deeptab

Release 1.6.1

Anton Frederik Thielmann, Manish Kumar, Christoph Weisser, Ben

Nov 23, 2025

GETTING STARTED

deeptab is a Python library for tabular deep learning. It includes models that leverage the Mamba (State Space Model) architecture, as well as other popular models like TabTransformer, FTTransformer, TabM and tabular ResNets. Check out our paper `Mambular: A Sequential Model for Tabular Deep Learning`, available here. Also check out our paper introducing TabulaRNN and analyzing the efficiency of NLP inspired tabular models.

CHAPTER

ONE

QUICKSTART

Similar to any sklearn model, deeptab models can be fit as easy as this:

```
from deeptab.models import MambularClassifier
# Initialize and fit your model
model = MambularClassifier()

# X can be a dataframe or something that can be easily transformed into a pd.DataFrame␣
↪as a np.array
model.fit(X, y, max_epochs=150, lr=1e-04)
```

CHAPTER

TWO

INTRODUCTION

deeptab is a Python package that brings the power of advanced deep learning architectures to tabular data, offering a suite of models for regression, classification, and distributional regression tasks. Designed with ease of use in mind, deeptab models adhere to scikit-learn's `BaseEstimator` interface, making them highly compatible with the familiar scikit-learn ecosystem. This means you can fit, predict, and evaluate using deeptab models just as you would with any traditional scikit-learn model, but with the added performance and flexibility of deep learning.

CHAPTER

THREE

MODELS

Model	Description
`Mambular`	A sequential model using Mamba blocks specifically designed for various tabular data tasks introduced here.
`TabM`	Batch Ensembling for a MLP as introduced by Gorishniy et al.
`NODE`	Neural Oblivious Decision Ensembles as introduced by Popov et al.
`FTTransformer`	A model leveraging transformer encoders, as introduced by Gorishniy et al., for tabular data.
`MLP`	A classical Multi-Layer Perceptron (MLP) model for handling tabular data tasks.
`ResNet`	An adaptation of the ResNet architecture for tabular data applications.
`TabTransforme`	A transformer-based model for tabular data introduced by Huang et al., enhancing feature learning capabilities.
`MambaTab`	A tabular model using a Mamba-Block on a joint input representation described here . Not a sequential model.
`TabulaRNN`	A Recurrent Neural Network for Tabular data, introduced here.
`MambAttention`	A combination between Mamba and Transformers, also introduced here.
`NDTF`	A neural decision forest using soft decision trees. See Kontschieder et al. for inspiration.
`SAINT`	Improve neural networs via Row Attention and Contrastive Pre-Training, introduced here.

All models are available for `regression`, `classification` and distributional regression, denoted by `LSS`. Hence, they are available as e.g. `MambularRegressor`, `MambularClassifier` or `MambularLSS`

CHAPTER

FOUR

DOCUMENTATION

You can find the deeptab API documentation here.

CHAPTER

FIVE

INSTALLATION

Install deeptab using pip:

```
pip install deeptab
```

If you want to use the original mamba and mamba2 implementations, additionally install mamba-ssm via:

```
pip install mamba-ssm
```

Be careful to use the correct torch and cuda versions:

```
pip install torch==2.0.0+cu118 torchvision==0.15.0+cu118 torchaudio==2.0.0+cu118 -f␣
↪https://download.pytorch.org/whl/cu118/torch_stable.html
pip install mamba-ssm
```

CHAPTER

SIX

USAGE

deeptab simplifies data preprocessing with a range of tools designed for easy transformation of tabular data.

- **Ordinal & One-Hot Encoding**: Automatically transforms categorical data into numerical formats using continuous ordinal encoding or one-hot encoding. Includes options for transforming outputs to `float` for compatibility with downstream models.
- **Binning**: Discretizes numerical features into bins, with support for both fixed binning strategies and optimal binning derived from decision tree models.
- **MinMax**: Scales numerical data to a specific range, such as [-1, 1], using Min-Max scaling or similar techniques.
- **Standardization**: Centers and scales numerical features to have a mean of zero and unit variance for better compatibility with certain models.
- **Quantile Transformations**: Normalizes numerical data to follow a uniform or normal distribution, handling distributional shifts effectively.
- **Spline Transformations**: Captures nonlinearity in numerical features using spline-based transformations, ideal for complex relationships.
- **Piecewise Linear Encodings (PLE)**: Captures complex numerical patterns by applying piecewise linear encoding, suitable for data with periodic or nonlinear structures.
- **Polynomial Features**: Automatically generates polynomial and interaction terms for numerical features, enhancing the ability to capture higher-order relationships.
- **Box-Cox & Yeo-Johnson Transformations**: Performs power transformations to stabilize variance and normalize distributions.
- **Custom Binning**: Enables user-defined bin edges for precise discretization of numerical data.

Fitting a model in deeptab is as simple as it gets. All models in deeptab are sklearn BaseEstimators. Thus, the `fit` method is implemented for all of them. Additionally, this allows for using all other sklearn inherent methods such as their built in hyperparameter optimization tools.

```
from deeptab.models import MambularClassifier
# Initialize and fit your model
model = MambularClassifier(
    d_model=64,
    n_layers=4,
    numerical_preprocessing="ple",
    n_bins=50,
    d_conv=8
)
# X can be a dataframe or something that can be easily transformed into a pd.DataFrame␣
```

(continues on next page)

(continued from previous page)

```
↪as a np.array
model.fit(X, y, max_epochs=150, lr=1e-04)
```

Predictions are also easily obtained:

```
# simple predictions
preds = model.predict(X)

# Predict probabilities
preds = model.predict_proba(X)
```

Since all of the models are sklearn base estimators, you can use the built-in hyperparameter optimizatino from sklearn.

```
from sklearn.model_selection import RandomizedSearchCV

param_dist = {
    'd_model': randint(32, 128),
    'n_layers': randint(2, 10),
    'lr': uniform(1e-5, 1e-3)
}

random_search = RandomizedSearchCV(
    estimator=model,
    param_distributions=param_dist,
    n_iter=50,  # Number of parameter settings sampled
    cv=5,       # 5-fold cross-validation
    scoring='accuracy',  # Metric to optimize
    random_state=42
)

fit_params = {"max_epochs":5, "rebuild":False}

# Fit the model
random_search.fit(X, y, **fit_params)

# Best parameters and score
print("Best Parameters:", random_search.best_params_)
print("Best Score:", random_search.best_score_)
```

Note: that using this, you can also optimize the preprocessing. Just use the prefix `prepro__` when specifying the preprocessor arguments you want to optimize:

```
param_dist = {
    'd_model': randint(32, 128),
    'n_layers': randint(2, 10),
    'lr': uniform(1e-5, 1e-3),
    "prepro__numerical_preprocessing": ["ple", "standardization", "box-cox"]
}
```

Since we have early stopping integrated and return the best model with respect to the validation loss, setting max_epochs to a large number is sensible.

Or use the built-in bayesian hpo simply by running:

```
best_params = model.optimize_hparams(X, y)
```

This automatically sets the search space based on the default config from `deeptab.configs`. See the documentation for all params with regard to `optimize_hparams()`. However, the preprocessor arguments are fixed and cannot be optimized here.

MambularLSS allows you to model the full distribution of a response variable, not just its mean. This is crucial when understanding variability, skewness, or kurtosis is important. All deeptab models are available as distributional models.

- **Full Distribution Modeling**: Predicts the entire distribution, not just a single value, providing richer insights.
- **Customizable Distribution Types**: Supports various distributions (e.g., Gaussian, Poisson, Binomial) for different data types.
- **Location, Scale, Shape Parameters**: Predicts key distributional parameters for deeper insights.
- **Enhanced Predictive Uncertainty**: Offers more robust predictions by modeling the entire distribution.
- **normal**: For continuous data with a symmetric distribution.
- **poisson**: For count data within a fixed interval.
- **gamma**: For skewed continuous data, often used for waiting times.
- **beta**: For data bounded between 0 and 1, like proportions.
- **dirichlet**: For multivariate data with correlated components.
- **studentt**: For data with heavier tails, useful with small samples.
- **negativebinom**: For over-dispersed count data.
- **inversegamma**: Often used as a prior in Bayesian inference.
- **categorical**: For data with more than two categories.
- **Quantile**: For quantile regression using the pinball loss.

These distribution classes make MambularLSS versatile in modeling various data types and distributions.

To integrate distributional regression into your workflow with `MambularLSS`, start by initializing the model with your desired configuration, similar to other deeptab models:

```
from deeptab.models import MambularLSS

# Initialize the MambularLSS model
model = MambularLSS(
    dropout=0.2,
    d_model=64,
    n_layers=8,

)

# Fit the model to your data
model.fit(
    X,
    y,
    max_epochs=150,
    lr=1e-04,
    patience=10,
    family="normal" # define your distribution
    )
```

CHAPTER

SEVEN

IMPLEMENT YOUR OWN MODEL

deeptab allows users to easily integrate their custom models into the existing logic. This process is designed to be straightforward, making it simple to create a PyTorch model and define its forward pass. Instead of inheriting from `nn.Module`, you inherit from deeptab's `BaseModel`. Each deeptab model takes three main arguments: the number of classes (e.g., 1 for regression or 2 for binary classification), `cat_feature_info`, and `num_feature_info` for categorical and numerical feature information, respectively. Additionally, you can provide a config argument, which can either be a custom configuration or one of the provided default configs.

One of the key advantages of using deeptab is that the inputs to the forward passes are lists of tensors. While this might be unconventional, it is highly beneficial for models that treat different data types differently. For example, the TabTransformer model leverages this feature to handle categorical and numerical data separately, applying different transformations and processing steps to each type of data.

Here's how you can implement a custom model with deeptab:

1. **First, define your config:** The configuration class allows you to specify hyperparameters and other settings for your model. This can be done using a simple dataclass.

```
from dataclasses import dataclass

@dataclass
class MyConfig:
    lr: float = 1e-04
    lr_patience: int = 10
    weight_decay: float = 1e-06
    lr_factor: float = 0.1
```

2. **Second, define your model:** Define your custom model just as you would for an `nn.Module`. The main difference is that you will inherit from `BaseModel` and use the provided feature information to construct your layers. To integrate your model into the existing API, you only need to define the architecture and the forward pass.

```
from deeptab.base_models import BaseModel
from deeptab.utils.get_feature_dimensions import get_feature_dimensions
import torch
import torch.nn

class MyCustomModel(BaseModel):
    def __init__(
        self,
        cat_feature_info,
        num_feature_info,
        num_classes: int = 1,
        config=None,
```

(continues on next page)

(continued from previous page)

```
        **kwargs,
    ):
        super().__init__(**kwargs)
        self.save_hyperparameters(ignore=["cat_feature_info", "num_feature_info"])

        input_dim = get_feature_dimensions(num_feature_info, cat_feature_info)

        self.linear = nn.Linear(input_dim, num_classes)

    def forward(self, num_features, cat_features):
        x = num_features + cat_features
        x = torch.cat(x, dim=1)

        # Pass through linear layer
        output = self.linear(x)
        return output
```

3. **Leverage the deeptab API:** You can build a regression, classification, or distributional regression model that can leverage all of deeptab's built-in methods by using the following:

```
from deeptab.models import SklearnBaseRegressor

class MyRegressor(SklearnBaseRegressor):
    def __init__(self, **kwargs):
        super().__init__(model=MyCustomModel, config=MyConfig, **kwargs)
```

4. **Train and evaluate your model:** You can now fit, evaluate, and predict with your custom model just like with any other deeptab model. For classification or distributional regression, inherit from `SklearnBaseClassifier` or `SklearnBaseLSS` respectively.

```
regressor = MyRegressor(numerical_preprocessing="ple")
regressor.fit(X_train, y_train, max_epochs=50)
```

CHAPTER

EIGHT

CUSTOM TRAINING

If you prefer to setup custom training, preprocessing and evaluation, you can simply use the `deeptab.base_models`. Just be careful that all basemodels expect lists of features as inputs. More precisely as list for numerical features and a list for categorical features. A custom training loop, with random data could look like this.

```
import torch
import torch.nn as nn
import torch.optim as optim
from deeptab.base_models import Mambular
from deeptab.configs import DefaultMambularConfig

# Dummy data and configuration
cat_feature_info = {
    "cat1": {
        "preprocessing": "imputer -> continuous_ordinal",
        "dimension": 1,
        "categories": 4,
    }
}  # Example categorical feature information
num_feature_info = {
    "num1": {"preprocessing": "imputer -> scaler", "dimension": 1, "categories": None}
} # Example numerical feature information
num_classes = 1
config = DefaultMambularConfig()  # Use the desired configuration

# Initialize model, loss function, and optimizer
model = Mambular(cat_feature_info, num_feature_info, num_classes, config)
criterion = nn.MSELoss()  # Use MSE for regression; change as appropriate for your task
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Example training loop
for epoch in range(10):  # Number of epochs
    model.train()
    optimizer.zero_grad()

    # Dummy Data
    num_features = [torch.randn(32, 1) for _ in num_feature_info]
    cat_features = [torch.randint(0, 5, (32,)) for _ in cat_feature_info]
    labels = torch.randn(32, num_classes)

    # Forward pass
```

(continues on next page)

(continued from previous page)

```
    outputs = model(num_features, cat_features)
    loss = criterion(outputs, labels)

    # Backward pass and optimization
    loss.backward()
    optimizer.step()

    # Print loss for monitoring
    print(f"Epoch [{epoch+1}/10], Loss: {loss.item():.4f}")
```

CHAPTER

NINE

CITATION

If you find this project useful in your research, please consider cite:

```
@article{thielmann2024mambular,
  title={Mambular: A Sequential Model for Tabular Deep Learning},
  author={Thielmann, Anton Frederik and Kumar, Manish and Weisser, Christoph and Reuter,␣
↪Arik and S{\"a}fken, Benjamin and Samiee, Soheila},
  journal={arXiv preprint arXiv:2408.06291},
  year={2024}
}
```

If you use TabulaRNN please consider to cite:

```
@article{thielmann2024efficiency,
  title={On the Efficiency of NLP-Inspired Methods for Tabular Deep Learning},
  author={Thielmann, Anton Frederik and Samiee, Soheila},
  journal={arXiv preprint arXiv:2411.17207},
  year={2024}
}
```

CHAPTER

TEN

LICENSE

The entire codebase is under MIT license.

10.1 Installation

Please follow the steps below for installing `deeptab`.

10.1.1 Install from the source:

```
cd deeptab

poetry install
```

Note: Make sure you in the same directory where `pyproject.toml` file resides.

10.1.2 Installation from PyPi:

The package is available on PyPi. You can install it using the following command:

```
pip install -U deeptab
```

PyPi link: deeptab

10.2 Classification

This example demonstrates how use Classification module from the `deeptab` package.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from deeptab.models import MambularClassifier
# Set random seed for reproducibility
np.random.seed(0)
```

Let's generate some random data to use for classification.

```
# Number of samples
n_samples = 1000
n_features = 5
```

Generate random features

```
X = np.random.randn(n_samples, n_features)
coefficients = np.random.randn(n_features)
```

Generate target variable

```
y = np.dot(X, coefficients) + np.random.randn(n_samples)
## Convert y to multiclass by categorizing into quartiles
y = pd.qcut(y, 4, labels=False)
```

Create a DataFrame to store the data

```
data = pd.DataFrame(X, columns=[f"feature_{i}" for i in range(n_features)])
data["target"] = y
```

Split data into features and target variable

```
X = data.drop(columns=["target"])
y = data["target"].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Instantiate the classifier and fit the model on training data

```
classifier = MambularClassifier()

# Fit the model on training data
classifier.fit(X_train, y_train, max_epochs=10)

print(classifier.evaluate(X_test, y_test))
```

10.3 Regression

This example demonstrates how use Regression module from the `deeptab` package.

```
# Simulate data
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from deeptab.models import MambularRegressor

# Set random seed for reproducibility
np.random.seed(0)

```

(continues on next page)

(continued from previous page)

```
# Number of samples
n_samples = 1000
n_features = 5

# Generate random features
X = np.random.randn(n_samples, n_features)
coefficients = np.random.randn(n_features)

# Generate target variable
y = np.dot(X, coefficients) + np.random.randn(n_samples)

# Create a DataFrame to store the data
data = pd.DataFrame(X, columns=[f"feature_{i}" for i in range(n_features)])
data["target"] = y

# Split data into features and target variable
X = data.drop(columns=["target"])
y = np.array(data["target"])


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Instantiate the regressor
regressor = MambularRegressor()

# Fit the model on training data
regressor.fit(X_train, y_train, max_epochs=10)

print(regressor.evaluate(X_test, y_test))
```

10.4 Distributional

This example demonstrates how use Distributional from the `deeptab` package.

```
# Simulate data
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from deeptab.models import MambularLSS

# Set random seed for reproducibility
np.random.seed(0)

# Number of samples and features
n_samples = 1000
n_features = 5

# Generate random features
```

(continues on next page)

(continued from previous page)

```
X = np.random.randn(n_samples, n_features)
coefficients = np.random.randn(n_features)

# Generate target variable
y = np.dot(X, coefficients) + np.random.randn(n_samples)

# Create a DataFrame to store the generated data
data = pd.DataFrame(X, columns=[f"feature_{i}" for i in range(n_features)])
data["target"] = y

# Split data into features and target variable
X = data.drop(columns=["target"])
y = np.array(data["target"])


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Instantiate the regressor
regressor = MambularLSS()

# Fit the model on training data
regressor.fit(X_train, y_train, family="normal", max_epochs=10)

print(regressor.evaluate(X_test, y_test))
```

10.5 Models

This module provides classes for the Mambular models that adhere to scikit-learn's `BaseEstimator` interface.

10.5.1 Mambular

Modules	Description
`MambularClassifier`	Multi-class and binary classification tasks with a sequential Mambular Model.
`MambularRegressor`	Regression tasks with a sequential Mambular Model.
`MambularLSS`	Various statistical distribution families for different types of regression and classification tasks.

10.5.2 FTTransformer

Modules	Description
FTTransformerClassifie	FT transformer for classification tasks.
FTTransformerRegressor	FT transformer for regression tasks.
FTTransformerLSS	Various statistical distribution families for different types of regression and classification tasks.

10.5.3 MLP Models

Modules	Description
MLPClassifier	Multi-class and binary classification tasks.
MLPRegressor	MLP for regression tasks.
MLPLSS	Various statistical distribution families for different types of regression and classification tasks.

10.5.4 TabTransformer

Modules	Description
TabTransformerClassifier	TabTransformer for classification tasks.
TabTransformerRegressor	TabTransformer for regression tasks.
TabTransformerLSS	TabTransformer for distributional tasks.

10.5.5 ResNet

Modules	Description
ResNetClassifier	Multi-class and binary classification tasks using ResNet.
ResNetRegressor	Regression tasks using ResNet.
ResNetLSS	Distributional tasks using ResNet.

10.5.6 MambaTab

Modules	Description
MambaTabClassifier	Multi-class and binary classification tasks using MambaTab.
MambaTabRegressor	Regression tasks using MambaTab.
MambaTabLSS	Distributional tasks using MambaTab.

10.5.7 MambaAttention

Modules	Description
MambAttentionClassifi	Multi-class and binary classification tasks using a Combination between Mamba and Attention layers.
MambAttentionRegressc	Regression tasks using sing a Combination between Mamba and Attention layers.
MambAttentionLSS	Distributional tasks using sing a Combination between Mamba and Attention layers.

10.5.8 RNN Models Including LSTM and GRU

Modules	Description
TabulaRNNClassifier	Multi-class and binary classification tasks using a RNN.
TabulaRNNRegressor	Regression tasks using a RNN.
TabulaRNNLSS	Distributional tasks using a RNN.

10.5.9 TabM

Modules	Description
TabMClassifier	Multi-class and binary classification tasks using TabM - Batch Ensembling MLP.
TabMRegressor	Regression tasks using TabM - Batch Ensembling MLP.
TabMLSS	Distributional tasks using TabM - Batch Ensembling MLP.

10.5.10 NODE

Modules	Description
NODEClassifier	Multi-class and binary classification tasks using Neural Oblivious Decision Ensembles.
NODERegressor	Regression tasks using Neural Oblivious Decision Ensembles.
NODELSS	Distributional tasks using Neural Oblivious Decision Ensembles.

10.5.11 NDTF

Modules	Description
NDTFClassifier	Multi-class and binary classification tasks using a Neural Decision Forest.
NDTFRegressor	Regression tasks using a Neural Decision Forest
NDTFLSS	Distributional tasks using a Neural Decision Forest.

10.5.12 SAINT

Modules	Description
SAINTClassifier	Multi-class and binary classification tasks using SAINT.
SAINTRegressor	Regression tasks using SAINT.
SAINTLSS	Distributional tasks using SAINT.

10.5.13 Base Classes

Modules	Description
SklearnBaseClassifier	Base class for classification tasks.
SklearnBaseLSS	Base class for distributional tasks.
SklearnBaseRegressor	Base class for regression tasks.

mambular.models

10.6 BaseModels

This module provides foundational classes and architectures for Mambular models, including various neural network architectures tailored for tabular data.

Modules	Description
BaseModel	Abstract base class defining the core structure and initialization logic for Mambular models.
TaskModel	PyTorch Lightning module for managing model training, validation, and testing workflows.
Mambular	Flexible neural network model leveraging the Mamba architecture with configurable normalization techniques for tabular data.
MLP	Multi-layer perceptron (MLP) model designed for tabular tasks, initialized with a custom configuration.
ResNet	Deep residual network (ResNet) model optimized for structured/tabular datasets.
FTTransforme	Feature Tokenizer (FTTransformer) model for tabular tasks, incorporating advanced embedding and normalization techniques.
TabTransform	TabTransformer model leveraging attention mechanisms for tabular data processing.
NODE	Neural Oblivious Decision Ensembles (NODE) for tabular tasks, combining decision tree logic with deep learning.
TabM	TabM architecture designed for tabular data, implementing batch-ensembling MLP techniques.
NDTF	Neural Decision Tree Forest (NDTF) model for tabular tasks, blending decision tree concepts with neural networks.
TabulaRNN	Recurrent neural network (RNN) model, including LSTM and GRU architectures, tailored for sequential or time-series tabular data.
MambAttentic	Attention-based architecture for tabular tasks, combining feature importance weighting with advanced normalization techniques.
SAINT	SAINT model. Transformer based model using row and column attetion.

10.6.1 deeptab.base_models

class deeptab.base_models.**Mambular**(*feature_information*, *num_classes=1*, *config=DefaultMambularConfig(lr=0.0001, lr_patience=10, weight_decay=1e-06, lr_factor=0.1, use_embeddings=False, embedding_activation=torch.nn.Identity, embedding_type='linear', embedding_bias=False, layer_norm_after_embedding=False, d_model=64, plr_lite=False, n_frequencies=48, frequencies_init_scale=0.01, embedding_projection=True, batch_norm=False, layer_norm=False, layer_norm_eps=1e-05, activation=torch.nn.SiLU, cat_encoding='int', n_layers=4, d_conv=4, dilation=1, expand_factor=2, bias=False, dropout=0.0, dt_rank='auto', d_state=128, dt_scale=1.0, dt_init='random', dt_max=0.1, dt_min=0.0001, dt_init_floor=0.0001, norm='RMSNorm', conv_bias=False, AD_weight_decay=True, BC_layer_norm=False, shuffle_embeddings=False, head_layer_sizes=[], head_dropout=0.5, head_skip_layers=False, head_activation=torch.nn.SELU, head_use_batch_norm=False, pooling_method='avg', bidirectional=False, use_learnable_interaction=False, use_cls=False, use_pscan=False, mamba_version='mamba-torch')*, ***kwargs*)

A Mambular model for tabular data, integrating feature embeddings, Mamba transformations, and a configurable architecture for processing categorical and numerical features with pooling and normalization.

Parameters

- **cat_feature_info** (*dict*) – Dictionary containing information about categorical features, including their names and dimensions.
- **num_feature_info** (*dict*) – Dictionary containing information about numerical features, including their names and dimensions.
- **num_classes** (*int, optional*) – The number of output classes or target dimensions for regression, by default 1.
- **config** (*DefaultMambularConfig, optional*) – Configuration object with model hyperparameters such as dropout rates, head layer sizes, Mamba version, and other architectural configurations, by default DefaultMambularConfig().
- ****kwargs** (*dict*) – Additional keyword arguments for the BaseModel class.

pooling_method

Pooling method to aggregate features after the Mamba layer.

Type
str

shuffle_embeddings

Flag indicating if embeddings should be shuffled, as specified in the configuration.

Type
bool

embedding_layer

Layer for embedding categorical and numerical features.

Type
EmbeddingLayer

mamba

Mamba-based transformation layer based on the version specified in config.

Type
Mamba or MambaOriginal

norm_f

Normalization layer for the processed features.

Type
nn.Module

tabular_head

MLP layer to produce the final prediction based on the output of the Mamba layer.

Type
MLP

perm

Permutation tensor used for shuffling embeddings, if enabled.

Type
torch.Tensor, optional

forward(*num_features*, *cat_features*)

Perform a forward pass through the model, including embedding, Mamba transformation, pooling, and prediction steps.

forward(**data*)

Defines the forward pass of the model.

Parameters
data (*tuple*) – Input tuple of tensors of num_features, cat_features, embeddings.

Returns
The output predictions of the model.

Return type
Tensor

class deeptab.base_models.**MLP**(*feature_information*, *num_classes=1*, *config=DefaultMLPConfig(lr=0.0001, lr_patience=10, weight_decay=1e-06, lr_factor=0.1, use_embeddings=False, embedding_activation=torch.nn.Identity, embedding_type='linear', embedding_bias=False, layer_norm_after_embedding=False, d_model=32, plr_lite=False, n_frequencies=48, frequencies_init_scale=0.01, embedding_projection=True, batch_norm=False, layer_norm=False, layer_norm_eps=1e-05, activation=torch.nn.ReLU, cat_encoding='int', layer_sizes=[256, 128, 32], skip_layers=False, dropout=0.2, use_glu=False, skip_connections=False)*, ***kwargs*)

A multi-layer perceptron (MLP) model for tabular data processing, with options for embedding, normalization, skip connections, and customizable activation functions.

Parameters

- **cat_feature_info** (*dict*) – Dictionary containing information about categorical features, including their names and dimensions.
- **num_feature_info** (*dict*) – Dictionary containing information about numerical features, including their names and dimensions.

- **num_classes** (*int, optional*) – The number of output classes or target dimensions for regression, by default 1.
- **config** (*DefaultMLPConfig, optional*) – Configuration object with model hyperparameters such as layer sizes, dropout rates, activation functions, embedding settings, and normalization options, by default DefaultMLPConfig().
- ****kwargs** (*dict*) – Additional keyword arguments for the BaseModel class.

layer_sizes

List specifying the number of units in each layer of the MLP.

Type
list of int

cat_feature_info

Stores categorical feature information.

Type
dict

num_feature_info

Stores numerical feature information.

Type
dict

layers

List containing the layers of the MLP, including linear layers, normalization layers, and activations.

Type
nn.ModuleList

skip_connections

Flag indicating whether skip connections are enabled between layers.

Type
bool

use_glu

Flag indicating if gated linear units (GLU) should be used as the activation function.

Type
bool

activation

Activation function applied between layers.

Type
callable

use_embeddings

Flag indicating if embeddings should be used for categorical and numerical features.

Type
bool

embedding_layer

Embedding layer for features, used if `use_embeddings` is enabled.

Type
EmbeddingLayer, optional

norm_f

Normalization layer applied to the output of the first layer, if specified in the configuration.

Type
nn.Module, optional

forward(*num_features*, *cat_features*)

Perform a forward pass through the model, including embedding (if enabled), linear transformations, activation, normalization, and prediction steps.

forward(**data*)

Forward pass of the MLP model.

Parameters
data (*tuple*) – Input tuple of tensors of num_features, cat_features, embeddings.

Returns
Output tensor.

Return type
torch.Tensor

class deeptab.base_models.**ResNet**(*feature_information*, *num_classes=1*, *config=DefaultResNetConfig(lr=0.0001, lr_patience=10, weight_decay=1e-06, lr_factor=0.1, use_embeddings=False, embedding_activation=torch.nn.Identity, embedding_type='linear', embedding_bias=False, layer_norm_after_embedding=False, d_model=32, plr_lite=False, n_frequencies=48, frequencies_init_scale=0.01, embedding_projection=True, batch_norm=False, layer_norm=False, layer_norm_eps=1e-05, activation=torch.nn.SELU, cat_encoding='int', layer_sizes=[256, 128, 32], skip_layers=False, dropout=0.5, norm=False, use_glu=False, skip_connections=True, num_blocks=3, average_embeddings=True)*, ***kwargs*)

A ResNet model for tabular data, combining feature embeddings, residual blocks, and customizable architecture for processing categorical and numerical features.

Parameters

- **cat_feature_info** (*dict*) – Dictionary containing information about categorical features, including their names and dimensions.
- **num_feature_info** (*dict*) – Dictionary containing information about numerical features, including their names and dimensions.
- **num_classes** (*int, optional*) – The number of output classes or target dimensions for regression, by default 1.
- **config** (*DefaultResNetConfig, optional*) – Configuration object containing model hyperparameters such as layer sizes, number of residual blocks, dropout rates, activation functions, and normalization settings, by default DefaultResNetConfig().
- ****kwargs** (*dict*) – Additional keyword arguments for the BaseModel class.

layer_sizes

List specifying the number of units in each layer of the ResNet.

Type
list of int

`cat_feature_info`

Stores categorical feature information.

Type
dict

`num_feature_info`

Stores numerical feature information.

Type
dict

`activation`

Activation function used in the residual blocks.

Type
callable

`use_embeddings`

Flag indicating if embeddings should be used for categorical and numerical features.

Type
bool

`embedding_layer`

Embedding layer for features, used if `use_embeddings` is enabled.

Type
EmbeddingLayer, optional

`initial_layer`

Initial linear layer to project input features into the model's hidden dimension.

Type
nn.Linear

`blocks`

List of residual blocks to process the hidden representations.

Type
nn.ModuleList

`output_layer`

Output layer that produces the final prediction.

Type
nn.Linear

`forward`(*num_features*, *cat_features*)

Perform a forward pass through the model, including embedding (if enabled), residual blocks, and prediction steps.

`forward`(**data*)

Forward pass of the ResNet model.

Parameters
`data` (*tuple*) – Input tuple of tensors of num_features, cat_features, embeddings.

Returns
Output tensor.

Return type
torch.Tensor

class deeptab.base_models.**FTTransformer**(*feature_information*, *num_classes=1*, *config=DefaultFTTransformerConfig(lr=0.0001, lr_patience=10, weight_decay=1e-06, lr_factor=0.1, use_embeddings=False, embedding_activation=torch.nn.Identity, embedding_type='linear', embedding_bias=False, layer_norm_after_embedding=False, d_model=128, plr_lite=False, n_frequencies=48, frequencies_init_scale=0.01, embedding_projection=True, batch_norm=False, layer_norm=False, layer_norm_eps=1e-05, activation=torch.nn.SELU, cat_encoding='int', n_layers=4, n_heads=8, attn_dropout=0.2, ff_dropout=0.1, norm='LayerNorm', transformer_activation=torch.nn.Module, transformer_dim_feedforward=256, norm_first=False, bias=True, head_layer_sizes=[], head_dropout=0.5, head_skip_layers=False, head_activation=torch.nn.SELU, head_use_batch_norm=False, pooling_method='avg', use_cls=False)*, ***kwargs*)

A Feature Transformer model for tabular data with categorical and numerical features, using embedding, transformer encoding, and pooling to produce final predictions.

Parameters

- **cat_feature_info** (*dict*) – Dictionary containing information about categorical features, including their names and dimensions.
- **num_feature_info** (*dict*) – Dictionary containing information about numerical features, including their names and dimensions.
- **num_classes** (*int, optional*) – The number of output classes or target dimensions for regression, by default 1.
- **config** (*DefaultFTTransformerConfig, optional*) – Configuration object containing model hyperparameters such as dropout rates, hidden layer sizes, transformer settings, and other architectural configurations, by default DefaultFTTransformerConfig().
- ****kwargs** (*dict*) – Additional keyword arguments for the BaseModel class.

pooling_method

The pooling method to aggregate features after transformer encoding.

Type
str

cat_feature_info

Stores categorical feature information.

Type
dict

num_feature_info

Stores numerical feature information.

Type
dict

embedding_layer

Layer for embedding categorical and numerical features.

Type
EmbeddingLayer

norm_f

Normalization layer for the transformer output.

Type
nn.Module

encoder

Transformer encoder for sequential processing of embedded features.

Type
nn.TransformerEncoder

tabular_head

MLPhead layer to produce the final prediction based on the output of the transformer encoder.

Type
MLPhead

forward(*num_features*, *cat_features*)

Perform a forward pass through the model, including embedding, transformer encoding, pooling, and prediction steps.

forward(**data*)

Defines the forward pass of the model.

Parameters
data (*tuple*) – Input tuple of tensors of num_features, cat_features, embeddings.

Returns
The output predictions of the model.

Return type
Tensor

class deeptab.base_models.**TabTransformer**(*feature_information*, *num_classes=1*, *config=DefaultTabTransformerConfig(lr=0.0001, lr_patience=10, weight_decay=1e-06, lr_factor=0.1, use_embeddings=False, embedding_activation=torch.nn.Identity, embedding_type='linear', embedding_bias=False, layer_norm_after_embedding=False, d_model=128, plr_lite=False, n_frequencies=48, frequencies_init_scale=0.01, embedding_projection=True, batch_norm=False, layer_norm=False, layer_norm_eps=1e-05, activation=torch.nn.SELU, cat_encoding='int', n_layers=4, n_heads=8, attn_dropout=0.2, ff_dropout=0.1, norm='LayerNorm', transformer_activation=torch.nn.Module, transformer_dim_feedforward=512, norm_first=True, bias=True, head_layer_sizes=[], head_dropout=0.5, head_skip_layers=False, head_activation=torch.nn.SELU, head_use_batch_norm=False, pooling_method='avg')*, ***kwargs*)

A PyTorch model for tasks utilizing the Transformer architecture and various normalization techniques.

Parameters

- **cat_feature_info** (*dict*) – Dictionary containing information about categorical features.
- **num_feature_info** (*dict*) – Dictionary containing information about numerical features.
- **num_classes** (*int, optional*) – Number of output classes (default is 1).
- **config** (*DefaultFTTransformerConfig, optional*) – Configuration object containing default hyperparameters for the model (default is DefaultMambularConfig()).
- ****kwargs** (*dict*) – Additional keyword arguments.

lr

Learning rate.

Type
float

lr_patience

Patience for learning rate scheduler.

Type
int

weight_decay

Weight decay for optimizer.

Type
float

lr_factor

Factor by which the learning rate will be reduced.

Type
float

pooling_method

Method to pool the features.

Type
str

cat_feature_info

Dictionary containing information about categorical features.

Type
dict

num_feature_info

Dictionary containing information about numerical features.

Type
dict

embedding_activation

Activation function for embeddings.

Type
callable

encoder

stack of N encoder layers

Type
callable

norm_f

Normalization layer.

Type
nn.Module

num_embeddings

Module list for numerical feature embeddings.

Type
nn.ModuleList

cat_embeddings

Module list for categorical feature embeddings.

Type
nn.ModuleList

tabular_head

Multi-layer perceptron head for tabular data.

Type
MLPhead

cls_token

Class token parameter.

Type
nn.Parameter

embedding_norm

Layer normalization applied after embedding if specified.

Type
nn.Module, optional

forward(**data*)

Defines the forward pass of the model.

Parameters
ata (*tuple*) – Input tuple of tensors of num_features, cat_features, embeddings.

Returns
The output predictions of the model.

Return type
Tensor

class deeptab.base_models.**TabulaRNN**(*feature_information*, *num_classes=1*, *config=DefaultTabulaRNNConfig(lr=0.0001, lr_patience=10, weight_decay=1e-06, lr_factor=0.1, use_embeddings=False, embedding_activation=torch.nn.Identity, embedding_type='linear', embedding_bias=False, layer_norm_after_embedding=False, d_model=128, plr_lite=False, n_frequencies=48, frequencies_init_scale=0.01, embedding_projection=True, batch_norm=False, layer_norm=False, layer_norm_eps=1e-05, activation=torch.nn.SELU, cat_encoding='int', model_type='RNN', n_layers=4, rnn_dropout=0.2, norm='RMSNorm', residuals=False, head_layer_sizes=[], head_dropout=0.5, head_skip_layers=False, head_activation=torch.nn.SELU, head_use_batch_norm=False, pooling_method='avg', norm_first=False, bias=True, rnn_activation='relu', dim_feedforward=256, d_conv=4, dilation=1, conv_bias=True)*, ***kwargs*)

forward(**data*)

Defines the forward pass of the model.

Parameters

- **num_features** (*Tensor*) – Tensor containing the numerical features.
- **cat_features** (*Tensor*) – Tensor containing the categorical features.

Returns
The output predictions of the model.

Return type
Tensor

10.7 Data Utils

This module provides class for data preparation input data.

Modules	Description
MambularDataset	A class for loading and preprocessing the dataset.
MambularDataModule	A class for preparing the dataset for training and testing etc.

10.7.1 deeptab.data_utils

class deeptab.data_utils.**MambularDataset**(**args: Any*, ***kwargs: Any*)

Custom dataset for handling structured data with separate categorical and numerical features, tailored for both regression and classification tasks.

Parameters

- **Tensors)** (*num_features_list (list of)*) –
- **Tensors)** –
- **Tensors** (*embeddings_list (list of)*) –
- **optional)** (*A flag indicating if the dataset is for a regression task. Defaults to True.*) –

- **(Tensor** (*labels*) –
- **optional)** –
- **(bool** (*regression*) –
- **optional)** –

class deeptab.data_utils.**MambularDataModule**(**args: Any*, ***kwargs: Any*)

A PyTorch Lightning data module for managing training and validation data loaders in a structured way.

This class simplifies the process of batch-wise data loading for training and validation datasets during the training loop, and is particularly useful when working with PyTorch Lightning's training framework.

Parameters

- **preprocessor** – object An instance of your preprocessor class.
- **batch_size** – int Size of batches for the DataLoader.
- **shuffle** – bool Whether to shuffle the training data in the DataLoader.
- **X_val** – DataFrame or None, optional Validation features. If None, uses train-test split.
- **y_val** – array-like or None, optional Validation labels. If None, uses train-test split.
- **val_size** – float, optional Proportion of data to include in the validation split if `X_val` and `y_val` are None.
- **random_state** – int, optional Random seed for reproducibility in data splitting.
- **regression** – bool, optional Whether the problem is regression (True) or classification (False).

preprocess_data(*X_train*, *y_train*, *X_val=None*, *y_val=None*, *embeddings_train=None*, *embeddings_val=None*, *val_size=0.2*, *random_state=101*)

Preprocesses the training and validation data.

Parameters

- **X_train** (*DataFrame or array-like, shape (n_samples_train, n_features)*) – Training feature set.
- **y_train** (*array-like, shape (n_samples_train,)*) – Training target values.
- **embeddings_train** (*array-like or list of array-like, optional*) – Training embeddings if available.
- **X_val** (*DataFrame or array-like, shape (n_samples_val, n_features), optional*) – Validation feature set. If None, a validation set will be created from `X_train`.
- **y_val** (*array-like, shape (n_samples_val,), optional*) – Validation target values. If None, a validation set will be created from `y_train`.
- **embeddings_val** (*array-like or list of array-like, optional*) – Validation embeddings if available.
- **val_size** (*float, optional*) – Proportion of data to include in the validation split if `X_val` and `y_val` are None.
- **random_state** (*int, optional*) – Random seed for reproducibility in data splitting.

Return type
None

`setup`(*stage*)

Transform the data and create DataLoaders.

`test_dataloader`()

Returns the test dataloader.

Returns
DataLoader instance for the test dataset.

Return type
DataLoader

`train_dataloader`()

Returns the training dataloader.

Returns
DataLoader instance for the training dataset.

Return type
DataLoader

`val_dataloader`()

Returns the validation dataloader.

Returns
DataLoader instance for the validation dataset.

Return type
DataLoader

10.8 Configurations

This module provides default configurations for deeptab models. Each configuration is implemented as a dataclass, offering a structured way to define model-specific hyperparameters.

10.8.1 Mambular

Dataclass	Description
`DefaultMambularConfig`	Default configuration for the Mambular model.

10.8.2 FTTransformer

Dataclass	Description
`DefaultFTTransformerConfig`	Default configuration for the FTTransformer model.

10.8.3 ResNet

Dataclass	Description
`DefaultResNetConfig`	Default configuration for the ResNet model.

10.8.4 MLP

Dataclass	Description
`DefaultMLPConfig`	Default configuration for the MLP model.

10.8.5 TabTransformer

Dataclass	Description
`DefaultTabTransformerConfig`	Default configuration for the TabTransformer model.

10.8.6 MambaTab

Dataclass	Description
`DefaultMambaTabConfig`	Default configuration for the MambaTab model.

10.8.7 RNN

Dataclass	Description
`DefaultTabulaRNNConfig`	Default configuration for RNN models (LSTM, GRU).

10.8.8 MambAttention

Dataclass	Description
`DefaultMambAttentionConfig`	Default configuration for the MambAttention model.

10.8.9 NDTF

Dataclass	Description
`DefaultNDTFConfig`	Default configuration for the Neural Decision Tree Forest (NDTF) model.

10.8.10 NODE

Dataclass	Description
`DefaultNODEConfig`	Default configuration for the Neural Oblivious Decision Ensembles (NODE) model.

10.8.11 TabM

Dataclass	Description
`DefaultTabMConfig`	Default configuration for the TabM model (Batch-Ensembling MLP).

10.8.12 SAINT

Dataclass	Description
`DefaultSAINTConfig`	Default configuration for the SAINT model.

Configurations

10.9 Contribution Guidelines

Thank you for considering contributing to our Python package! We appreciate your time and effort in helping us improve our project. Please take a moment to review the following guidelines to ensure a smooth and efficient contribution process.

10.9.1 Code of Conduct

We kindly request all contributors to adhere to our Code of Conduct when participating in this project. It outlines our expectations for respectful and inclusive behavior within the community.

10.9.2 Setting Up Development Environment

Before you start contributing to the project, you need to set up your development environment. This will allow you to make changes to the codebase, run tests, and build the documentation locally. The project uses `poetry` for dependency management and packaging. Along with that, `ruff` is used for source code formatting and linting.

To set up the development environment for this Python package, follow these steps:

1. Clone the repository to your local machine using the command:

```
git clone https://github.com/OpenTabular/DeepTab

cd DeepTab
```

2. Install tools required for setting up development environment:

- Install `poetry` for dependency management and packaging. You can install it using the following command or refer to the official documentation for more information.

```
pip install poetry
```

- Install `just` command runner. You can install it using the following command or refer to the official documentation for more information.

`justfile` in the source directory is used to define and run common tasks like testing, building, and formatting the codebase.

3. In case you are able to successfully install `poetry` and `just`, you can run the following command to install the dependencies and set up the development environment:

```
# it will install the dependencies as defined in the pyproject.toml file
# it will also install the pre-commit hooks

just install
```

In case you are not able to install `just`, you can follow the below steps to set up the development environment:

```
cd DeepTab

poetry install

poetry run pre-commit install
```

If you need to update the documentation, please install the dependencies requried for documentation:

```
pip install -r docs/requirements_docs.txt
```

Note: You can also set up a virtual environment to isolate your development environment.

10.9.3 How to Contribute

1. Create a new branch from the `develop` branch for your contributions. Please use descriptive and concise branch names.
2. Make your desired changes or additions to the codebase.
3. Ensure that your code adheres to PEP8 coding style guidelines.
4. Write appropriate tests for your changes, ensuring that they pass.
    - `make test`
5. Update the documentation and examples, if necessary.
6. Build the html documentation and verify if it works as expected. We have used Sphinx for documentation, you could build the documents as follows:
    - `cd src/docs`
    - `make clean`
    - `make html`
7. Verify the html documents created under `docs/_build/html` directory. `index.html` file is the main file which contains link to all other files and doctree.
8. Commit your changes following the Conventional Commits specification (see below).
9. Submit a pull request from your branch to the development branch of the original repository.
10. Wait for the maintainers to review your pull request. Address any feedback or comments if required.
11. Once approved, your changes will be merged into the main codebase.

10.9.4 Release Workflow

This project uses automated semantic versioning and releases. Here's how releases work:

Automated Release Process

```
1. Make Changes → 2. Conventional Commit → 3. Merge to Master → 4. Automated Release
```

Step-by-Step:

1. **Development Phase**
    - Create feature branch from `develop`
    - Make your changes
    - Commit using conventional commits (e.g., `feat:`, `fix:`)
2. **Merge to Develop**
    - Create PR to `develop` branch
    - After review, merge to `develop`
    - ReadTheDocs dev documentation updates automatically
3. **Merge to Master** (Triggers Release)
    - Merge `develop` to `master`

- GitHub Actions semantic-release workflow runs automatically

4. **Automated Release (on Master)**
   - Analyzes conventional commits since last release
   - Determines version bump (major/minor/patch)
   - Updates version in `pyproject.toml` and `__version__.py`
   - Generates/updates `CHANGELOG.md`
   - Creates git tag (e.g., `v1.7.0`)
   - Builds package (`poetry build`)
   - Publishes to PyPI
   - Creates GitHub Release with notes

What Triggers a Release?

Commit Type	Version Bump	PyPI Release
`feat:`	Minor (1.x.0)	Yes
`fix:`	Patch (1.6.x)	Yes
`perf:`	Patch (1.6.x)	Yes
`feat!:` or `BREAKING CHANGE:`	Major (x.0.0)	Yes
`docs:`, `style:`, `refactor:`, `test:`, `chore:`, `ci:`	None	No

Example Scenarios

Scenario 1: Documentation Update (No Release)

```
git commit -m "docs: update API reference"
# Merge to master → No version bump, no PyPI release
```

Scenario 2: Bug Fix (Patch Release)

```
git commit -m "fix: resolve memory leak in dataloader"
# Merge to master → Version 1.6.1 → 1.6.2 → PyPI release
```

Scenario 3: New Feature (Minor Release)

```
git commit -m "feat(models): add TabNet architecture"
# Merge to master → Version 1.6.1 → 1.7.0 → PyPI release
```

Scenario 4: Breaking Change (Major Release)

```
git commit -m "feat!: remove Python 3.9 support

BREAKING CHANGE: Python 3.10 is now the minimum required version"
# Merge to master → Version 1.6.1 → 2.0.0 → PyPI release
```

Important Notes

- **Only master branch** triggers releases
- **Semantic-release is fully automated** - no manual version bumping needed
- **Never manually edit version numbers** in `pyproject.toml` or `deeptab/__version__.py` - they are automatically updated by semantic-release
- **PyPI token** is configured in GitHub repository secrets
- **Review commits carefully** before merging to master (they determine the version!)

Working with Updated Versions

Q: What happens to `develop` branch after a release?

After semantic-release completes on `master`, the version files are automatically updated. The `develop` branch syncs automatically:

Automatic Sync Flow:

```
┌─────────────────────────────────────────────────────────┐
│              Release & Sync Process                     │
└─────────────────────────────────────────────────────────┘

1. Merge develop → master
   │
   │
2. Semantic Release runs on master
   │
   ├──→ Version: 1.6.1 → 1.7.0
   ├──→ Update pyproject.toml
   ├──→ Update __version__.py
   ├──→ Update CHANGELOG.md
   └──→ Create tag v1.7.0
   │
   │
3. Auto-Sync Workflow triggers
   │
   ├──→ [No Conflicts]
   │    │
   │    ├──→ Merge master → develop automatically
   │    └──→ Develop updated within 60 seconds
   │
   └──→ [Conflicts Detected]
        │
        ├──→ Create PR: "chore: sync develop with master"
        ├──→ Notify maintainers
        └──→ Manual merge required (rare)
```

For Contributors:

Before starting new work, always pull the latest `develop`:

```
# Pull latest develop (already synced automatically)
git checkout develop
git pull origin develop

# Create your feature branch
git checkout -b feature/my-new-feature
```

Note: 95% of the time, `develop` syncs automatically. If you see a PR titled "sync develop with master", it means manual conflict resolution is needed (maintainers handle this).

Don't worry if the version seems "old" in your branch - semantic-release calculates the correct new version based on commits when merging to master.

10.9.5 Submitting Contributions

When submitting your contributions, please ensure the following:

- Include a clear and concise description of the changes made in your pull request.
- Reference any relevant issues or feature requests in the pull request description.
- Make sure your code follows the project's coding style and conventions.
- Include appropriate tests that cover your changes, ensuring they pass successfully.
- Update the documentation if necessary to reflect the changes made.
- Ensure that your pull request has a single, logical focus.

10.9.6 Issue Tracker

If you encounter any bugs, have feature requests, or need assistance, please visit our Issue Tracker. Make sure to search for existing issues before creating a new one.

10.9.7 License

By contributing to this project, you agree that your contributions will be licensed under the LICENSE of the project. Please note that the above guidelines are subject to change, and the project maintainers hold the right to reject or request modifications to any contributions. Thank you for your understanding and support in making this project better!

10.10 Build and release

The document outlines the steps to build and release the `deeptab` package. At this point, it is assumed that the development and testing of the package have been completed successfully.

10.10.1 1. Test documentation

It is expected from the contributor to update the documentation as an when required along side the change in source code. Please use the below process to test the documentation:

```
cd deeptab/docs/

make doctest
```

Fix any docstring related issue, then proceed with next steps.

10.10.2 2. Version update

The package version is mantained in `deeptab/__version__.py` and `pyproject.toml` file. Increment the version according to the changes such as patch, minor, major or all.

- The version number should be in the format `major.minor.patch`. For example, `1.0.1`.

Note: Don't forget to update the version in the `pyproject.toml` file as well.

10.10.3 3. Release

- Create a pull request from your `feature` branch to the `develop` branch.
- Once the pull request is approved and merged to develop. The maintainer will test the package and documentation. If everything is fine, the maintainer will proceed further to merge the changed to `master` and `release` branch.
- Ideally content of `master` and `release` branch should be same. The `release` branch is used to publish the package to PyPi while `master` branch is used to publish the documentation to readthedocs and can be accesseed at deeptab.readthedocs.io.

10.10.4 4. Publish package to PyPi

The package is published to PyPi using GitHub Actions. The workflow is triggered when a new tag is pushed to the repository. The workflow will build the package, upload it to PyPi.

10.10.5 5. GitHub Release

Create a new release on GitHub with the version number and release notes. The release notes should include the changes made in the release.

INDEX