.

Input shape

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

Output shape

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

Arguments

- **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](https://arxiv.org/pdf/1905.09433.pdf)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**CIN**(*layer_size=(128, 128), activation='relu', split_half=True, l2_reg=1e-05, seed=1024, **kwargs*)
Compressed Interaction Network used in xDeepFM.This implemention is adapted from code that the author of the paper published on https://github.com/Leavingseason/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

- **layer_size** : list of int.Feature maps in each layer.
- **activation** : activation function 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

- [Lian J, Zhou X, Zhang F, et al. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems[J]. arXiv preprint arXiv:1803.05170, 2018.] (https://arxiv.org/pdf/1803.05170.pdf)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**CrossNet**(*layer_num=2*, *l2_reg=0*, *seed=1024*, ***kwargs*)
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

- **layer_num**: Positive integer, the cross layer number
- **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

- [Wang R, Fu B, Fu G, et al. Deep & cross network for ad click predictions[C]//Proceedings of the ADKDD'17. ACM, 2017: 12.](https://arxiv.org/abs/1708.05123)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**FGCNNLayer**(*filters=(14, 16), kernel_width=(7, 7), new_maps=(3, 3), pooling_width=(2, 2), **kwargs*)
Feature Generation Layer used in FGCNN,including Convolution,MaxPooling and Recombination.

Input shape

- A 3D tensor with shape: `(batch_size,field_size,embedding_size)`.

Output shape

- 3D tensor with shape: `(batch_size,new_feture_num,embedding_size)`.

References

- [Liu B, Tang R, Chen Y, et al. Feature Generation by Convolutional Neural Network for Click-Through Rate Prediction[J]. arXiv preprint arXiv:1904.04447, 2019.](https://arxiv.org/pdf/1904.04447)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**FM**(***kwargs*)
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)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

class deepctr.layers.interaction.**FieldWiseBiInteraction**(*use_bias=True*, *seed=1024*, ***kwargs*)
Field-Wise Bi-Interaction Layer used in FLEN,compress the pairwise element-wise product of features into one single vector.

Input shape

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

Output shape

- 2D tensor with shape: `(batch_size,embedding_size)`.

Arguments

- **use_bias** : Boolean, if use bias.
- **seed** : A Python integer to use as random seed.

References

- [FLEN: Leveraging Field for Scalable CTR Prediction](https://arxiv.org/pdf/1911.04690)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**FwFMLayer**(*num_fields=4*, *regularizer=1e-06*, ***kwargs*)
Field-weighted Factorization Machines

Input shape

- 3D tensor with shape: `(batch_size,field_size,embedding_size)`.

Output shape

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

Arguments

- **num_fields** : integer for number of fields
- **regularizer** : L2 regularizer weight for the field strength parameters of FwFM

References

- [Field-weighted Factorization Machines for Click-Through Rate Prediction in Display Advertising]

https://arxiv.org/pdf/1806.03514.pdf

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**InnerProductLayer**(*reduce_sum=True*, ***kwargs*)
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

- [Qu Y, Cai H, Ren K, et al. Product-based neural networks for user response prediction[C]//Data Mining (ICDM), 2016 IEEE 16th International Conference on. IEEE, 2016: 1149-1154.](https://arxiv.org/pdf/1611.00144.pdf)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**InteractingLayer**(*att_embedding_size=8*, *head_num=2*, *use_res=True*, *seed=1024*, ***kwargs*)

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

- **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

- [Song W, Shi C, Xiao Z, et al. AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks[J]. arXiv preprint arXiv:1810.11921, 2018.](https://arxiv.org/abs/1810.11921)

build(*input_shape*)

Creates the variables of the layer.

call(*inputs*, ***kwargs*)

This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**OutterProductLayer**(*kernel_type='mat'*, *seed=1024*, ***kwargs*)

OutterProduct Layer used in PNN.This implemention is adapted from code that the author of the paper published on 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*(N-1)/2 )`.

Arguments

- **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

- [Qu Y, Cai H, Ren K, et al. Product-based neural networks for user response prediction[C]//Data Mining (ICDM), 2016 IEEE 16th International Conference on. IEEE, 2016: 1149-1154.](https://arxiv.org/pdf/1611.00144.pdf)

build(*input_shape*)

Creates the variables of the layer.

call(*inputs*, ***kwargs*)

This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.interaction.**SENETLayer**(*reduction_ratio=3*, *seed=1024*, ***kwargs*)

SENETLayer used in FiBiNET.

Input shape

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

Output shape

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

Arguments

- **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](https://arxiv.org/pdf/1905.09433.pdf)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, *training=None*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_mask(*inputs*, *mask=None*)
Computes an output mask tensor.

Arguments: inputs: Tensor or list of tensors. mask: Tensor or list of tensors.

Returns:

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

2.8.3 deepctr.layers.activation module

Author: Weichen Shen,wcshen1994@163.com

class deepctr.layers.activation.**Dice**(*axis=-1*, *epsilon=1e-09*, ***kwargs*)
The Data Adaptive Activation Function in DIN,which can be viewed as a generalization of PReLu and can adaptively adjust the rectified point according to distribution of input data.

Input shape

- Arbitrary. Use the keyword argument *input_shape* (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output shape

- Same shape as the input.

Arguments

- **axis** : Integer, the axis that should be used to compute data distribution (typically the features axis).
- **epsilon** : Small float added to variance to avoid dividing by zero.

References

- [Zhou G, Zhu X, Song C, et al. Deep interest network for click-through rate prediction[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. ACM, 2018: 1059-1068.](https://arxiv.org/pdf/1706.06978.pdf)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, *training=None*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

2.8.4 deepctr.layers.normalization module

Author: Weichen Shen,wcshen1994@163.com

class deepctr.layers.normalization.**LayerNormalization**(*axis=-1*, *eps=1e-09*, *center=True*, *scale=True*, ***kwargs*)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

2.8.5 deepctr.layers.sequence module

Author: Weichen Shen,wcshen1994@163.com

class deepctr.layers.sequence.**AttentionSequencePoolingLayer**(*att_hidden_units=(80, 40), att_activation='sigmoid', weight_normalization=False, return_score=False, supports_masking=False, **kwargs*)

The Attentional sequence pooling operation used in DIN.

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)`.

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

- [Zhou G, Zhu X, Song C, et al. Deep interest network for click-through rate prediction[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. ACM, 2018: 1059-1068.](https://arxiv.org/pdf/1706.06978.pdf)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, *mask=None*, *training=None*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_mask(*inputs*, *mask*)
Computes an output mask tensor.

Arguments: inputs: Tensor or list of tensors. mask: Tensor or list of tensors.

Returns:

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.sequence.**BiLSTM**(*units*, *layers=2*, *res_layers=0*, *dropout_rate=0.2*, *merge_mode='ave'*, ***kwargs*)
A multiple layer Bidirectional Residual LSTM Layer.

Input shape

- 3D tensor with shape `(batch_size, timesteps, input_dim)`.

Output shape

- 3D tensor with shape: `(batch_size, timesteps, units)`.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **layers**:Positive integer, number of LSTM layers to stacked.

- **res_layers**: Positive integer, number of residual connection to used in last `res_layers`.
- **dropout_rate**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **merge_mode**: merge_mode: Mode by which outputs of the forward and backward RNNs will be combined. One of { `'fw'` , `'bw'` , `'sum'` , `'mul'` , `'concat'` , `'ave'` , `None` }. If None, the outputs will not be combined, they will be returned as a list.

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, *mask=None*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_mask(*inputs*, *mask*)
Computes an output mask tensor.

Arguments: inputs: Tensor or list of tensors. mask: Tensor or list of tensors.

Returns:

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.sequence.**BiasEncoding**(*sess_max_count*, *seed=1024*, ***kwargs*)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, *mask=None*)

Parameters **concated_embeds_value** – None * field_size * embedding_size

Returns None*1

compute_mask(*inputs*, *mask=None*)
Computes an output mask tensor.

Arguments: inputs: Tensor or list of tensors. mask: Tensor or list of tensors.

Returns:

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.sequence.**DynamicGRU**(*num_units=None*, *gru_type='GRU'*, *return_sequence=True*, ***kwargs*)

build(*input_shape*)
Creates the variables of the layer.

call(*input_list*)

Parameters **concated_embeds_value** – None * field_size * embedding_size

Returns None*1

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.sequence.**KMaxPooling**(*k=1*, *axis=-1*, ***kwargs*)
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.

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.sequence.**SequencePoolingLayer**(*mode='mean'*, *supports_masking=False*, ***kwargs*)
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.
- **supports_masking**:If True,the input need to support masking.

build(*input_shape*)
Creates the variables of the layer.

call(*seq_value_len_list*, *mask=None*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_mask(*inputs*, *mask*)
Computes an output mask tensor.

Arguments: inputs: Tensor or list of tensors. mask: Tensor or list of tensors.

Returns:

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.sequence.**Transformer**(*att_embedding_size=1*, *head_num=8*, *dropout_rate=0.0*, *use_positional_encoding=True*, *use_res=True*, *use_feed_forward=True*, *use_layer_norm=False*, *blinding=True*, *seed=1024*, *supports_masking=False*, ***kwargs*)
Simplified version of Transformer proposed in Attention is all you need

Input shape

- a list of two 3D tensor with shape `(batch_size, timesteps, input_dim)` if supports_masking=True.
- a list of two 4 tensors, first two tensors with shape `(batch_size, timesteps, input_dim)`,last two tensors with shape `(batch_size, 1)` if supports_masking=False.

Output shape

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

Arguments

- **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.
- **dropout_rate**: float between 0 and 1. Fraction of the units to drop.
- **use_positional_encoding**: bool. Whether or not use positional_encoding
- **use_res**: bool. Whether or not use standard residual connections before output.
- **use_feed_forward**: bool. Whether or not use pointwise feed foward network.
- **use_layer_norm**: bool. Whether or not use Layer Normalization.
- **blinding**: bool. Whether or not use blinding.
- **seed**: A Python integer to use as random seed.
- **supports_masking**:bool. Whether or not support masking.

References

- [Vaswani, Ashish, et al. "Attention is all you need." Advances in Neural Information Processing Systems. 2017.](https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf)

build(*input_shape*)
Creates the variables of the layer.

call(*inputs*, *mask=None*, *training=None*, ***kwargs*)
This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_mask(*inputs*, *mask=None*)
Computes an output mask tensor.

Arguments: inputs: Tensor or list of tensors. mask: Tensor or list of tensors.

Returns:

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

class deepctr.layers.sequence.**WeightedSequenceLayer**(*weight_normalization=True*, *supports_masking=False*, ***kwargs*)

The WeightedSequenceLayer is used to apply weight score on variable-length sequence feature/multi-value feature.

Input shape

- A list of two tensor [seq_value,seq_len,seq_weight]
- 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.
- seq_weight is a 3D tensor with shape: `(batch_size, T, 1)`

Output shape

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

Arguments

- **weight_normalization**: bool.Whether normalize the weight score before applying to sequence.
- **supports_masking**:If True,the input need to support masking.

build(*input_shape*)

Creates the variables of the layer.

call(*input_list*, *mask=None*, ***kwargs*)

This is where the layer's logic lives.

Arguments: inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

Returns: A tensor or list/tuple of tensors.

compute_mask(*inputs*, *mask*)

Computes an output mask tensor.

Arguments: inputs: Tensor or list of tensors. mask: Tensor or list of tensors.

Returns:

None or a tensor (or list of tensors, one per output tensor of the layer).

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Arguments:

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns: An input shape tuple.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

Returns: Python dictionary.

deepctr.layers.sequence.**positional_encoding**(*inputs*, *pos_embedding_trainable=True*, *zero_pad=False*, *scale=True*)

Sinusoidal Positional_Encoding.

Args:

- inputs: A 2d Tensor with shape of (N, T).
- num_units: Output dimensionality
- zero_pad: Boolean. If True, all the values of the first row (id = 0) should be constant zero
- scale: Boolean. If True, the output will be multiplied by sqrt num_units(check details from paper)
- scope: Optional scope for *variable_scope*.
- reuse: Boolean, whether to reuse the weights of a previous layer by the same name.

Returns:

- A 'Tensor' with one more rank than inputs's, with the dimensionality should be 'num_units'

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

deepctr.estimator.models.afm, 69
deepctr.estimator.models.autoint, 72
deepctr.estimator.models.ccpm, 63
deepctr.estimator.models.dcn, 70
deepctr.estimator.models.deepfm, 67
deepctr.estimator.models.fibinet, 73
deepctr.estimator.models.fnn, 64
deepctr.estimator.models.nfm, 68
deepctr.estimator.models.pnn, 65
deepctr.estimator.models.wdl, 66
deepctr.estimator.models.xdeepfm, 71
deepctr.layers.activation, 88
deepctr.layers.core, 75
deepctr.layers.interaction, 77
deepctr.layers.normalization, 89
deepctr.layers.sequence, 90
deepctr.models.afm, 54
deepctr.models.autoint, 59
deepctr.models.ccpm, 50
deepctr.models.dcn, 55
deepctr.models.deepfm, 53
deepctr.models.dien, 57
deepctr.models.din, 56
deepctr.models.dsin, 57
deepctr.models.fgcnn, 61
deepctr.models.fibinet, 61
deepctr.models.flen, 62
deepctr.models.fnn, 51
deepctr.models.mlr, 53
deepctr.models.nfm, 54
deepctr.models.onn, 60
deepctr.models.pnn, 51
deepctr.models.wdl, 52
deepctr.models.xdeepfm, 58

Index

A

AFM() (*in module deepctr.models.afm*), 54
AFMEstimator() (*in module deepctr.estimator.models.afm*), 69
AFMLayer (*class in deepctr.layers.interaction*), 77
AttentionSequencePoolingLayer (*class in deepctr.layers.sequence*), 90
AutoInt() (*in module deepctr.models.autoint*), 59
AutoIntEstimator() (*in module deepctr.estimator.models.autoint*), 72

B

BiasEncoding (*class in deepctr.layers.sequence*), 92
BiInteractionPooling (*class in deepctr.layers.interaction*), 78
BilinearInteraction (*class in deepctr.layers.interaction*), 79
BiLSTM (*class in deepctr.layers.sequence*), 91
build() (*deepctr.layers.activation.Dice method*), 89
build() (*deepctr.layers.core.DNN method*), 75
build() (*deepctr.layers.core.LocalActivationUnit method*), 76
build() (*deepctr.layers.core.PredictionLayer method*), 77
build() (*deepctr.layers.interaction.AFMLayer method*), 78
build() (*deepctr.layers.interaction.BiInteractionPooling method*), 79
build() (*deepctr.layers.interaction.BilinearInteraction method*), 79
build() (*deepctr.layers.interaction.CIN method*), 80
build() (*deepctr.layers.interaction.CrossNet method*), 81
build() (*deepctr.layers.interaction.FGCNNLayer method*), 82
build() (*deepctr.layers.interaction.FieldWiseBiInteraction method*), 83
build() (*deepctr.layers.interaction.FM method*), 83
build() (*deepctr.layers.interaction.FwFMLayer method*), 84
build() (*deepctr.layers.interaction.InnerProductLayer method*), 85
build() (*deepctr.layers.interaction.InteractingLayer method*), 86
build() (*deepctr.layers.interaction.OutterProductLayer method*), 87
build() (*deepctr.layers.interaction.SENETLayer method*), 88
build() (*deepctr.layers.normalization.LayerNormalization method*), 89
build() (*deepctr.layers.sequence.AttentionSequencePoolingLayer method*), 91
build() (*deepctr.layers.sequence.BiasEncoding method*), 92
build() (*deepctr.layers.sequence.BiLSTM method*), 92
build() (*deepctr.layers.sequence.DynamicGRU method*), 93
build() (*deepctr.layers.sequence.KMaxPooling method*), 94
build() (*deepctr.layers.sequence.SequencePoolingLayer method*), 94
build() (*deepctr.layers.sequence.Transformer method*), 96
build() (*deepctr.layers.sequence.WeightedSequenceLayer method*), 97

C

call() (*deepctr.layers.activation.Dice method*), 89
call() (*deepctr.layers.core.DNN method*), 75
call() (*deepctr.layers.core.LocalActivationUnit method*), 76
call() (*deepctr.layers.core.PredictionLayer method*), 77
call() (*deepctr.layers.interaction.AFMLayer method*), 78
call() (*deepctr.layers.interaction.BiInteractionPooling method*), 79
call() (*deepctr.layers.interaction.BilinearInteraction method*), 79

call() (*deepctr.layers.interaction.CIN method*), 80
call() (*deepctr.layers.interaction.CrossNet method*), 81
call() (*deepctr.layers.interaction.FGCNNLayer method*), 82
call() (*deepctr.layers.interaction.FieldWiseBiInteraction method*), 83
call() (*deepctr.layers.interaction.FM method*), 83
call() (*deepctr.layers.interaction.FwFMLayer method*), 84
call() (*deepctr.layers.interaction.InnerProductLayer method*), 85
call() (*deepctr.layers.interaction.InteractingLayer method*), 86
call() (*deepctr.layers.interaction.OutterProductLayer method*), 87
call() (*deepctr.layers.interaction.SENETLayer method*), 88
call() (*deepctr.layers.normalization.LayerNormalization method*), 89
call() (*deepctr.layers.sequence.AttentionSequencePoolingLayer method*), 91
call() (*deepctr.layers.sequence.BiasEncoding method*), 92
call() (*deepctr.layers.sequence.BiLSTM method*), 92
call() (*deepctr.layers.sequence.DynamicGRU method*), 93
call() (*deepctr.layers.sequence.KMaxPooling method*), 94
call() (*deepctr.layers.sequence.SequencePoolingLayer method*), 95
call() (*deepctr.layers.sequence.Transformer method*), 96
call() (*deepctr.layers.sequence.WeightedSequenceLayer method*), 97
CCPM() (*in module deepctr.models.ccpm*), 50
CCPMEstimator() (*in module deepctr.estimator.models.ccpm*), 63
CIN (*class in deepctr.layers.interaction*), 80
compute_mask() (*deepctr.layers.core.LocalActivationUnit method*), 76
compute_mask() (*deepctr.layers.interaction.SENETLayer method*), 88
compute_mask() (*deepctr.layers.sequence.AttentionSequencePoolingLayer method*), 91
compute_mask() (*deepctr.layers.sequence.BiasEncoding method*), 92
compute_mask() (*deepctr.layers.sequence.BiLSTM method*), 92
compute_mask() (*deepctr.layers.sequence.SequencePoolingLayer method*), 95
compute_mask() (*deepctr.layers.sequence.Transformer method*), 96
compute_mask() (*deepctr.layers.sequence.WeightedSequenceLayer method*), 97
compute_output_shape() (*deepctr.layers.activation.Dice method*), 89
compute_output_shape() (*deepctr.layers.core.DNN method*), 75
compute_output_shape() (*deepctr.layers.core.LocalActivationUnit method*), 76
compute_output_shape() (*deepctr.layers.core.PredictionLayer method*), 77
compute_output_shape() (*deepctr.layers.interaction.AFMLayer method*), 78
compute_output_shape() (*deepctr.layers.interaction.BiInteractionPooling method*), 79
compute_output_shape() (*deepctr.layers.interaction.BilinearInteraction method*), 79
compute_output_shape() (*deepctr.layers.interaction.CIN method*), 80
compute_output_shape() (*deepctr.layers.interaction.CrossNet method*), 81
compute_output_shape() (*deepctr.layers.interaction.FGCNNLayer method*), 82
compute_output_shape() (*deepctr.layers.interaction.FieldWiseBiInteraction method*), 83
compute_output_shape() (*deepctr.layers.interaction.FM method*), 83
compute_output_shape() (*deepctr.layers.interaction.FwFMLayer method*), 84
compute_output_shape() (*deepctr.layers.interaction.InnerProductLayer method*), 85
compute_output_shape() (*deepctr.layers.interaction.InteractingLayer method*), 86
compute_output_shape() (*deepctr.layers.interaction.OutterProductLayer method*), 87
compute_output_shape() (*deepctr.layers.interaction.SENETLayer method*), 88
compute_output_shape() (*deepctr.layers.normalization.LayerNormalization method*), 90

compute_output_shape() (*deepctr.layers.sequence.AttentionSequencePoolingLayer method*), 91
compute_output_shape() (*deepctr.layers.sequence.BiasEncoding method*), 93
compute_output_shape() (*deepctr.layers.sequence.BiLSTM method*), 92
compute_output_shape() (*deepctr.layers.sequence.DynamicGRU method*), 93
compute_output_shape() (*deepctr.layers.sequence.KMaxPooling method*), 94
compute_output_shape() (*deepctr.layers.sequence.SequencePoolingLayer method*), 95
compute_output_shape() (*deepctr.layers.sequence.Transformer method*), 96
compute_output_shape() (*deepctr.layers.sequence.WeightedSequenceLayer method*), 97
CrossNet (*class in deepctr.layers.interaction*), 81

D

DCN() (*in module deepctr.models.dcn*), 55
DCNEstimator() (*in module deepctr.estimator.models.dcn*), 70
deepctr.estimator.models.afm (*module*), 69
deepctr.estimator.models.autoint (*module*), 72
deepctr.estimator.models.ccpm (*module*), 63
deepctr.estimator.models.dcn (*module*), 70
deepctr.estimator.models.deepfm (*module*), 67
deepctr.estimator.models.fibinet (*module*), 73
deepctr.estimator.models.fnn (*module*), 64
deepctr.estimator.models.nfm (*module*), 68
deepctr.estimator.models.pnn (*module*), 65
deepctr.estimator.models.wdl (*module*), 66
deepctr.estimator.models.xdeepfm (*module*), 71
deepctr.layers.activation (*module*), 88
deepctr.layers.core (*module*), 75
deepctr.layers.interaction (*module*), 77
deepctr.layers.normalization (*module*), 89
deepctr.layers.sequence (*module*), 90
deepctr.models.afm (*module*), 54
deepctr.models.autoint (*module*), 59
deepctr.models.ccpm (*module*), 50
deepctr.models.dcn (*module*), 55
deepctr.models.deepfm (*module*), 53
deepctr.models.dien (*module*), 57
deepctr.models.din (*module*), 56
deepctr.models.dsin (*module*), 57
deepctr.models.fgcnn (*module*), 61
deepctr.models.fibinet (*module*), 61
deepctr.models.flen (*module*), 62
deepctr.models.fnn (*module*), 51
deepctr.models.mlr (*module*), 53
deepctr.models.nfm (*module*), 54
deepctr.models.onn (*module*), 60
deepctr.models.pnn (*module*), 51
deepctr.models.wdl (*module*), 52
deepctr.models.xdeepfm (*module*), 58
DeepFM() (*in module deepctr.models.deepfm*), 53
DeepFMEstimator() (*in module deepctr.estimator.models.deepfm*), 67
Dice (*class in deepctr.layers.activation*), 88
DIEN() (*in module deepctr.models.dien*), 57
DIN() (*in module deepctr.models.din*), 56
DNN (*class in deepctr.layers.core*), 75
DSIN() (*in module deepctr.models.dsin*), 57
DynamicGRU (*class in deepctr.layers.sequence*), 93

F

FGCNN() (*in module deepctr.models.fgcnn*), 61
FGCNNLayer (*class in deepctr.layers.interaction*), 82
FiBiNET() (*in module deepctr.models.fibinet*), 61
FiBiNETEstimator() (*in module deepctr.estimator.models.fibinet*), 73
FieldWiseBiInteraction (*class in deepctr.layers.interaction*), 83
FLEN() (*in module deepctr.models.flen*), 62
FM (*class in deepctr.layers.interaction*), 82
FNN() (*in module deepctr.models.fnn*), 51
FNNEstimator() (*in module deepctr.estimator.models.fnn*), 64
FwFMLayer (*class in deepctr.layers.interaction*), 84

G

get_config() (*deepctr.layers.activation.Dice method*), 89
get_config() (*deepctr.layers.core.DNN method*), 75
get_config() (*deepctr.layers.core.LocalActivationUnit method*), 77
get_config() (*deepctr.layers.core.PredictionLayer method*), 77
get_config() (*deepctr.layers.interaction.AFMLayer method*), 78
get_config() (*deepctr.layers.interaction.BilinearInteraction method*), 80
get_config() (*deepctr.layers.interaction.CIN method*), 81

get_config() (*deepctr.layers.interaction.CrossNet method*), 82
get_config() (*deepctr.layers.interaction.FGCNNLayer method*), 82
get_config() (*deepctr.layers.interaction.FieldWiseBiInteraction method*), 84
get_config() (*deepctr.layers.interaction.FwFMLayer method*), 84
get_config() (*deepctr.layers.interaction.InnerProductLayer method*), 85
get_config() (*deepctr.layers.interaction.InteractingLayer method*), 86
get_config() (*deepctr.layers.interaction.OutterProductLayer method*), 87
get_config() (*deepctr.layers.interaction.SENETLayer method*), 88
get_config() (*deepctr.layers.normalization.LayerNormalization method*), 90
get_config() (*deepctr.layers.sequence.AttentionSequencePoolingLayer method*), 91
get_config() (*deepctr.layers.sequence.BiasEncoding method*), 93
get_config() (*deepctr.layers.sequence.BiLSTM method*), 92
get_config() (*deepctr.layers.sequence.DynamicGRU method*), 93
get_config() (*deepctr.layers.sequence.KMaxPooling method*), 94
get_config() (*deepctr.layers.sequence.SequencePoolingLayer method*), 95
get_config() (*deepctr.layers.sequence.Transformer method*), 96
get_config() (*deepctr.layers.sequence.WeightedSequenceLayer method*), 97

I

InnerProductLayer (*class in deepctr.layers.interaction*), 85
InteractingLayer (*class in deepctr.layers.interaction*), 86

K

KMaxPooling (*class in deepctr.layers.sequence*), 93

L

LayerNormalization (*class in deepctr.layers.normalization*), 89
LocalActivationUnit (*class in deepctr.layers.core*), 76

M

MLR() (*in module deepctr.models.mlr*), 53

N

NFM() (*in module deepctr.models.nfm*), 54
NFMEstimator() (*in module deepctr.estimator.models.nfm*), 68

O

ONN() (*in module deepctr.models.onn*), 60
OutterProductLayer (*class in deepctr.layers.interaction*), 86

P

PNN() (*in module deepctr.models.pnn*), 51
PNNEstimator() (*in module deepctr.estimator.models.pnn*), 65
positional_encoding() (*in module deepctr.layers.sequence*), 97
PredictionLayer (*class in deepctr.layers.core*), 77

S

SENETLayer (*class in deepctr.layers.interaction*), 87
SequencePoolingLayer (*class in deepctr.layers.sequence*), 94

T

Transformer (*class in deepctr.layers.sequence*), 95

W

WDL() (*in module deepctr.models.wdl*), 52
WDLEstimator() (*in module deepctr.estimator.models.wdl*), 66
WeightedSequenceLayer (*class in deepctr.layers.sequence*), 96

X

xDeepFM() (*in module deepctr.models.xdeepfm*), 58
xDeepFMEstimator() (*in module deepctr.estimator.models.xdeepfm*), 71