DeepTables

Release 0.1.1

Sep 27, 2020

Home:

CHAPTER 1

DeepTables: Deep-learning Toolkit for Tabular data

DeepTables(DT) is a easy-to-use toolkit that enables deep learning to unleash great power on tabular data.

CHAPTER 2

Overview

MLP (also known as Fully-connected neural networks) have been shown inefficient in learning distribution representation. The “add” operations of the perceptron layer have been proven poor performance to exploring multiplicative feature interactions. In most cases, manual feature engineering is necessary and this work requires extensive domain knowledge and very cumbersome. How learning feature interactions efficiently in neural networks becomes the most important problem.

A lot of models have been proposed to CTR prediction and continue to outperform existing state-of-the-art approaches to the late years. Well-known examples include FM, DeepFM, Wide&Deep, DCN, PNN, etc. These models can also provide good performance on tabular data under reasonable utilization.

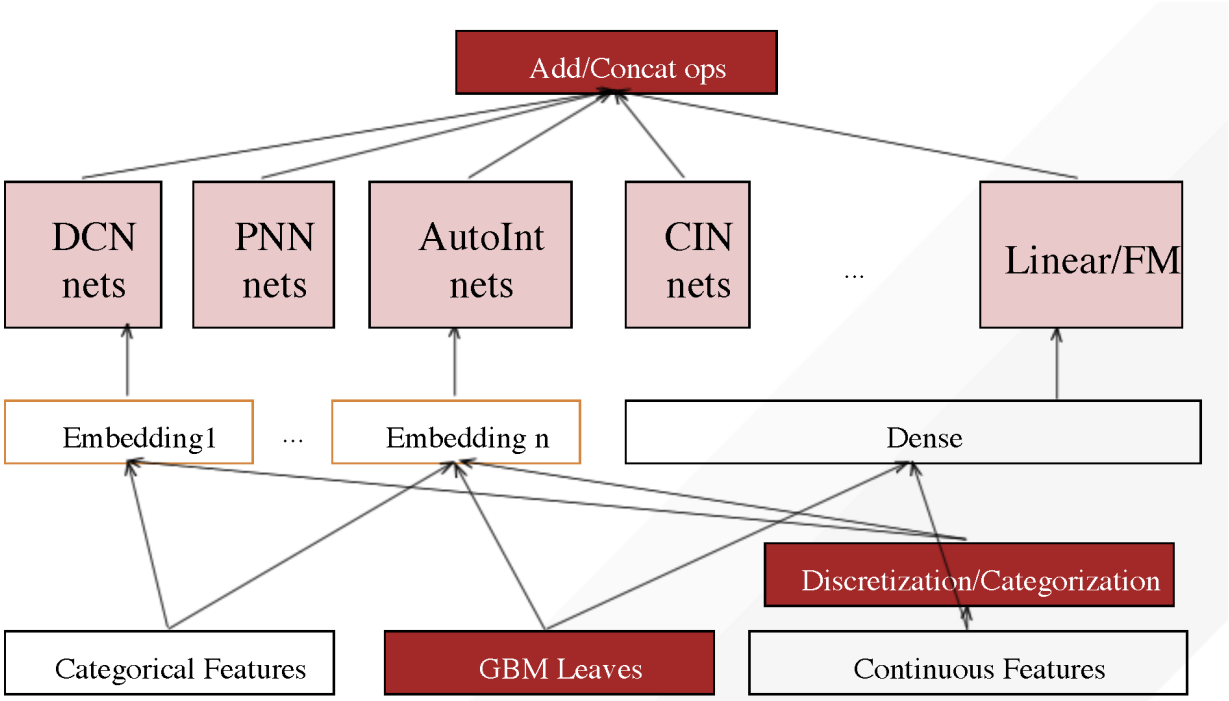
DT aims to utilize the latest research findings to provide users with an end-to-end toolkit on tabular data.

DT has been designed with these key goals in mind:

- Easy to use, non-experts can also use.
- Provide good performance out of the box.
- Flexible architecture and easy expansion by user.

DT follow these steps to build a neural network:

1. Category features -> Embedding Layer.
2. Continuous feature -> Dense Layer or to Embedding Layer after discretization/categorization.
3. Embedding/Dense layers -> Feature Interactions/Extractions nets.
4. Stacking(add/concat) outputs of nets as the output of the model.


Add/Concat ops
DCN nets
PNN nets
AutoInt nets
CIN nets
...
Linear/FM
Embedding1
...
Embedding n
Dense
Discretization/Categorization
Categorical Features
GBM Leaves
Continuous Features

CHAPTER 3

Why use DeepTables?

- Free preprocessing and processing.
  - Easy to expert data scientist or a business analyst without modeling ability.
  - Simpler than the traditional machine learning algorithm which highly depends on manual feature engineering.
- Excellent performance out of the box.
  - Builtin a group of neural network components (NETs) from the most excellent research results in recent years.
- Extremely easy to use.
  - Only 5 lines of code can complete the modeling of any data set.
- Very open architecture design.
  - supports plug-in extension.

CHAPTER 4

Example

```
from deeptables.models.deeptable import DeepTable, ModelConfig
from deeptables.models.deepnets import DeepFM

dt = DeepTable(ModelConfig(nets=DeepFM))
dt.fit(X, y)
preds = dt.predict(X_test)
```

4.1 Quick-Start

4.1.1 Installation Guide

Requirements

Python 3: DT requires Python version 3.6 or 3.7.

Tensorflow >= 2.0.0: DT is based on TensorFlow. Please follow this tutorial to install TensorFlow for python3.

Install DeepTables

```
pip install deeptables
```

GPU Setup (Optional): If you have GPUs on your machine and want to use them to accelerate the training, you can use the following command.

```
pip install deeptables[gpu]
```

Verify the install:

```
python -c "from deeptables.utils.quicktest import test; test()"
```

Launch a DeepTables Docker Container

You can also quickly try DeepTables through the Docker:

1. Pull a DeepTables image (optional).
2. Launch Docker container.

Pull the latest image:

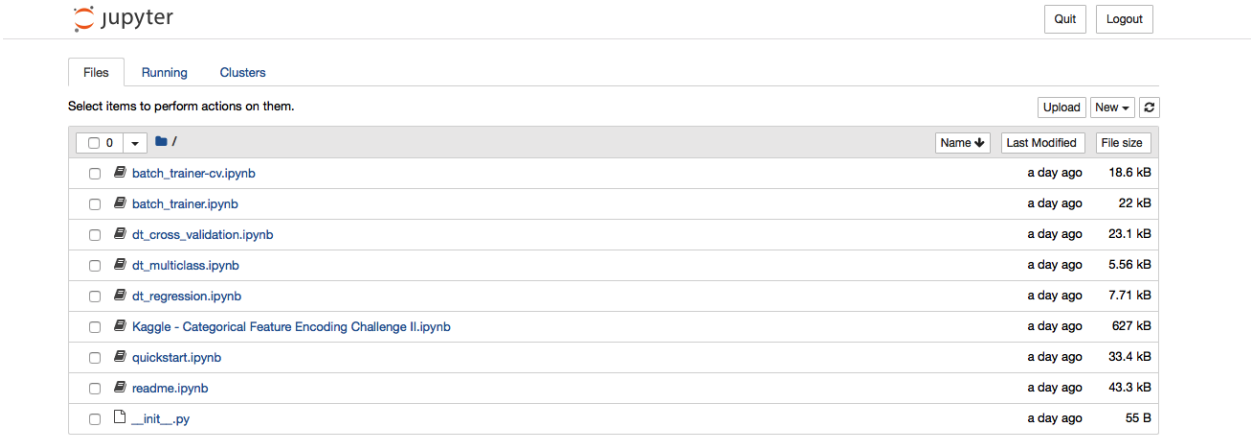
```
docker pull datacanvas/deeptables-example
```

Then launch Docker container with this command line:

```
docker run -it -p 8830:8888 -e NotebookToken="your-token"  datacanvas/deeptables-
↪example
```

The value "your-token" is a user specified string for the notebook and can be empty.

As a result, notebook server should be running at: https://host-ip-address:8830?token=your-token Launch a browser and connect to that URL you will see the Jupyter Notebook like this:



4.1.2 Getting started: 5 lines to DT

Supported Tasks

DT can be use to solve **classification** and **regression** prediction problems on tabular data.

Simple Example

DT supports these tasks with extremely simple interface without dealing with data cleaning and feature engineering. You don't even specify the task type, DT will automatically infer.

```
from deeptables.models.deeptable import DeepTable, ModelConfig
from deeptables.models.deepnets import DeepFM

dt = DeepTable(ModelConfig(nets=DeepFM))
dt.fit(X, y)
preds = dt.predict(X_test)
```

4.1.3 Datasets

DT has several build-in datasets for the demos or testing which covered binary classification, multi-class classification and regression task. All datasets are accessed through `deeptables.datasets.dsutils`.

Adult

Associated Tasks: **Binary Classification**

Predict whether income exceeds $50K/yr based on census data. Also known as "Census Income" dataset.

```
from deeptables.datasets import dsutils
df = dsutils.load_adult()
```

See: http://archive.ics.uci.edu/ml/datasets/Adult

Glass Identification

Associated Tasks: **Multi-class Classification**

From USA Forensic Science Service; 6 types of glass; defined in terms of their oxide content (i.e. Na, Fe, K, etc)

```
from deeptables.datasets import dsutils
df = dsutils.load_glass_uci()
```

See: http://archive.ics.uci.edu/ml/datasets/Glass+Identification

Boston house-prices

Associated Tasks: **Regression**

```
from deeptables.datasets import dsutils
df = dsutils.load_boston()
```

See: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_boston.html

Examples

See: Examples

4.2 Exapmles

4.2.1 Binary Classification

This example demonstrate how to use WideDeep nets to solve a binary classification prediction problem.

```
from deeptables.models.deeptable import DeepTable, ModelConfig
from deeptables.models.deepnets import WideDeep
from deeptables.datasets import dsutils
from sklearn.model_selection import train_test_split

```

(continues on next page)

(continued from previous page)

```
#Adult Data Set from UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/
↪datasets/Adult
df_train = dsutils.load_adult()
y = df_train.pop(14)
X = df_train

#`auto_discrete` is used to decide wether to discretize continous varibles␣
↪automatically.
conf = ModelConfig(nets=WideDeep, metrics=['AUC','accuracy'], auto_discrete=True)
dt = DeepTable(config=conf)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=42)

model, history = dt.fit(X_train, y_train, epochs=100)

score = dt.evaluate(X_test, y_test)

preds = dt.predict(X_test)
```

4.2.2 Multiclass Classification

This simple example demonstrate how to use a DNN(MLP) nets to solve a multiclass task on MNIST dataset.

```
from deeptables.models import deeptable
from tensorflow import keras

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255

conf = deeptable.ModelConfig(nets=['dnn_nets'], optimizer=keras.optimizers.RMSprop())
dt = deeptable.DeepTable(config=conf)

model, history = dt.fit(x_train, y_train, epochs=10)

score = dt.evaluate(x_test, y_test, batch_size=512, verbose=0)

preds = dt.predict(x_test)
```

4.2.3 Regression

This example shows how to use DT to predicting Boston housing price.

```
from deeptables.models.deeptable import DeepTable, ModelConfig
from deeptables.datasets import dsutils
from sklearn.model_selection import train_test_split

df_train = dsutils.load_boston()
y = df_train.pop('target')
X = df_train

conf = ModelConfig(
```

(continues on next page)

(continued from previous page)

```
    metrics=['RootMeanSquaredError'],
    nets=['dnn_nets'],
    dnn_params={
        'hidden_units': ((256, 0.3, True), (256, 0.3, True)),
        'dnn_activation': 'relu',
    },
    earlystopping_patience=5,
)

dt = DeepTable(config=conf)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=42)
model, history = dt.fit(X_train, y_train, epochs=100)

score = dt.evaluate(X_test, y_test)
```

4.3 ModelConfig

ModelConfig is the most important parameter in DT. It is used to set how to clean and preprocess the data automatically, and how to assemble various network components to building a neural nets for prediction tasks, as well as the setting of hyper-parameters of nets, etc. If you do not change any settings in ModelConfig, DT will work in most cases as well. However, you can get a better performance by tuning the parameters in ModelConfig.

We describe in detail below.

4.3.1 Simple use case for ModelConfig

```
from deeptables.models.deeptable import DeepTable, ModelConfig
from deeptables.models.deepnets import DeepFM

conf = ModelConfig(
    nets=DeepFM, # same as `nets=['linear','dnn_nets','fm_nets']`
    categorical_columns='auto', # or categorical_columns=['x1', 'x2', 'x3', ...]
    metrics=['AUC', 'accuracy'], # can be `metrics=['RootMeanSquaredError']` for␣
↪regression task
    auto_categorize=True,
    auto_discrete=False,
    embeddings_output_dim=20,
    embedding_dropout=0.3,
    )
dt = DeepTable(config=conf)
dt.fit(X, y)
```

4.3.2 Parameters

nets

list of str or custom function, (default=`['dnn_nets']`)

You can use multiple components to compose neural network joint training to perform prediction tasks.

The value of nets can be any combination of component name, preset model and custom function.

components:

- 'dnn_nets'
- 'linear'
- 'cin_nets'
- 'fm_nets'
- 'afm_nets'
- 'opnn_nets'
- 'ipnn_nets'
- 'pnn_nets',
- 'cross_nets'
- 'cross_dnn_nets'
- 'dcn_nets',
- 'autoint_nets'
- 'fg_nets'
- 'fgcnn_cin_nets'
- 'fgcnn_fm_nets'
- 'fgcnn_ipnn_nets'
- 'fgcnn_dnn_nets'
- 'fibi_nets'
- 'fibi_dnn_nets'

preset models: in package deeptables.models.deepnets

- DeepFM
- xDeepFM
- DCN
- PNN
- WideDeep
- AutoInt
- AFM
- FGCNN
- FibiNet

custom function:

```
def custom_net(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense, config,␣
↪model_desc):
    out = layers.Dense(10)(flatten_emb_layer)
    return out
```

examples:

```
from deeptables.models.deeptable import ModelConfig, DeepTable
from deeptables.models import deepnets
from tensorflow.keras import layers

#preset model
conf = ModelConfig(nets=deepnets.DeepFM)

#list of str(name of component)
conf = ModelConfig(nets=['linear','dnn_nets','cin_nets','cross_nets'])

#mixed preset model and names
conf = ModelConfig(nets=deepnets.WideDeep+['cin_nets'])

#mixed names and custom function
def custom_net(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense, config,␣
↪model_desc):
    out = layers.Dense(10)(flatten_emb_layer)
    return out
conf = ModelConfig(nets=['linear', custom_net])
```

categorical_columns

list of strings or 'auto', optional, (default=`'auto'`)

Only categorical features will be passed into embedding layer, and most of the components in DT are specially designed for the embedding outputs for feature extraction. Reasonable selection of categorical features is critical to model performance.

If **list of strings**, interpreted as column names.

If `'auto'`, get the categorical columns automatically. `object`, `bool` and `category` columns will be selected by default, and [auto_categorize] will no longer take effect.

If not necessary, we strongly recommend use default value `'auto'`.

exclude_columns

list of strings, (default=[])

pos_label

str or int, (default=`None`)

The label of positive class, used only when task is binary.

metrics

list of strings or callable object, (default=`['accuracy']`)

List of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']` or `metrics=['AUC']`. Every metric should be a built-in evaluation metric in tf.keras.metrics or a callable object like `r2(y_true, y_pred):...`.

See also: https://tensorflow.google.cn/versions/r2.0/api_docs/python/tf/keras/metrics

auto_categorize

bool, (default=`False`)

Whether to automatically categorize eligible continuous features.

- `True`:
- `False`:

cat_exponent

float, (default=`0.5`), between 0 and 1

Only usable when auto_categrization = `True`.

Columns with `(number of unique values < number of samples ** cat_exponent)` will be treated as categorical feature.

cat_remain_numeric

bool, (default=`True`)

Only usable when auto_categrization = `True`.

Whether continuous features transformed into categorical retain numerical features.

- `True`:
- `False`:

auto_encode_label

bool, (default=`True`)

Whether to automatically perform label encoding on categorical features.

auto_imputation

bool, (default=`True`)

Whether to automatically perform imputation on all features.

auto_discrete

bool, (default=`False`)

Whether to discretize all continuous features into categorical features.

fixed_embedding_dim

bool, (default=`True`)

Whether the embeddings output of all categorical features uses the same 'output_dim'. It should be noted that some components require that the output_dim of embeddings must be the same, including **FM**, **AFM**, **CIN**, **MultiheadAttention**, **SENET**, **InnerProduct**, etc.

If `False` and embedding_output_dim=0, then the output_dim of embeddings will be calculated using the following formula:

```
min(4 * int(pow(voc_size, 0.25)), 20)
#voc_size is the number of unique values of each feature.
```

embeddings_output_dim

int, (default=`4`)

embeddings_initializer

str or object, (default=`'uniform'`)

Initializer for the `embeddings` matrix.

embeddings_regularizer

str or object, (default=`None`)

Regularizer function applied to the `embeddings` matrix.

dense_dropout

float, (default=`0`) between 0 and 1

Fraction of the dense input units to drop.

embedding_dropout

float, (default=`0.3`) between 0 and 1

Fraction of the embedding input units to drop.

stacking_op

str, (default=`'add'`)

- `'add'`
- `'concat'`

output_use_bias

bool, (default=`True`)

apply_class_weight

bool, (default='False')

Whether to calculate the weight of each class automatically. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

optimizer

str(name of optimizer) or optimizer instance or 'auto', (default=`'auto'`)

See `tf.keras.optimizers`.

- 'auto': Automatically select optimizer based on task type.

loss

str(name of objective function) or objective function or `tf.losses.Loss` instance or 'auto', (default='auto')

See `tf.losses`.

- 'auto': Automatically select objective function based on task type.

home_dir

str, (default=`None`)

The home directory for saving model-related files. Each time running `fit(...)` or `fit_cross_validation(...)`, a subdirectory with a time-stamp will be created in this directory.

monitor_metric

str, (default=`None`)

earlystopping_patience

int, (default=`1`)

gpu_usage_strategy

str, (default=`'memory_growth'`)

- `'memory_growth'`
- `'None'`

distribute_strategy:

tensorflow.python.distribute.distribute_lib.Strategy, (default=`None`)

dnn_params

dictionary Only usable when 'dnn_nets' or a component using 'dnn' like 'pnn_nets','dcn_nets' included in [nets].

```
{
    'hidden_units': ((128, 0, False), (64, 0, False)),
    'dnn_activation': 'relu'}
)
```

autoint_params

dictionary Only usable when 'autoint_nets' included in [nets].

```
{
    'num_attention': 3,
    'num_heads': 1,
    'dropout_rate': 0,
    'use_residual': True
}
```

fgcnn_params

dictionary Only usable when 'fgcnn_nets' or a component using 'fgcnn' included in [nets].

```
{
    'fg_filters': (14, 16),
    'fg_widths': (7, 7),
    'fg_pool_widths': (2, 2),
    'fg_new_feat_filters': (2, 2),
}
```

fibinet_params

dictionary Only usable when 'fibi_nets' included in [nets].

```
{
    'senet_pooling_op': 'mean',
    'senet_reduction_ratio': 3,
    'bilinear_type': 'field_interaction',
}
```

cross_params

dictionary Only usable when 'cross_nets' included in [nets].

```
{
    'num_cross_layer': 4,
}
```

pnn_params

dictionary Only usable when 'pnn_nets' or 'opnn_nets' included in [nets].

```
{
    'outer_product_kernel_type': 'mat',
}
```

afm_params

dictionary Only usable when 'afm_nets' included in [nets].

```
{
    'attention_factor': 4,
    'dropout_rate': 0
}
```

cin_params

dictionary Only usable when 'cin_nets' included in [nets].

```
{
    'cross_layer_size': (128, 128),
    'activation': 'relu',
    'use_residual': False,
    'use_bias': False,
    'direct': False,
    'reduce_D': False,
}
```

4.4 Models

In recent years, a lot of neural nets have been proposed to CTR prediction and continue to outperform existing state-of-the-art approaches. Well-known examples include FM, DeepFM, Wide&Deep, DCN, PNN, etc. DT provides most of these models and will continue to introduce the latest research findings in the future.

4.4.1 Wide&Deep

Cheng, Heng-Tze, et al. "Wide & deep learning for recommender systems." Proceedings of the 1st workshop on deep learning for recommender systems. 2016.

Retrieve from: https://dl.acm.org/doi/abs/10.1145/2988450.2988454

> Wide & Deep learning—jointly trained wide linear models and deep neural networks—to combine the benefits of memorization and generalization for recommender systems. We productionized and evaluated the system on Google Play, a commercial mobile app store with over one billion active users and over one million apps. Online experiment results show that Wide & Deep significantly increased app acquisitions compared with wide-only and deep-only models.

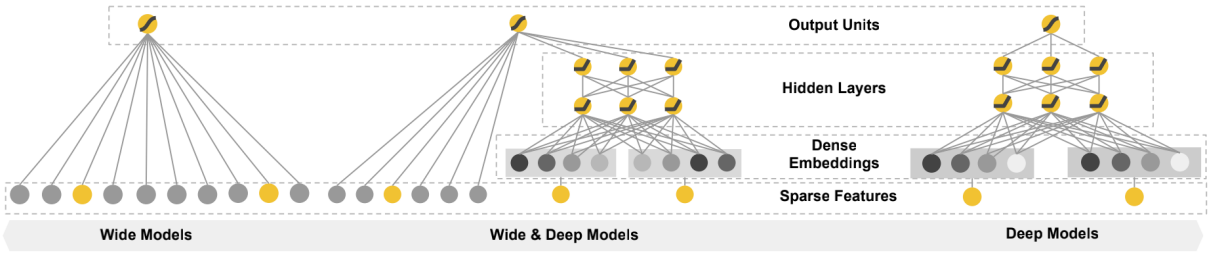


Figure 1: The spectrum of Wide & Deep models.

4.4.2 DCN(Deep & Cross Network)

Wang, Ruoxi, et al. "Deep & cross network for ad click predictions." Proceedings of the ADKDD'17. 2017. 1-7.

Retrived from: https://dl.acm.org/doi/abs/10.1145/3124749.3124754

> Deep & Cross Network (DCN) keeps the benefits of a DNN model, and beyond that, it introduces a novel cross network that is more efficient in learning certain bounded-degree feature interactions. In particular, DCN explicitly applies feature crossing at each layer, requires no manual feature engineering, and adds negligible extra complexity to the DNN model.

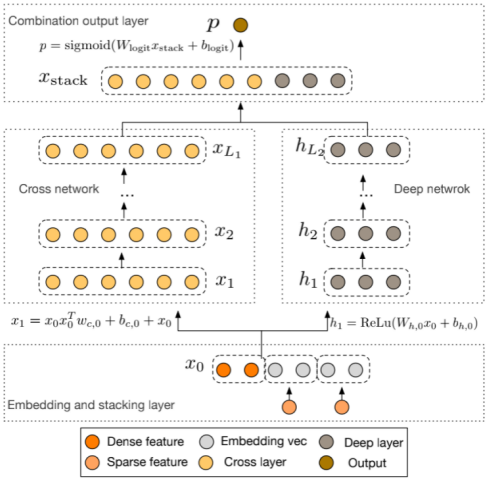


Figure 1: The Deep & Cross Network

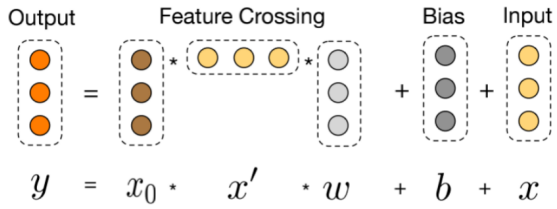


Figure 2: Visualization of a cross layer.

4.4.3 PNN

Qu, Yanru, et al. "Product-based neural networks for user response prediction." 2016 IEEE 16th International Conference on Data Mining (ICDM). IEEE, 2016.

Retrived from: https://ieeexplore.ieee.org/abstract/document/7837964/

> Product-based Neural Networks (PNN) with an embedding layer to learn a distributed representation of the categorical data, a product layer to capture interactive patterns between inter-field categories, and further fully connected layers to explore high-order feature interactions.

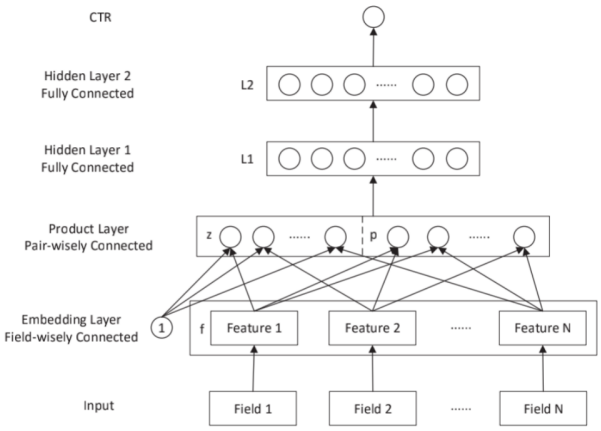


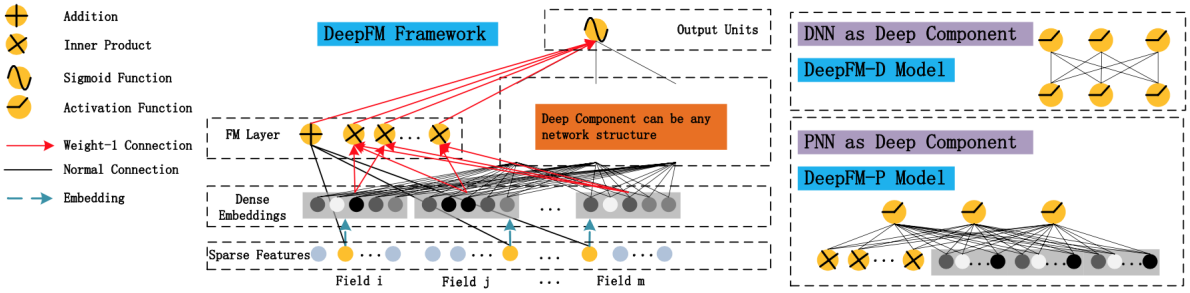
Fig. 1: Product-based Neural Network Architecture.

4.4.4 DeepFM

Guo, Huifeng, et al. "Deepfm: An end-to-end wide & deep learning framework for CTR prediction." arXiv preprint arXiv:1804.04950 (2018).

Retrieve from: https://arxiv.org/abs/1804.04950

> DeepFM, combines the power of factorization machines for recommendation and deep learning for feature learning in a new neural network architecture. Compared to the latest Wide & Deep model from Google, DeepFM has a shared raw feature input to both its "wide" and "deep" components, with no need of feature engineering besides raw features. DeepFM, as a general learning framework, can incorporate various network architectures in its deep component.



4.4.5 xDeepFM

Lian, Jianxun, et al. "xdeepfm: Combining explicit and implicit feature interactions for recommender systems." Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018.

Retrieve from: https://dl.acm.org/doi/abs/10.1145/3219819.3220023

> A novel Compressed Interaction Network (CIN), which aims to generate feature interactions in an explicit fashion and at the vector-wise level. We show that the CIN share some functionalities with convolutional neural networks (CNNs) and recurrent neural networks (RNNs). We further combine a CIN and a classical DNN into one unified model, and named this new model eXtreme Deep Factorization Machine (xDeepFM).

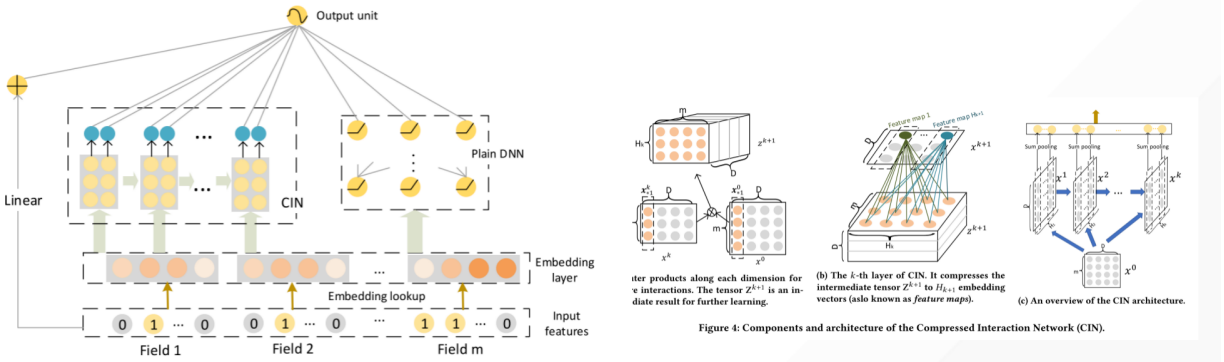


Figure 5: The architecture of xDeepFM.

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

4.4.6 AFM

Xiao, Jun, et al. "Attentional factorization machines: Learning the weight of feature interactions via attention networks." arXiv preprint arXiv:1708.04617 (2017).

Retrieve from: https://arxiv.org/abs/1708.04617

> Attentional Factorization Machine (AFM), which learns the importance of each feature interaction from data via a neural attention network. Extensive experiments on two real-world datasets demonstrate the effectiveness of AFM. Empirically, it is shown on regression task AFM betters FM with a 8.6% relative improvement, and consistently outperforms the state-of-the-art deep learning methods Wide&Deep and DeepCross with a much simpler structure and fewer model parameters.

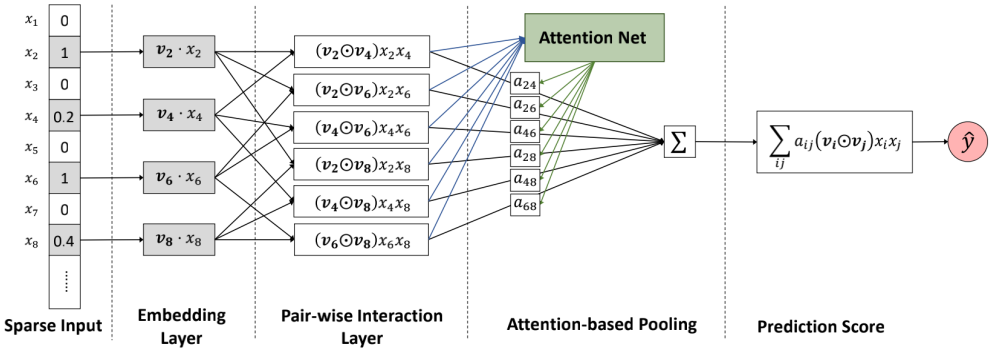


Figure 1: The neural network architecture of our proposed Attentional Factorization Machine model.

4.4.7 AutoInt

Song, Weiping, et al. "Autoint: Automatic feature interaction learning via self-attentive neural networks." Proceedings of the 28th ACM International Conference on Information and Knowledge Management. 2019.

Retrieve from: https://dl.acm.org/doi/abs/10.1145/3357384.3357925

> AutoInt can be applied to both numerical and categorical input features. Specifically, we map both the numerical and categorical features into the same low-dimensional space. Afterwards, a multihead self-attentive neural network with residual connections is proposed to explicitly model the feature interactions

in the lowdimensional space. With different layers of the multi-head selfattentive neural networks, different orders of feature combinations of input features can be modeled. The whole model can be efficiently fit on large-scale raw data in an end-to-end fashion.

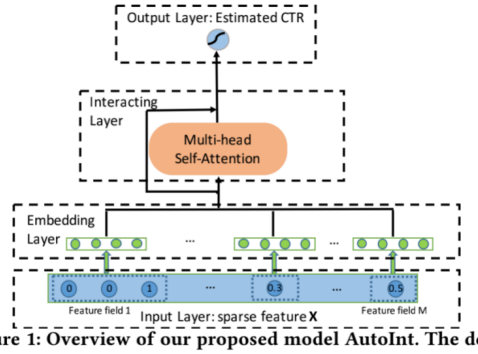


Figure 1: Overview of our proposed model AutoInt. The details of embedding layer and interacting layer are illustrated in Figure 2 and Figure 3 respectively.

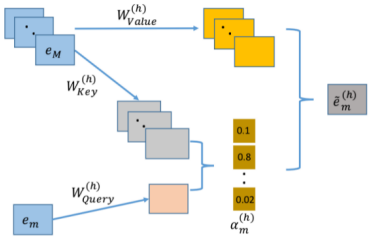


Figure 3: The architecture of interacting layer. Combinatorial features are conditioned on attention weights, i.e., $\alpha_m^{(h)}$.

4.4.8 FiBiNet

Huang, Tongwen, Zhiqi Zhang, and Junlin Zhang. "FiBiNET: combining feature importance and bilinear feature interaction for click-through rate prediction." Proceedings of the 13th ACM Conference on Recommender Systems. 2019.

Retrieve from: https://dl.acm.org/doi/abs/10.1145/3298689.3347043

FiBiNET as an abbreviation for 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.

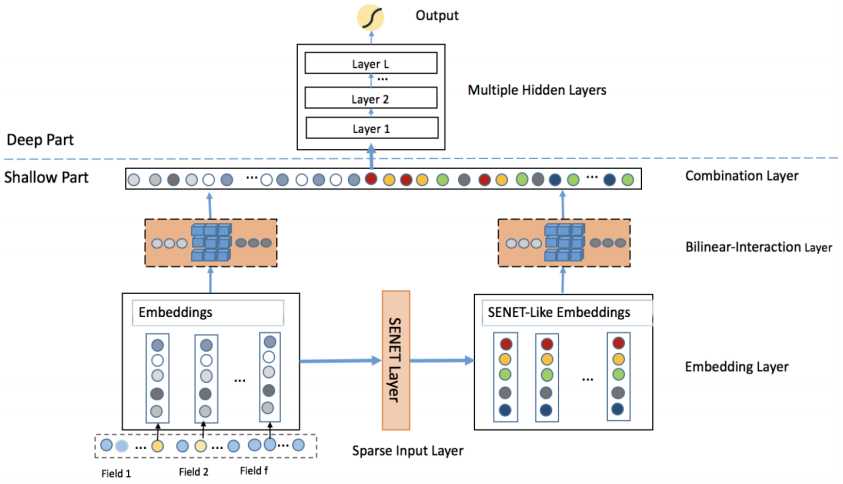


Figure 1: The architecture of our proposed FiBiNET

4.4.9 FGCNN

Liu, Bin, et al. "Feature generation by convolutional neural network for click-through rate prediction." The World Wide Web Conference. 2019.

Retrieve from: https://dl.acm.org/doi/abs/10.1145/3308558.3313497

> Feature Generation by Convolutional Neural Network (FGCNN) model 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. Experimental results on three large-scale datasets show that FGCNN significantly outperforms nine state-of-the-art models. Moreover, when applying some state-of-the-art models as Deep Classifier, better performance is always achieved, showing the great compatibility of our FGCNN model. This work explores a novel direction for CTR predictions: it is quite useful to reduce the learning difficulties of DNN by automatically identifying important features.

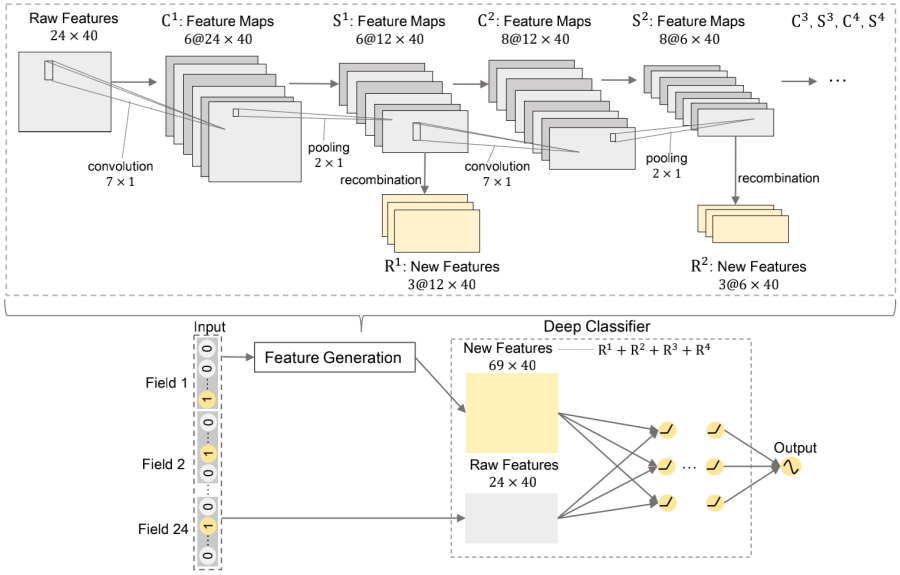


Figure 2: An overview of Feature Generation by Convolutional Neural Network Model (The hyper-parameters in the figure are the best setting of FGCNN on Avazu Dataset)

4.5 Layers

4.5.1 FM

Factorization Machine to model order-2 feature interactions.

Call arguments:

- x: A 3D tensor.

Input shape:

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

Output shape:

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

References:

- [1] Rendle S. Factorization machines[C]//2010 IEEE International Conference on Data Mining. IEEE, 2010: 995-1000.
- [2] Guo H, Tang R, Ye Y, et al. Deepfm: An end-to-end wide & deep learning framework for CTR prediction[J]. arXiv preprint arXiv:1804.04950, 2018.

4.5.2 AFM

Attentional Factorization Machine (AFM), which learns the importance of each feature interaction from data via a neural attention network.

Arguments:

- hidden_factor: int, (default=16)
- activation_function : str, (default='relu')
- kernel_regularizer : str or object, (default=None)
- dropout_rate: float, (default=0)

Call arguments:

- x: A list of 3D tensor.

Input shape:

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

Output shape:

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

References:

- [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.
- [2] https://github.com/hexiangnan/attentional_factorization_machine

4.5.3 CIN

Compressed Interaction Network (CIN), with the following considerations: (1) interactions are applied at vector-wise level, not at bit-wise level; (2) high-order feature interactions is measured explicitly; (3) the complexity of network will not grow exponentially with the degree of interactions.

Arguments:

- cross_layer_size: tuple of int, (default = (128, 128,))
- activation: str, (default='relu')
- use_residual: bool, (default=False)
- use_bias: bool, (default=False)
- direct: bool, (default=False)
- reduce_D:bool, (default=False)

Call arguments:

- x: A 3D tensor.

Input shape:

- A 3D tensor with shape: (batch_size, num_fields, embedding_size)

Output shape:

- 2D tensor with shape: (batch_size, *)

References:

- [1] Lian J, Zhou X, Zhang F, et al. xdeepfm: Combining explicit and implicit feature interactions for recommender systems[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018: 1754-1763.
- [2] https://github.com/Leavingseason/xDeepFM

4.5.4 MultiheadAttention

A multihead self-attentive nets with residual connections to explicitly model the feature interactions.

Arguments:

- num_head: int, (default=1)
- dropout_rate: float, (default=0)
- use_residual: bool, (default=True)

Call arguments:

- x: A 3D tensor.

Input shape:

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

Output shape:

- 3D tensor with shape: (batch_size, field_size, embedding_size*num_head)

References:

- [1] Song W, Shi C, Xiao Z, et al. Autoint: Automatic feature interaction learning via self-attentive neural networks[C]//Proceedings of the 28th ACM International Conference on Information and Knowledge Management. 2019: 1161-1170.
- [2] https://github.com/shichence/AutoInt

4.5.5 FGCNN

Feature Generation nets leverages the strength of CNN to generate local patterns and recombine them to generate new features.

Arguments:

- filters: int, the filters of convolutional layer
- kernel_height: int, the height of kernel_size of convolutional layer
- new_filters: int, the number of new features' map in recombination layer
- pool_height: int, the height of pool_size of pooling layer
- activation: str, (default='tanh')

Call arguments:

- x: A 4D tensor.

Input shape:

- 4D tensor with shape: (batch_size, field_size, embedding_size, 1)

Output shape:

- pooling_output - 4D tensor
- new_features - 3D tensor with shape: (batch_size, field_size*new_filters, embedding_size)

References:

- [1] Liu B, Tang R, Chen Y, et al. Feature generation by convolutional neural network for click-through rate prediction[C]//The World Wide Web Conference. 2019: 1119-1129.

4.5.6 SENET

SENET layer can dynamically increase the weights of important features and decrease the weights of uninformative features to let the model pay more attention to more important features.

Arguments:

- pooling_op: str, (default='mean'). Pooling methods to squeeze the original embedding E into a statistic vector Z
- reduction_ratio: float, (default=3). Hyper-parameter for dimensionality-reduction

Call arguments:

- x: A 3D tensor.

Input shape:

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

Output shape:

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

References:

- [1] Huang T, Zhang Z, Zhang J. FiBiNET: combining feature importance and bilinear feature interaction for click-through rate prediction[C]//Proceedings of the 13th ACM Conference on Recommender Systems. 2019: 169-177.

4.5.7 BilinearInteraction

The Bilinear-Interaction layer combines the inner product and Hadamard product to learn the feature interactions.

Arguments:

- bilinear_type: str, (default='field_interaction'). The type of bilinear functions
  - `field_interaction`
  - `field_all`
  - `field_each`

Call arguments:

- x: A 3D tensor.

Input shape:

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

Output shape:

- 3D tensor with shape: (batch_size, *, embedding_size)

References:

- [1] Huang T, Zhang Z, Zhang J. FiBiNET: combining feature importance and bilinear feature interaction for click-through rate prediction[C]//Proceedings of the 13th ACM Conference on Recommender Systems. 2019: 169-177.

4.5.8 Cross

The cross network is composed of cross layers to apply explicit feature crossing in an efficient way.

Arguments:

- num_cross_layer: int, (default=2). The number of cross layers

Call arguments:

- x: A 2D tensor.

Input shape:

- 2D tensor with shape: (batch_size, field_size)

Output shape:

- 2D tensor with shape: (batch_size, field_size)

References:

- [1] Wang R, Fu B, Fu G, et al. Deep & cross network for ad click predictions[M]//Proceedings of the ADKDD'17. 2017: 1-7.

4.5.9 InnerProduct

InnerProduct layer used in PNN

Call arguments:

- x: A list of 3D tensor.

Input shape:

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

Output shape:

- 2D tensor with shape: (batch_size, num_fields*(num_fields-1)/2)

References:

- [1] Qu Y, Cai H, Ren K, et al. Product-based neural networks for user response prediction[C]//2016 IEEE 16th International Conference on Data Mining (ICDM). IEEE, 2016: 1149-1154.
- [2] Qu Y, Fang B, Zhang W, et al. Product-based neural networks for user response prediction over multi-field categorical data[J]. ACM Transactions on Information Systems (TOIS), 2018, 37(1): 1-35.
- [3] https://github.com/Atomu2014/product-nets

4.5.10 OuterProduct

OuterProduct layer used in PNN

Arguments:

- outer_product_kernel_type: str, (default='mat'). The type of outer product kernel
  - `mat`
  - `vec`
  - `num` **Call arguments**:
- x: A list of 3D tensor.

Input shape:

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

Output shape:

- 2D tensor with shape: (batch_size, num_fields*(num_fields-1)/2)

References:

- [1] Qu Y, Cai H, Ren K, et al. Product-based neural networks for user response prediction[C]//2016 IEEE 16th International Conference on Data Mining (ICDM). IEEE, 2016: 1149-1154.
- [2] Qu Y, Fang B, Zhang W, et al. Product-based neural networks for user response prediction over multi-field categorical data[J]. ACM Transactions on Information Systems (TOIS), 2018, 37(1): 1-35.
- [3] https://github.com/Atomu2014/product-nets

4.6 deeptables

4.6.1 deeptables package

Subpackages

deeptables.datasets package

Submodules

deeptables.datasets.dsutils module

deeptables.datasets.dsutils.**load_adult**()

deeptables.datasets.dsutils.**load_bank**()

deeptables.datasets.dsutils.**load_boston**()

deeptables.datasets.dsutils.**load_glass_uci**()

deeptables.datasets.dsutils.**load_heart_disease_uci**()

Module contents

deeptables.eda package

Submodules

deeptables.eda.utils module

Module contents

deeptables.ensemble package

Module contents

deeptables.fe package

Submodules

deeptables.fe.dae module

class deeptables.fe.dae.**DAE**(*encoder_units=(500, 500)*, *feature_units=20*, *activation='relu'*, *kernel_initializer='glorot_uniform'*, *optimizer=<tensorflow.python.keras.optimizer_v2.adam.Adam object>*, *noise_rate=0*)

Bases: object

build_dae(*X*)

build_dae2(*X*)

fit(*X*, *batch_size=128*, *epochs=1000*)

fit_transform(*X*, *batch_size=128*, *epochs=1000*)

mix_generator(*x*, *batch_size*, *swaprate=0.15*, *shuffle=True*)

x_generator(*x*, *batch_size*, *shuffle=True*)

Module contents

deeptables.models package

Submodules

deeptables.models.config module

class deeptables.models.config.**ModelConfig**

Bases: *deeptables.models.config.ModelConfig*

first_metric_name

deeptables.models.deepmodel module

class deeptables.models.deepmodel.**DeepModel**(*task*, *num_classes*, *config*, *categorical_columns*, *continuous_columns*, *model_file=None*)

Bases: object

Class for neural network models

apply(*X*, *output_layers=[]*, *concat_outputs=False*, *batch_size=128*, *verbose=0*, *transformer=None*)

evaluate(*X_test*, *y_test*, *batch_size=256*, *verbose=0*)

fit(*X=None*, *y=None*, *batch_size=128*, *epochs=1*, *verbose=1*, *callbacks=None*, *validation_split=0.2*, *validation_data=None*, *shuffle=True*, *class_weight=None*, *sample_weight=None*, *initial_epoch=0*, *steps_per_epoch=None*, *validation_steps=None*, *validation_freq=1*, *max_queue_size=10*, *workers=1*, *use_multiprocessing=False*)

predict(*X*, *batch_size=128*, *verbose=0*)

release()

save(*filepath*)

class deeptables.models.deepmodel.**IgnoreCaseDict**(**args*, ***kwargs*)

Bases: collections.UserDict

class deeptables.models.deepmodel.**ModelDesc**

Bases: object

add_input(*name*, *num_columns*)

add_net(*name*, *input_shape*, *output_shape*)

nets_desc()

optimizer_info()

set_concat_embed_dense(*output_shape*)

set_dense(*dense_dropout*, *use_batchnormalization*)

set_embeddings(*input_dims*, *output_dims*, *embedding_dropout*)

set_output(*activation*, *output_shape*, *use_bias*)

deeptables.models.deepnets module

deeptables.models.deepnets.**afm_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

Attentional Factorization Machine (AFM), which learns the importance of each feature interaction from datasets via a neural attention network.

deeptables.models.deepnets.**autoint_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks.

deeptables.models.deepnets.**cin_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

Compressed Interaction Network (CIN), with the following considerations: (1) interactions are applied at vector-wise level, not at bit-wise level; (2) high-order feature interactions is measured explicitly; (3) the complexity of network will not grow exponentially with the degree of interactions.

deeptables.models.deepnets.**cross_dnn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

Cross nets -> DNN -> logit_out

deeptables.models.deepnets.**cross_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

The Cross networks is composed of cross layers to apply explicit feature crossing in an efficient way.

deeptables.models.deepnets.**custom_dnn_D_A_D_B**(*x*, *params*, *cellname='dnn_D_A_D_B'*)

deeptables.models.deepnets.**dcn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

Concat the outputs from Cross nets and DNN nets and feed into a standard logits layer

deeptables.models.deepnets.**deserialize**(*name*, *custom_objects=None*)

deeptables.models.deepnets.**dnn**(*x*, *params*, *cellname='dnn'*)

deeptables.models.deepnets.**dnn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

MLP (fully-connected feed-forward neural nets)

deeptables.models.deepnets.**fg_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

Feature Generation leverages the strength of CNN to generate local patterns and recombine them to generate new features.

References

for click-through rate prediction[C]//The World Wide Web Conference. 2019: 1119-1129.‘

deeptables.models.deepnets.**fgcnn_afm_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

FGCNN with AFM as deep classifier

deeptables.models.deepnets.**fgcnn_cin_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

FGCNN with CIN as deep classifier

deeptables.models.deepnets.**fgcnn_dnn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

FGCNN with DNN as deep classifier

deeptables.models.deepnets.**fgcnn_fm_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

FGCNN with FM as deep classifier

deeptables.models.deepnets.**fgcnn_ipnn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

FGCNN with IPNN as deep classifier

deeptables.models.deepnets.**fibi_dnn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

FiBiNet with DNN as deep classifier

deeptables.models.deepnets.**fibi_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)

The SENET layer can convert an embedding layer into the SENET-Like embedding features, which helps to boost feature discriminability. The following Bilinear-Interaction layer models second order feature interactions on the original embedding and the SENET-Like embedding respectively. Subsequently, these cross features are concatenated by a combination layer which merges the outputs of Bilinear-Interaction layer.

deeptables.models.deepnets.**fm_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)
FM models pairwise(order-2) feature interactions

deeptables.models.deepnets.**get**(*identifier*)
Returns function. :param identifier: Function or string

Returns

- Function corresponding to the input string or input function.

Return type Nets function denoted by input

For example: >>> nets.get('dnn_nets')

<function dnnlogit at 0x1222a3d90>

deeptables.models.deepnets.**get_nets**(*nets*)

deeptables.models.deepnets.**ipnn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)
Inner Product-based Neural Network InnerProduct+DNN

deeptables.models.deepnets.**linear**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)
Linear(order-1) interactions

deeptables.models.deepnets.**opnn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)
Outer Product-based Neural Network OuterProduct+DNN

deeptables.models.deepnets.**pnn_nets**(*embeddings*, *flatten_emb_layer*, *dense_layer*, *concat_emb_dense*, *config*, *model_desc*)
Concatenation of inner product and outer product + DNN

deeptables.models.deepnets.**register_nets**(*nets_fn*)

deeptables.models.deepnets.**serialize**(*nets_fn*)

deeptables.models.deeptable module

Training and inference for tabular datasets using neural nets.

class deeptables.models.deeptable.**DeepTable**(*config=None*, *preprocessor=None*)
Bases: object

DeepTables can be use to solve classification and regression prediction problems on tabular datasets. Easy to use and provide good performance out of box, no datasets preprocessing is required.

Parameters **config** (ModelConfig) –

name: str, (default='conf-1')

nets: list of str or callable object, (default=['dnn_nets'])

- DeepFM -> ['linear','dnn_nets','fm_nets']
- xDeepFM
- DCN
- PNN
- WideDeep
- AutoInt

- AFM
- FGCNN
- FibiNet

- 'dnn_nets'
- 'linear'
- 'cin_nets'
- 'fm_nets'
- 'afm_nets'
- 'opnn_nets'
- 'ipnn_nets'
- 'pnn_nets',
- 'cross_nets'
- 'cross_dnn_nets'
- 'dcn_nets',
- 'autoint_nets'
- 'fg_nets'
- 'fgcnn_cin_nets'
- 'fgcnn_fm_nets'
- 'fgcnn_ipnn_nets'
- 'fgcnn_dnn_nets'
- 'fibi_nets'
- 'fibi_dnn_nets'

>>>from deeptables.models import deepnets >>>#preset nets >>>conf = ModelConfig(nets=deepnets.DeepFM) >>>#list of names of nets >>>conf = ModelConfig(nets=['linear','dnn_nets','cin_nets','cross_nets']) >>>#mixed preset nets and names >>>conf = ModelConfig(nets=deepnets.WideDeep+['cin_nets']) >>>#mixed names and custom nets >>>def custom_net(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense, config, model_desc): >>> out = layers.Dense(10)(flatten_emb_layer) >>> return out >>>conf = ModelConfig(nets=['linear', custom_net])

categorical_columns: list of strings, (default='auto')

- **'auto'** get the columns of categorical type automatically. By default, the object, bool and category will be selected. if 'auto' the [auto_categorize] will no longer takes effect.
- **list of strings** e.g. ['x1','x2','x3','..']

exclude_columns: list of strings, (default=[])

pos_label: str or int, (default=None) The label of positive class, used only when task is binary.

metrics: list of string or callable object, (default=['accuracy']) List of metrics to be evaluated by the model during training and testing. Typically you will use *metrics=['accuracy']* or *metrics=['AUC']*. Every metric should be a built-in evaluation metric in tf.keras.metrics or a callable object like *r2(y_true, y_pred):...* . See also: https://tensorflow.google.cn/versions/r2.0/api_docs/python/tf/keras/metrics

auto_categorize: bool, (default=False)

cat_exponent: float, (default=0.5)

cat_remain_numeric: bool, (default=True)

auto_encode_label: bool, (default=True)

auto_imputation: bool, (default=True)

auto_discrete: bool, (default=False)

apply_gbm_features: bool, (default=False)

gbm_params: dict, (default={})

gbm_feature_type: str, (default=embedding)

- embedding
- dense

fixed_embedding_dim: bool, (default=True)

embeddings_output_dim: int, (default=4)

embeddings_initializer: str or object, (default='uniform') Initializer for the *embeddings* matrix.

embeddings_regularizer: str or object, (default=None) Regularizer function applied to the *embeddings* matrix.

dense_dropout: float, (default=0) between 0 and 1 Fraction of the dense input units to drop.

embedding_dropout: float, (default=0.3) between 0 and 1 Fraction of the embedding input units to drop.

stacking_op: str, (default='add')

- add
- concat

output_use_bias: bool, (default=True)

apply_class_weight: bool, (default=False)

optimizer: str or object, (default='auto')

- auto
- str
- object

loss: str or object, (default='auto')

dnn_params: dict, (default={'hidden_units': ((128, 0, False), (64, 0, False)), 'dnn_activation': 'relu'})

autoint_params:dict, (default={'num_attention': 3,'num_heads': 1,

'dropout_rate': 0,'use_residual': True})

fgcnn_params={'fg_filters': (14, 16), 'fg_widths': (7, 7), 'fg_pool_widths': (2, 2), 'fg_new_feat_filters': (2, 2), },

fibinet_params={ 'senet_pooling_op': 'mean', 'senet_reduction_ratio': 3, 'bilinear_type': 'field_interaction',

}, cross_params={

'num_cross_layer': 4,

}, pnn_params={

'outer_product_kernel_type': 'mat',

}, afm_params={

'attention_factor': 4, 'dropout_rate': 0

}, cin_params={

'cross_layer_size': (128, 128), 'activation': 'relu', 'use_residual': False, 'use_bias': False, 'direct': False, 'reduce_D': False,

},

home_dir: str, (default=None) The home directory for saving model-related files. Each time running *fit(...)* or *fit_cross_validation(...)*, a subdirectory with a timestamp will be created in this directory.

monitor_metric: str, (default=None)

earlystopping_patience: int, (default=1)

gpu_usage_strategy: str, (default='memory_growth')

- memory_growth
- None

distribute_strategy: tensorflow.python.distribute.distribute_lib.Strategy, (default=None)

-

task
Type of prediction problem, if 'config.task = None'(by default), it will be inferred base on the values of *y* when calling 'fit(...)' or 'fit_cross_validation(...)'. -'binary' : binary classification task -'multiclass' multiclass classfication task -'regression' regression task

Type str

num_classes
The number of classes, used only when task is multiclass.

Type int

pos_label
The label of positive class, used only when task is binary.

Type str or int

output_path
Path to directory used to save models. In addition, if a valid 'X_test' is passed into *fit_cross_validation(...)*, the prediction results of the test set will be saved in this path as well. The

path is a subdirectory with time-stamp created in the *home directory*. *home directory* is specified through *config.home_dir*, if *config.home_dir=None output_path* will be created in working directory.

Type str

preprocessor
Preprocessor is used to perform datasets preprocessing, such as categorization, label encoding, imputation, discretization, etc., before feeding into neural nets.

Type *AbstractPreprocessor* (default = DefaultPreprocessor)

nets
List of the network cells used to build the DeepModel

Type list(str)

monitor
The metric for monitoring the quality of model in early_stopping, if not specified, the first metric in [config.metrics] will be used. (e.g. log_loss/auc_val/accuracy_val...)

Type str

modelset
The models produced by *fit(...)* or *fit_cross_validation(...)*

Type *ModelSet*

best_model
A set of models will be produced by *fit_cross_validation(...)*, instead of only one model by *fit(...)*. The Best Model is the model with best performance on specific metric. The first metric in [config.metrics] will be used by default.

Type Model

leaderboard
List sorted by specific metric with some meta information and scores. The first metric in [config.metrics] will be used by default.

Type pandas.DataFrame

References

Examples

>>>X_train = pd.read_csv('https://storage.googleapis.com/tf-datasets/titanic/train.csv') >>>X_eval = pd.read_csv('https://storage.googleapis.com/tf-datasets/titanic/eval.csv') >>>y_train = X_train.pop('survived') >>>y_eval = X_eval.pop('survived') >>> >>>config = ModelConfig(nets=deepnets.DeepFM, fixed_embedding_dim=True, embeddings_output_dim=4, auto_discrete=True) >>>dt = DeepTable(config=config) >>> >>>model, history = dt.fit(train, y_train, epochs=100) >>>preds = dt.predict(X_eval)

apply(*X*, *output_layers*, *concat_outputs=False*, *batch_size=128*, *verbose=0*, *model_selector='current'*, *auto_transform_data=True*, *transformer=None*)

best_model

classes_

concat_emb_dense(*flatten_emb_layer*, *dense_layer*)

evaluate(*X_test*, *y_test*, *batch_size=256*, *verbose=0*, *model_selector='current'*)

fit(*X=None*, *y=None*, *batch_size=128*, *epochs=1*, *verbose=1*, *callbacks=None*, *validation_split=0.2*, *validation_data=None*, *shuffle=True*, *class_weight=None*, *sample_weight=None*, *initial_epoch=0*, *steps_per_epoch=None*, *validation_steps=None*, *validation_freq=1*, *max_queue_size=10*, *workers=1*, *use_multiprocessing=False*)

fit_cross_validation(*X*, *y*, *X_eval=None*, *X_test=None*, *num_folds=5*, *stratified=False*, *iterators=None*, *batch_size=None*, *epochs=1*, *verbose=1*, *callbacks=None*, *n_jobs=1*, *random_state=9527*, *shuffle=True*, *class_weight=None*, *sample_weight=None*, *initial_epoch=0*, *steps_per_epoch=None*, *validation_steps=None*, *validation_freq=1*, *max_queue_size=10*, *workers=1*, *use_multiprocessing=False*)

get_class_weight(*y*)

get_model(*model_selector='current'*)

leaderboard

static load(*filepath*)

load_deepmodel(*filepath*)

modelset

monitor

num_classes

pos_label

predict(*X*, *encode_to_label=True*, *batch_size=128*, *verbose=0*, *model_selector='current'*, *auto_transform_data=True*)

predict_proba(*X*, *batch_size=128*, *verbose=0*, *model_selector='current'*, *auto_transform_data=True*)

predict_proba_all(*X*, *batch_size=128*, *verbose=0*, *auto_transform_data=True*)

proba2predict(*proba*, *encode_to_label=True*)

restore_modelset(*filepath*)

save(*filepath*, *deepmodel_basename=None*)

task

deeptables.models.deeptable.**infer_task_type**(*y*)

deeptables.models.deeptable.**probe_evaluate**(*dt*, *X*, *y*, *X_test*, *y_test*, *layers*, *score_fn={}*)

deeptables.models.evaluation module

deeptables.models.evaluation.**calc_score**(*y_true*, *y_proba*, *y_preds*, *metrics*, *task*, *pos_label=1*)

deeptables.models.layers module

class deeptables.models.layers.**AFM**(*params*, ***kwargs*)

Bases: tensorflow.python.keras.engine.base_layer.Layer

Attentional Factorization Machine (AFM), which learns the importance of each feature interaction from datasets via a neural attention network.

Parameters

- **hidden_factor** – int, (default=16)
- **activation_function** – str, (default='relu')
- **kernel_regularizer** – str or object, (default=None)
- **dropout_rate** – float, (default=0)

Call arguments: x: A list of 3D tensor.

- A list of 3D tensor with shape: (batch_size, 1, embedding_size)
  - 2D tensor with shape:

  (batch_size, 1)

References

interactions via attention networks[J]. arXiv preprint arXiv:1708.04617, 2017.' .. [2] https://github.com/hexiangnan/attentional_factorization_machine

build(*input_shape*)
Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

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

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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 deeptables.models.layers.**BilinearInteraction**(*bilinear_type='field_interaction'*, ***kwargs*)

Bases: tensorflow.python.keras.engine.base_layer.Layer

The Bilinear-Interaction layer combines the inner product and Hadamard product to learn the feature interactions.

Parameters **bilinear_type** – str, (default='field_interaction') the type of bilinear functions - field_interaction - field_all - field_each

Call arguments: x: A 3D tensor.

- 3D tensor with shape:

  (batch_size, field_size, embedding_size)

- 3D tensor with shape:

  *(batch_size, *, embedding_size)*

References

interaction for click-through rate prediction[C]//Proceedings of the 13th ACM Conference on Recommender Systems. 2019: 169-177.‘

build(*input_shape*)

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*x*, ***kwargs*)

This is where the layer's logic lives.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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 deeptables.models.layers.**BinaryFocalLoss**(*gamma=2.0*, *alpha=0.25*, *reduction='auto'*, *name='focal_loss'*)

Bases: tensorflow.python.keras.losses.Loss

Binary form of focal loss. FL(p_t) = -alpha * (1 - p_t)**gamma * log(p_t) where p = sigmoid(x), p_t = p or 1 - p depending on if the label is 1 or 0, respectively.

Parameters

- **-- the same as weighing factor in balanced cross entropy** (*alpha*) –
- **-- focusing parameter for modulating factor** (*gamma*) –

Default value: gamma – 2.0 as mentioned in the paper alpha – 0.25 as mentioned in the paper

References

https://arxiv.org/pdf/1708.02002.pdf https://github.com/umbertogriffo/focal-loss-keras

Usage: model.compile(loss=[BinaryFocalLoss(alpha=.25, gamma=2)], metrics=["accuracy"], optimizer=adam)

call(*y_true*, *y_pred*)

Invokes the *Loss* instance.

Parameters

- **y_true** – Ground truth values. shape = *[batch_size, d0, .. dN]*, except sparse loss functions such as sparse categorical crossentropy where shape = *[batch_size, d0, .. dN-1]*
- **y_pred** – The predicted values. shape = *[batch_size, d0, .. dN]*

Returns Loss values with the shape *[batch_size, d0, .. dN-1]*.

get_config()

Returns the config dictionary for a *Loss* instance.

class deeptables.models.layers.**CIN**(*params*, ***kwargs*)

Bases: tensorflow.python.keras.engine.base_layer.Layer

Compressed Interaction Network (CIN), with the following considerations: (1) interactions are applied at vector-wise level, not at bit-wise level; (2) high-order feature interactions is measured explicitly; (3) the complexity of network will not grow exponentially with the degree of interactions.

Parameters

- **cross_layer_size** – tuple of int, (default = (128, 128,))
- **activation** – str, (default='relu')
- **use_residual** – bool, (default=False)
- **use_bias** – bool, (default=False)
- **direct** – bool, (default=False)
- **reduce_D** – bool, (default=False)

Call arguments: x: A 3D tensor.

- A 3D tensor with shape:

(batch_size, num_fields, embedding_size)

- 2D tensor with shape:

*(batch_size, *)*

References

for recommender systems[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018: 1754-1763.‘ .. [2] https://github.com/Leavingseason/xDeepFM

build(*input_shape*)

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*x*, ***kwargs*)

This is where the layer's logic lives.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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 deeptables.models.layers.**CategoricalFocalLoss**(*gamma=2.0*, *alpha=0.25*, *reduction='auto'*, *name='focal_loss'*)

Bases: tensorflow.python.keras.losses.Loss

Softmax version of focal loss.

m

FL = -alpha * (1 - p_o,c)^gamma * y_o,c * log(p_o,c) c=1

where m = number of classes, c = class and o = observation

Parameters

- **-- the same as weighing factor in balanced cross entropy** (*alpha*) –
- **-- focusing parameter for modulating factor** (*gamma*) –

Default value: gamma – 2.0 as mentioned in the paper alpha – 0.25 as mentioned in the paper

References

Official paper: https://arxiv.org/pdf/1708.02002.pdf https://github.com/umbertogriffo/focal-loss-keras

Usage: model.compile(loss=[CategoricalFocalLoss(alpha=.25, gamma=2)], metrics=["accuracy"], optimizer=adam)

call(*y_true*, *y_pred*)
Invokes the *Loss* instance.

Parameters

- **y_true** – Ground truth values. shape = *[batch_size, d0, .. dN]*, except sparse loss functions such as sparse categorical crossentropy where shape = *[batch_size, d0, .. dN-1]*
- **y_pred** – The predicted values. shape = *[batch_size, d0, .. dN]*

Returns Loss values with the shape *[batch_size, d0, .. dN-1]*.

get_config()
Returns the config dictionary for a *Loss* instance.

class deeptables.models.layers.**Cross**(*params*, ***kwargs*)
Bases: tensorflow.python.keras.engine.base_layer.Layer

The cross network is composed of cross layers to apply explicit feature crossing in an efficient way.

Parameters **num_cross_layer** – int, (default=2) the number of cross layers

Call arguments: x: A 2D tensor.

- 2D tensor with shape:

  (batch_size, field_size)

- 2D tensor with shape:

  (batch_size, field_size)

References

of the ADKDD'17. 2017: 1-7.'

build(*input*)
Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*x*, ***kwargs*)

This is where the layer's logic lives.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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 deeptables.models.layers.**FGCNN**(*filters*, *kernel_height*, *new_filters*, *pool_height*, *activation='tanh'*, ***kwargs*)

Bases: tensorflow.python.keras.engine.base_layer.Layer

Feature Generation nets leverages the strength of CNN to generate local patterns and recombine them to generate new features.

Arguments:

filters: int the filters of convolutional layer

kernel_height the height of kernel_size of convolutional layer

new_filters the number of new features' map in recombination layer

pool_height the height of pool_size of pooling layer

activation: str, (default='tanh')

Call arguments: x: A 4D tensor.

- 4D tensor with shape:

(batch_size, field_size, embedding_size, 1)

pooling_output - 4D tensor new_features - 3D tensor with shape:

*(batch_size, field_size*new_filters, embedding_size)*

for click-through rate prediction[C]//The World Wide Web Conference. 2019: 1119-1129.'

build(*input_shape*)

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*x*, ***kwargs*)

This is where the layer's logic lives.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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 deeptables.models.layers.**FM**(***kwargs*)

Bases: tensorflow.python.keras.engine.base_layer.Layer

Factorization Machine to model order-2 feature interactions Arguments:

Call arguments: x: A 3D tensor.

- 3D tensor with shape:

  (batch_size, field_size, embedding_size)

- 2D tensor with shape:

  (batch_size, 1)

References

call(*x*, ***kwargs*)

This is where the layer's logic lives.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

class deeptables.models.layers.**GHMCLoss**(*bins=10*, *momentum=0.75*)
Bases: object

calc(*input*, *target*, *mask=None*, *is_mask=False*)
Args: input [batch_num, class_num]:

The direct prediction of classification fc layer.

target [batch_num, class_num]: Binary target (0 or 1) for each sample each class. The value is -1 when the sample is ignored.

mask [batch_num, class_num]

get_acc_sum(*bins*)

get_edges(*bins*)

class deeptables.models.layers.**InnerProduct**(***kwargs*)
Bases: tensorflow.python.keras.engine.base_layer.Layer

Inner-Product layer

Arguments:

Call arguments: x: A list of 3D tensor.

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

  - 2D tensor with shape:

  (batch_size, num_fields(num_fields-1)/2)*

References

IEEE 16th International Conference on Data Mining (ICDM). IEEE, 2016: 1149-1154.‘ .. [2] *Qu Y, Fang B, Zhang W, et al. Product-based neural networks for user response prediction over multi-field categorical datasets[J]. ACM Transactions on Information Systems (TOIS), 2018, 37(1): 1-35.* .. [3] https://github.com/Atomu2014/product-nets

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

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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 deeptables.models.layers.**MultiColumnEmbedding**(*input_dims*, *output_dims*, *dropout_rate=0.0*, *embeddings_initializer='uniform'*, *embeddings_regularizer=None*, *activity_regularizer=None*, *embeddings_constraint=None*, *mask_zero=False*, ***kwargs*)

Bases: tensorflow.python.keras.engine.base_layer.Layer

This class is adapted from tensorflow's implementation of Embedding We modify the code to make it suitable for multiple variables from different column in one input.

build(*input_shape*)

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*inputs*)

This is where the layer's logic lives.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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

Computes an output mask tensor.

Parameters

- **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).

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 deeptables.models.layers.**MultiheadAttention**(*params*, ***kwargs*)
Bases: tensorflow.python.keras.engine.base_layer.Layer

A multihead self-attentive nets with residual connections to explicitly model the feature interactions.

Parameters

- **params** – dict
- **------** –
  - num_head: int, (default=1)
  - dropout_rate: float, (default=0)
  - use_residual: bool, (default=True)

Call arguments: x: A 3D tensor.

- 3D tensor with shape:

  (batch_size, field_size, embedding_size)

- 3D tensor with shape:

  *(batch_size, field_size, embedding_size*num_head)*

References

self-attentive neural networks[C]//Proceedings of the 28th ACM International Conference on Information and Knowledge Management. 2019: 1161-1170.‘ .. [2] https://github.com/shichence/AutoInt

build(*input_shape*)
Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

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

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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 deeptables.models.layers.**OuterProduct**(*params*, ***kwargs*)

Bases: tensorflow.python.keras.engine.base_layer.Layer

Outer-Product layer

Parameters **outer_product_kernel_type** – str, (default='mat') the type of outer product kernel - mat - vec - num

Call arguments: x: A list of 3D tensor.

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

  - 2D tensor with shape:

  (batch_size, num_fields(num_fields-1)/2)*

References

IEEE 16th International Conference on Data Mining (ICDM). IEEE, 2016: 1149-1154.‘ .. [2] *Qu Y, Fang B, Zhang W, et al. Product-based neural networks for user response prediction over multi-field categorical datasets[J]. ACM Transactions on Information Systems (TOIS), 2018, 37(1): 1-35.* .. [3] https://github.com/Atomu2014/product-nets

build(*input*)

Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

call(*x*, ***kwargs*)

This is where the layer's logic lives.

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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 deeptables.models.layers.**SENET**(*pooling_op='mean'*, *reduction_ratio=3*, ***kwargs*)
Bases: tensorflow.python.keras.engine.base_layer.Layer

SENET layer can dynamically increase the weights of important features and decrease the weights of uninformative features to let the model pay more attention to more important features.

Arguments:

pooling_op: str, (default='mean') pooling methods to squeeze the original embedding E into a statistic vector Z - mean - max

reduction_ratio: float, (default=3) hyper-parameter for dimensionality-reduction

Call arguments: x: A 3D tensor.

- 3D tensor with shape:

  (batch_size, field_size, embedding_size)

- 3D tensor with shape:

  (batch_size, field_size, embedding_size)

interaction for click-through rate prediction[C]//Proceedings of the 13th ACM Conference on Recommender Systems. 2019: 169-177.'

build(*input_shape*)
Creates the variables of the layer (optional, for subclass implementers).

This is a method that implementers of subclasses of *Layer* or *Model* can override if they need a state-creation step in-between layer instantiation and layer call.

This is typically used to create the weights of *Layer* subclasses.

Parameters **input_shape** – Instance of *TensorShape*, or list of instances of *TensorShape* if the layer expects a list of inputs (one instance per input).

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

Note here that *call()* method in *tf.keras* is little bit different from *keras* API. In *keras* API, you can pass support masking for layers as additional arguments. Whereas *tf.keras* has *compute_mask()* method to support masking.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments. Currently unused.

Returns A tensor or list/tuple of tensors.

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.

deeptables.models.layers.**register_custom_objects**(*objs_dict: dict*)

deeptables.models.metainfo module

class deeptables.models.metainfo.**CategoricalColumn**
Bases: *deeptables.models.metainfo.CategoricalColumn*

class deeptables.models.metainfo.**ContinuousColumn**
Bases: *deeptables.models.metainfo.ContinuousColumn*

deeptables.models.modelset module

class deeptables.models.modelset.**ModelInfo**(*type*, *name*, *model*, *score*, ***meta*)
Bases: object

dict_lower_keys(*dict*)

get_score(*metric_name*)

class deeptables.models.modelset.**ModelSet**(*metric='AUC'*, *best_mode='max'*)
Bases: object

best_model()

clear()

get_modelinfo(*name*)

get_modelinfos(*type=None*)

get_models(*type=None*)

leaderboard(*top=0*, *type=None*)

push(*modelinfo*)

top_n(*top=0*, *type=None*)

deeptables.models.preprocessor module

class deeptables.models.preprocessor.**AbstractPreprocessor**(*config: deeptables.models.config.ModelConfig*)
Bases: object

fit_transform(*X*, *y*, *copy_data=True*)

get_X_y_signature(*X*, *y*)

get_categorical_columns()

get_continuous_columns()

inverse_transform_y(*y_indicator*)

labels

static load(*filepath*)

pos_label

save(*filepath*)

signature

task

transform(*X*, *y*, *copy_data=True*)

transform_X(*X*, *copy_data=True*)

transform_y(*y*, *copy_data=True*)

class deeptables.models.preprocessor.**DefaultPreprocessor**(*config: deeptables.models.config.ModelConfig*, *cache_home=None*, *use_cache=False*)

Bases: *deeptables.models.preprocessor.AbstractPreprocessor*

clear_cache()

fit_transform(*X*, *y*, *copy_data=True*)

fit_transform_y(*y*)

get_categorical_columns()

get_continuous_columns()

get_transformed_X_y_from_cache(*sign*)

inverse_transform_y(*y_indicator*)

load_transformers_from_cache()

prepare_X(*X*)

reset()

save_transformed_X_y_to_cache(*sign*, *X*, *y*)

save_transformers_to_cache()

transform(*X*, *y*, *copy_data=True*)

transform_X(*X*, *copy_data=True*)

transform_y(*y*, *copy_data=True*)

Module contents

deeptables.preprocessing package

Submodules

deeptables.preprocessing.transformer module

class deeptables.preprocessing.transformer.**CategorizeEncoder**(*columns=None*, *remain_numeric=True*)

Bases: object

fit(*X*)

fit_transform(*X*)

transform(*X*)

class deeptables.preprocessing.transformer.**DataFrameWrapper**(*transform*, *columns=None*)

Bases: object

fit(*X*)

fit_transform(*X*)

transform(*X*)

class deeptables.preprocessing.transformer.**GaussRankScaler**

Bases: object

fit_transform(*X*)

class deeptables.preprocessing.transformer.**LgbmLeavesEncoder**(*cat_vars*, *cont_vars*, *task*, ***params*)

Bases: object

fit(*X*, *y*)

fit_transform(*X*, *y*)

transform(*X*)

class deeptables.preprocessing.transformer.**MultiKBinsDiscretizer**(*columns=None*, *bins=None*, *strategy='quantile'*)

Bases: object

fit(*X*)

fit_transform(*X*)

transform(*X*)

class deeptables.preprocessing.transformer.**MultiLabelEncoder**(*columns=None*)

Bases: object

fit(*X*)

fit_transform(*X*)

transform(*X*)

class deeptables.preprocessing.transformer.**PassThroughEstimator**

Bases: object

fit(*X*)

fit_transform(*X*)

transform(*X*)

class deeptables.preprocessing.transformer.**SafeLabelEncoder**

Bases: sklearn.preprocessing._label.LabelEncoder

transform(*y*)

Transform labels to normalized encoding.

Parameters **y** (*array-like of shape [n_samples]*) – Target values.

Returns **y**

Return type array-like of shape [n_samples]

deeptables.preprocessing.utils module

deeptables.preprocessing.utils.**target_encoding**(*train*, *target*, *test=None*, *feat_to_encode=None*, *smooth=0.2*, *random_state=9527*)

deeptables.preprocessing.utils.**target_rate_encodeing**(*feat_to_encode*, *target*, *df*, *mode='order'*)

Module contents

deeptables.utils package

Submodules

deeptables.utils.batch_trainer module

class deeptables.utils.batch_trainer.**BatchTrainer**(*data_train, target, data_test=None, test_as_eval=False, eval_size=0.2, validation_size=0.2, eval_metrics=[], dt_config=None, dt_nets=[['dnn_nets']], dt_epochs=5, dt_batch_size=128, seed=9527, pos_label=None, verbose=1, cross_validation=False, retain_single_model=False, num_folds=5, stratified=True, n_jobs=1, lightgbm_params={}, catboost_params={}*)

Bases: object

static bayes_search(*estimator*, *search_spaces*, *X*, *y*, *fit_params=None*, *scoring=None*, *n_jobs=1*, *cv=None*, *n_points=1*, *n_iter=50*, *refit=False*, *random_state=9527*, *verbose=0*, *deadline=60*)

ensemble_predict_proba(*models*, *X=None*, *y=None*, *submission=None*, *submission_target='target'*)

first_metric_name

static fit_cross_validation(*estimator_type*, *fit_fn*, *X*, *y*, *X_test=None*, *score_fn=<function roc_auc_score>*, *estimator_params={}*, *categorical_feature=None*, *task_type='binary'*, *num_folds=5*, *stratified=True*, *iterators=None*, *batch_size=None*, *preds_filepath=None*)

gbm_model_predict_proba(*dt*, *model*, *X*)

get_models(*models*)

static grid_search(*estimator*, *param_grid*, *X*, *y*, *fit_params=None*, *scoring=None*, *n_jobs=None*, *cv=None*, *refit=False*, *verbose=0*)

static hyperopt_search(*optimizer*, *X*, *y*, *fit_params*, *title*, *callbacks=None*)

probe_evaluate(*models*, *layers*, *score_fn={}*)

static randomized_search(*estimator*, *param_distributions*, *X*, *y*, *fit_params=None*, *scoring=None*, *n_jobs=None*, *cv=None*, *n_iter=10*, *refit=False*, *verbose=0*)

start(*models=['dt']*)

train_catboost(*config*)

train_dt(*model_set, config, nets=['dnn_nets']*)

train_lgbm(*config*)

train_model(*model_set*, *config*, *fit_fn*, *model_name*, ***params*)

deeptables.utils.batch_trainer.**catboost_fit**(*X*, *y*, *X_val*, *y_val*, *cat_vars*, *task*, *estimator_params*)

deeptables.utils.batch_trainer.**lgbm_fit**(*X*, *y*, *X_val*, *y_val*, *cat_vars*, *task*, *estimator_params*)

deeptables.utils.batch_trainer.**timer**(*title*)

deeptables.utils.consts module

DeepTables constants module.

deeptables.utils.dart_early_stopping module

deeptables.utils.dart_early_stopping.**dart_early_stopping**(*stopping_rounds*, *first_metric_only=False*, *verbose=True*)

Create a callback that activates early stopping.

Activates early stopping. The model will train until the validation score stops improving. Validation score needs to improve at least every `early_stopping_rounds` round(s) to continue training. Requires at least one validation datasets and one metric. If there's more than one, will check all of them. But the training datasets is ignored anyway. To check only the first metric set `first_metric_only` to True.

Parameters

- **stopping_rounds** (*int*) – The possible number of rounds without the trend occurrence.
- **first_metric_only** (*bool, optional (default=False)*) – Whether to use only the first metric for early stopping.
- **verbose** (*bool, optional (default=True)*) – Whether to print message with early stopping information.

Returns **callback** – The callback that activates early stopping.

Return type function

deeptables.utils.dt_logging module

DeepTables logging module.

deeptables.utils.dt_logging.**get_logger**(*logger_name=None*)

deeptables.utils.gpu module

deeptables.utils.gpu.**set_memory_growth**()

deeptables.utils.gpu.**set_memory_limit**(*limit*)

deeptables.utils.quicktest module

deeptables.utils.quicktest.**test**()

Module contents

Module contents

4.7 FAQ

4.7.1 How...

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

X