DeepMTP Documentation

Dimitrios Iliadis

Aug 17, 2023
DeepMTP is a Python package implementing a flexible two-branch neural network architecture that makes it compatible with the majority of multi-target prediction problems.
1.1 Installation

1.1.1 Dependencies

DeepMTP works with Python 3.7 or later.

1.1.2 Installation

DeepMTP is available on PyPI and can be installed using pip:

```bash
# create and activate a conda environment
conda create -n DeepMTP_env python=3.8
conda activate DeepMTP_env

# if a gpu is available
pip install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu113

# if a gpu is NOT available
pip install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cpu

# install DeepMTP
pip install DeepMTP
```

Alternatively, you can install directly from the source code. Simply clone the Git repository of the project and run the following commands:

```bash
git clone https://github.com/diliadis/DeepMTP.git
cd DeepMTP
conda env create -f environment.yml
conda activate DeepMTP_env
```
1.2 Loading a dataset

1.2.1 Loading a built-in dataset

Loading one of the datasets offered natively by DeepMTP In the example above, the multi-label classification dataset is loaded one of the built-in functions offered by DeepMTP. More specifically the available functions are the following:

**load_process_MLC()**

The user can load the multi-label classification datasets available in the MULAN repository. The different datasets can be accessed by specifying a valid name in the dataset_name parameter.:

```python
data_mlc = load_process_MLC(dataset_name='bibtex', variant='undivided', features_type='dataframe')
train_mlc, val_mlc, test_mlc, data_info_mlc = data_process(data_mlc, validation_setting='B', verbose=True)
```

**load_process_MTR()**

The user can load the multivariate regression datasets available in the MULAN repository. The different datasets can be accessed by specifying a valid name in the dataset_name parameter.:

```python
mtr_data = load_process_MTR(dataset_name='enb', features_type='dataframe')
train_mtr, val_mtr, test_mtr, data_info_mtr = data_process(mtr_data, validation_setting='B', verbose=True)
```

**load_process_MTL()**

The user can load the multi-task learning dataset dog, a crowdsourcing dataset first introduced in Liu et al. More specifically, the dataset contains 800 images of dogs who have been partially labelled by 52 annotators with one of 5 possible breeds. To modify this multi-class problem to a binary problem, we modify the task so that the prediction involves the correct or incorrect labelling by the annotator. In a future version of the software another dataset of the same type will be added.:

```python
train_mtl, val_mtl, test_mtl, data_info_mtl = data_process(mtl_data, validation_setting='B', verbose=True)
```

**load_process_MC()**

The user can load the matrix completion dataset MovieLens 100K, a movie rating prediction dataset available by the the GroupLens lab that contains 100k ratings from 1000 users on 1700 movies. In a future version of the software larger versions of the movielens dataset will be added:

```python
dp_data = load_process_DP(dataset_name='ern')
train_dp, val_dp, test_dp, data_info_dp = data_process(dp_data, validation_setting='D', verbose=True)
```
load_process_DP()

The user can load dyadic prediction datasets available here. These are four different biological network datasets (ern, srn, dpie, dpii) which can be accessed by specifying one of the four keywords in the dataset_name parameter:

```python
mc_data = load_process_MC(dataset_name='ml-100k')
train_mc, val_mc, test_mc, data_info_mc = data_process(mc_data, validation_setting='A', verbose=True)
```

### 1.2.2 Loading a custom dataset

In the most abstract view of a multi-target prediction problem there are three at most datasets that can be needed. These include the interaction matrix, the instance features, and the target features. When accounting for a train, val, test split the total number raises to 9 possible data sources. To group this info and avoid passing 9 different parameters in the data_process function, the framework uses a single dictionary with 3 key-value pairs: `{'train':{}, 'val':{}, 'test':{}}`. The values should also be a dictionaries with 3 key-value pairs: `{'y':{}, 'X_instance':{}, 'X_target':{}}`. When combined the dictionary can have the following form: `{'train':{}, 'val':{}, 'test':{}}`:

```python
data = {
    'train': {
        'y': ,
        'X_instance': ,
        'X_target': ,
    },
    'val': {
        'y': ,
        'X_instance': ,
        'X_target': ,
    },
    'test': {
        'y': ,
        'X_instance': ,
        'X_target': ,
    },
}
```

### 1.3 Configuration options

#### 1.3.1 General configuration

In the code snippet above the function generate_config is shown without any specific parameters. In practice, the function offers many parameters that define multiple characteristics of the architecture of the two-branch neural network, aspects of training, validating, testing etc. The following section can be used as a cheatsheet for users, explaining the meaning and rationale of every parameter.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>training</td>
<td></td>
</tr>
<tr>
<td>num_epochs</td>
<td>The max number of epochs allowed for training</td>
</tr>
<tr>
<td>learning_rate</td>
<td>The learning rate used to determine the step size at each iteration of the optimization process</td>
</tr>
<tr>
<td>decay</td>
<td>The weight decay (L2 penalty) used by the Adam optimizer</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_mode</td>
<td>The device that is going to be used to actually train the neural network. The valid options are <code>cpu</code> if the user wants to train slowly or <code>cuda:id</code> if the user wants to train on the id gpu of the system</td>
</tr>
<tr>
<td>num_workers</td>
<td>The number of sub-processes to use for data loading. Larger values usually improve performance but after a point training speed will become worse</td>
</tr>
<tr>
<td>train_batchsize</td>
<td>The number of samples that comprise a batch from the training set</td>
</tr>
<tr>
<td>val_batchsize</td>
<td>The number of samples that comprise a batch from the validation and test sets</td>
</tr>
<tr>
<td>patience</td>
<td>The number of epochs that the network is allowed to continue training for while observing worse overall performance</td>
</tr>
<tr>
<td>delta</td>
<td>Minimum change in the monitored quantity to qualify as an improvement</td>
</tr>
<tr>
<td>return_results_per_target</td>
<td>Whether or not to return the performance for every target separately</td>
</tr>
<tr>
<td>evaluate_train</td>
<td>Whether or not to calculate performance metrics over the training set</td>
</tr>
<tr>
<td>evaluate_val</td>
<td>Whether or not to calculate performance metrics over the validation set</td>
</tr>
<tr>
<td>eval_every_n_epochs</td>
<td>The interval that indicates when the performance metrics are computed</td>
</tr>
<tr>
<td>use_early_stopping</td>
<td>Whether or not to use early stopping while training</td>
</tr>
<tr>
<td>metrics</td>
<td>The performance metrics that will be calculated. For classification tasks the available metrics are ['hamming_loss', 'auroc', 'f1_score', 'aupr', 'accuracy', 'recall', 'precision'] while for regression tasks the available metrics are ['RMSE', 'MSE', 'MAE', 'R2', 'RRMSE']</td>
</tr>
<tr>
<td>metrics_average</td>
<td>The averaging strategy that will be used to calculate the metric. The available options are ['macro', 'micro', 'instance']</td>
</tr>
<tr>
<td>metric_to_optimize_early_stopping</td>
<td>The metric that will be used for tracking by the early stopping routine. The value can be one of the available performance metrics.</td>
</tr>
<tr>
<td>metric_to_optimize_best_epoch_selection</td>
<td>The validation metric that will be used to determine the best configuration. The value can be one of the available performance metrics.</td>
</tr>
<tr>
<td>verbose</td>
<td>Whether or not to print useful in the terminal</td>
</tr>
<tr>
<td>use_tensorboard_logger</td>
<td>Whether or not to log results in files that Tensorboard can read and visualize</td>
</tr>
<tr>
<td>wandb_project_name</td>
<td>Defines the name of the wandb project that the results of an experiment will be logged</td>
</tr>
<tr>
<td>wandb_project_entity</td>
<td>Defines the user name of the wandb account</td>
</tr>
<tr>
<td>results_path</td>
<td>Defines the path the all relevant information will be saved to</td>
</tr>
<tr>
<td>experiment_name</td>
<td>Defines the name of the current experiment. This name will be used to local save and the wandb save</td>
</tr>
<tr>
<td>save_model</td>
<td>Whether or not to save the model of the epoch with the best validation performance</td>
</tr>
<tr>
<td>General architecture</td>
<td>Enables a specific version of the general neural network architecture. Available options are mlp for the mlp version, dot_product for the dot product version, kronecker: for the kronecker product version. Default value is dot_product</td>
</tr>
<tr>
<td>batch_norm</td>
<td>The option to use batch normalization between the fully connected layers in the two branches</td>
</tr>
<tr>
<td>dropout_rate</td>
<td>The amount of dropout used in the layers of the two branches</td>
</tr>
<tr>
<td>Instance branch architecture</td>
<td>The type of architecture that will be used in the instance branch. Currently, there are two available options, MLP: a basic fully connected feed-forward neural network is used, CONV a convolutional neural network is used</td>
</tr>
<tr>
<td>instance_branch_input_dim</td>
<td>The input dimension of the instance branch</td>
</tr>
<tr>
<td>instance_train_transforms</td>
<td>The Pytorch compatible transforms that can be used on the training samples. Useful when using images with convolutional architectures</td>
</tr>
</tbody>
</table>

continues on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>instance_inference_transforms</strong></td>
<td>The Pytorch compatible transforms that can be used on the validation and test samples. Useful when using images with convolutional architectures</td>
</tr>
<tr>
<td><strong>instance_branch_params</strong></td>
<td>A dictionary that holds all the hyperparameters needed to configure the architecture present in the instance branch. The include key-value pairs like the following:</td>
</tr>
<tr>
<td><strong>target_branch_architecture</strong></td>
<td>The type of architecture that will be used in the target branch. Currently, there are two available options, <strong>MLP</strong>: a basic fully connected feed-forward neural network is used, <strong>CONV</strong>: a convolutional neural network is used</td>
</tr>
<tr>
<td><strong>target_branch_input_dim</strong></td>
<td>The input dimension of the target branch</td>
</tr>
<tr>
<td><strong>target_train_transforms</strong></td>
<td>The Pytorch compatible transforms that can be used on the validation and test samples. Useful when using images with convolutional architectures</td>
</tr>
<tr>
<td><strong>target_inference_transforms</strong></td>
<td>The Pytorch compatible transforms that can be used on the validation and test samples. Useful when using images with convolutional architectures</td>
</tr>
<tr>
<td><strong>target_branch_params</strong></td>
<td>A dictionary that holds all the hyperparameters needed to configure the architecture present in the target branch.</td>
</tr>
<tr>
<td><strong>Combination branch architecture</strong></td>
<td></td>
</tr>
<tr>
<td><strong>comb_mlp_nodes_per_layer</strong></td>
<td>The number of nodes in the combination branch. If list, each element defines the number of nodes in the corresponding layer. If int, the same number of nodes is used ‘comb_mlp_layers’ times. (Only used if <strong>general_architecture_version</strong> == <strong>mlp</strong>)</td>
</tr>
<tr>
<td><strong>comb_mlp_layers</strong></td>
<td>The number of layers in the combination branch. (Only used if <strong>general_architecture_version</strong> == <strong>mlp</strong>)</td>
</tr>
<tr>
<td><strong>embedding_size</strong></td>
<td>The size of the embeddings outputted by the two branches. (Only used if <strong>general_architecture_version</strong> == <strong>dot_product</strong>)</td>
</tr>
<tr>
<td><strong>Pretrained models</strong></td>
<td></td>
</tr>
<tr>
<td><strong>load_pretrained_model</strong></td>
<td>Whether or not a pretrained model will be loaded</td>
</tr>
<tr>
<td><strong>pretrained_model_path</strong></td>
<td>The path to the .pt file with the pretrained model (Only used if <strong>load_pretrained_model</strong> == True)</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
</tr>
<tr>
<td><strong>additional_info</strong></td>
<td>A dictionary that holds all other relevant info. Can be used as log adittional info for an experiment in wandb</td>
</tr>
<tr>
<td><strong>validation_setting</strong></td>
<td>The validation setting of the specific example</td>
</tr>
</tbody>
</table>

---

### 1.3.2 Instance and target branch hyperparameters

As mentioned before, all hyperparameters needed to define the architecture of the instance or target branch are passed as key-value pairs in the **instance_branch_params** and **target_branch_params**.
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible key names currently supported in the instance_branch_params dictionary</td>
<td>Defines the number of nodes in the MLP version of the instance branch. if list, each element defines the number of nodes in the corresponding layer. If int, the same number of nodes is used instance_branch_layers times.</td>
</tr>
<tr>
<td>instance_branch_nodes_per_layer</td>
<td>The number of layers in the MLP version of the instance branch. (Only used if instance_branch_nodes_per_layer is int)</td>
</tr>
<tr>
<td>instance_branch_layers</td>
<td>The type of the convolutional architecture that is used in the instance branch.</td>
</tr>
<tr>
<td>instance_branch_conv_architecture</td>
<td>The version of the specific type of convolutional architecture that is used in the instance branch.</td>
</tr>
<tr>
<td>instance_branch_conv_architecture_version</td>
<td>The number of dense layers that are used at the end of the convolutional architecture of architecture_dense_layers.</td>
</tr>
<tr>
<td>instance_branch_conv_architecture_last_layer_trained</td>
<td>When using pre-trained architectures, the user can define that last layer that will be frozen during training.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Possible key names currently supported in the instance_branch_params dictionary</th>
<th>Defines the number of nodes in the MLP version of the target branch. if list, each element defines the number of nodes in the corresponding layer. If int, the same number of nodes is used target_branch_layers times.</th>
</tr>
</thead>
<tbody>
<tr>
<td>target_branch_nodes_per_layer</td>
<td>The number of layers in the MLP version of the target branch. (Only used if target_branch_nodes_per_layer is int)</td>
</tr>
<tr>
<td>target_branch_layers</td>
<td>The type of the convolutional architecture that is used in the target branch.</td>
</tr>
<tr>
<td>target_branch_conv_architecture</td>
<td>The version of the specific type of convolutional architecture that is used in the target branch.</td>
</tr>
<tr>
<td>target_branch_conv_architecture_version</td>
<td>The number of dense layers that are used at the end of the convolutional architecture of architecture_dense_layers.</td>
</tr>
<tr>
<td>target_branch_conv_architecture_last_layer_trained</td>
<td>When using pre-trained architectures, the user can define that last layer that will be frozen during training.</td>
</tr>
</tbody>
</table>

Example of generating a configuration:

```python
config = generate_config(
    instance_branch_input_dim = data_info['instance_branch_input_dim'],
    target_branch_input_dim = data_info['target_branch_input_dim'],
    validation_setting = data_info['detected_validation_setting'],
    general_architecture_version = 'dot_product',
    problem_mode = data_info['detected_problem_mode'],
    learning_rate = 0.001,
    decay = 0,
    batch_norm = False,
    dropout_rate = 0,
    momentum = 0.9,
    weighted_loss = False,
    compute_mode = 'cuda:0',
    train_batchsize = 1024,
    val_batchsize = 1024,
    num_epochs = 200,
)```

(continues on next page)
num_workers = 8,
metrics = ['RMSE', 'MSE'],
metrics_average = ['macro', 'micro'],
patience = 10,

evaluate_train = True,
evaluate_val = True,

verbose = False,
results_verbose = False,
use_early_stopping = True,
use_tensorboard_logger = True,

wandb_project_name = 'Dummy_project_1',

wandb_project_entity = None,

metric_to_optimize_early_stopping = 'loss',
delta=0.01,
metric_to_optimize_best_epoch_selection = 'loss',

instance_branch_architecture = 'MLP',
use_instance_features = True,

instance_branch_params = {

    'instance_branch_nodes_reducing_factor': 2,
    'instance_branch_nodes_per_layer': [100, 100],
    'instance_branch_layers': None,
    # 'instance_branch_conv_architecture': 'resnet',
    # 'instance_branch_conv_architecture_version': 'resnet101',
    # 'instance_branch_conv_architecture_dense_layers': 1,
    # 'instance_branch_conv_architecture_last_layer_trained': 'last',
},

target_branch_architecture = 'MLP',
use_target_features = True,

target_branch_params = {

    'target_branch_nodes_reducing_factor': 2,
    'target_branch_nodes_per_layer': [100, 100],
    'target_branch_layers': None,
    # 'target_branch_conv_architecture': 'resnet',
    # 'target_branch_conv_architecture_version': 'resnet101',
    # 'target_branch_conv_architecture_dense_layers': 1,
    # 'target_branch_conv_architecture_last_layer_trained': 'last',
},

eMBEDDING_SIZE = 100,
comb_mlp_nodes_reducing_factor = 2,
comb_mlp_nodes_per_layer = [2048, 2048, 2048],
comb_mlp_layers = None,

save_model = True,

eval_every_n_epochs = 1,

additional_info = {}
1.4 Hyperparameter Optimization

To further automate DeepMTP we decided to benchmark different popular hyperparameter optimization (HPO) methods (The resulting paper will be published in the near future). Based on the results, we concluded that Hyperband is a viable option for the majority of the MTP problem settings DeepMTP considers.

1.4.1 Hyperband

One of the core steps in any standard HPO method is the performance evaluation of a given configuration. This can be manageable for simple models that are relatively cheap to train and test, but can be a significant bottleneck for more complex models that need hours or even days to train. This is particularly evident in deep learning, as big neural networks with millions of parameters trained on increasingly larger datasets can deem traditional black-box HPO methods impractical.

Addressing this issue, multi-fidelity HPO methods have been devised to discard unpromising hyperparameter configurations already at an early stage. To this end, the evaluation procedure is adapted to support cheaper evaluations of hyperparameter configurations, such as evaluating on sub-samples (feature-wise or instance-wise) of the provided data set or executing the training procedure only for a certain number of epochs in the case of iterative learners. The more promising candidates are subsequently evaluated on increasing budgets until a maximum assignable budget is reached.

A popular representative of such methods is Hyperband. Hyperband builds upon Successive Halving (SH), where a set of \(n\) candidates is first evaluated on a small budget.

1.4.2 Combining Hyperband with DeepMTP

DeepMTP offers a basic Hyperband implementation natively, so the code needs only modification:

```python
from DeepMTP.dataset import load_process_MLC
from DeepMTP.main import DeepMTP
from DeepMTP.utils.utils import generate_config
from DeepMTP.simple_hyperband import BaseWorker
from DeepMTP.simple_hyperband import HyperBand
import ConfigSpace.hyperparameters as CSH

# define the configuration space
cs= CS.ConfigurationSpace()
# REALLY IMPORTANT: all hyperparameters for the instance or target branch should have...
# the 'instance_' or 'target_' prefix
lr= CSH.UniformFloatHyperparameter('learning_rate', lower=1e-6, upper=1e-3, default_value="1e-3", log=True)
embedding_size= CSH.UniformIntegerHyperparameter('embedding_size', lower=8, upper=2048, default_value=64, log=False)
instance_branch_layers= CSH.UniformIntegerHyperparameter('instance_branch_layers', lower=1, upper=2, default_value=1, log=False)
instance_branch_nodes_per_layer= CSH.UniformIntegerHyperparameter('instance_branch_nodes_per_layer', lower=8, upper=2048, default_value=64, log=False)
target_branch_layers = CSH.UniformIntegerHyperparameter('target_branch_layers', lower=1, upper=2, default_value=1, log=False)
target_branch_nodes_per_layer = CSH.UniformIntegerHyperparameter('target_branch_nodes_per_layer', lower=8, upper=2048, default_value=64, log=False)
dropout_rate = CSH.UniformFloatHyperparameter('dropout_rate', lower=0.0, upper=0.9, default_value=0.4, log=False)
batch_norm = CSH.CategoricalHyperparameter('batch_norm', [True, False])

cs.add_hyperparameters(
(continues on next page))```


```python
[ lr,
  embedding_size,
  instance_branch_layers,
  instance_branch_nodes_per_layer,
  target_branch_layers,
  target_branch_nodes_per_layer,
  dropout_rate,
  batch_norm,
]

# adding condition on the hyperparameter values
cond = CS.GreaterThanCondition(dropout_rate, instance_branch_layers, 1)
cond2 = CS.GreaterThanCondition(batch_norm, instance_branch_layers, 1)
cond3 = CS.GreaterThanCondition(dropout_rate, target_branch_layers, 1)
cond4 = CS.GreaterThanCondition(batch_norm, target_branch_layers, 1)
cs.add_condition(CS.OrConjunction(cond, cond3))
cs.add_condition(CS.OrConjunction(cond2, cond4))

# load dataset
data = load_process_MLC(dataset_name='yeast', variant='undivided', features_type=˓→'numpy')
# process and split
train, val, test, data_info = data_process(data, validation_setting='B', verbose=True)

# initialize the minimal configuration
config = {
  'hpo_results_path': './hyperband/',
  'instance_branch_input_dim': data_info['instance_branch_input_dim'],
  'target_branch_input_dim': data_info['target_branch_input_dim'],
  'validation_setting': data_info['detected_validation_setting'],
  'general_architecture_version': 'dot_product',
  'problem_mode': data_info['detected_problem_mode'],
  'compute_mode': 'cuda:1',
  'train_batchsize': 512,
  'val_batchsize': 512,
  'num_epochs': 6,
  'num_workers': 8,
  'metrics': ['hamming_loss', 'auroc'],
  'metrics_average': ['macro'],
  'patience': 10,
  'evaluate_train': True,
  'evaluate_val': True,
  'verbose': True,
  'results_verbose': False,
  'use_early_stopping': True,
  'use_tensorboard_logger': True,
  'wandb_project_name': 'DeepMTP_v2',
  'wandb_project_entity': 'diliadis',
  'metric_to_optimize_early_stopping': 'loss',
  'metric_to_optimize_best_epoch_selection': 'loss',
  'instance_branch_architecture': 'MLP',
  'target_branch_architecture': 'MLP',
  'save_model': True,
  'eval_every_n_epochs': 10,
}
```

(continues on next page)
additional_info': {'eta': 3, 'max_budget': 9}
}

# initialize the BaseWorker that will be used by Hyperband's optimizer
worker = BaseWorker(train, val, test, data_info, config, 'loss')
# initialize the optimizers
hb = HyperBand(
    base_worker=worker,
    configspace=cs,
    eta=config['additional_info']['eta'],
    max_budget=config['additional_info']['max_budget'],
    direction='min',
    verbose=True
)
# start-up the optimizer
best_overall_config = hb.run_optimizer()

# load the best model and generate predictions on the test set
best_model = DeepMTP(best_overall_config.info['config'], best_overall_config.info['model_dir'])
best_model_results = best_model.predict(test, verbose=True)

1.5 DEMOS

The following table contains links to several DEMO google colab notebooks.

<table>
<thead>
<tr>
<th>Name</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loading built-in datasets</td>
<td>link1</td>
</tr>
<tr>
<td>Multi-label classification (MLC)</td>
<td>link2</td>
</tr>
<tr>
<td>Multivariate regression (MTR)</td>
<td>link3</td>
</tr>
<tr>
<td>Multi-task learning (MTL)</td>
<td>link4</td>
</tr>
<tr>
<td>Matrix Completion (MC)</td>
<td>link5</td>
</tr>
<tr>
<td>Dyadic Prediction (DP)</td>
<td>link6</td>
</tr>
</tbody>
</table>

1.6 DeepMTP

1.6.1 DeepMTP package

Subpackages

DeepMTP.utils package

Submodules

DeepMTP.utils.data_utils module
DeepMTP.utils.eval_utils module
DeepMTP.utils.model_utils module
DeepMTP.utils.utils module

class DeepMTP.utils.utils.BaseExperimentInfo(config, budget)

Bases: object

A class used to keep track of all relevant info of a given experiment. This is mainly used by the HPO methods.

get_budget()

get_config()

update_score(score)

DeepMTP.utils.utils.generate_config(validation_setting=None,
    general_architecture_version='dot_product',
    problem_mode=None, learning_rate=0.001, decay=0,
    batch_norm=False, dropout_rate=0,
    dropout_rate_instance_branch=None,
    dropout_rate_target_branch=None,
    momentum=0.9,
    weighted_loss=False, compute_mode='cuda:0', num_workers=1,
    train_batchsize=512, val_batchsize=512, num_epochs=100,
    metrics=['hamming_loss', 'aucroc', 'f1_score', 'aupr', 'accuracy',
             'recall', 'precision'], metrics_average=['macro', 'micro'],
    top_k=None, patience=10, delta=0, evaluate_train=False,
    evaluate_val=False, verbose=False, results_verbose=False,
    eval_instance_verbose=False, eval_target_verbose=False,
    return_results_per_target=False, use_early_stopping=True,
    use_tensorboard_logger=False, wandb_project_name=None,
    wandb_project_entity=None, results_path='./results/',
    experiment_name=None, save_model=True,
    metric_to_optimize_best_epoch_selection='loss',
    metric_to_optimize_early_stopping='loss',
    instance_branch_architecture=None,
    use_instance_features=False, instance_branch_input_dim=None,
    instance_train_transforms=None,
    instance_inference_transforms=None,
    target_branch_architecture=None, use_target_features=False,
    target_branch_input_dim=None, target_train_transforms=None,
    target_inference_transforms=None,
    comb_mlp_nodes_reducing_factor=2,
    comb_mlp_nodes_per_layer=[10, 10, 10],
    comb_mlp_layers=None, embedding_size=100,
    eval_every_n_epochs=10, load_pretrained_model=False,
    pretrained_model_path='", running_hpo=False,
    additional_info={}, instance_branch_params={},
    target_branch_params={}, hpo_results_path="/")

Creates a dictionary that is used to configure the neural network. Contains some base logic that checks if some of the parameters make sense. It has to be updated each time a new feature is added.

Parameters

- **validation_setting** *(str, optional)* – The validation setting of the given problem. The possible values are A, B, C, D. Defaults to None.

- **problem_mode** *(str, optional)* – The type of task for the given problem. The possible values are classification or regression. Defaults to None.

- **learning_rate** *(float, optional)* – The learning rate that will be used during training. Defaults to 0.001.

- **decay** *(float, optional)* – The weight decay (L2 penalty) used by the Adam optimizer. Defaults to 0.

- **batch_norm** *(bool, optional)* – The option to use batch normalization between the fully connected layers in the two branches. Defaults to False.

- **dropout_rate** *(float, optional)* – The amount of dropout used in the layers of the two branches. Defaults to 0.

- **dropout_rate_instance_branch** *(float, optional)* – The amount of dropout used in the layers of the instance branch. Can be used when asymmetric overfitting between branches is observed. Defaults to 0.

- **dropout_rate_target_branch** *(float, optional)* – The amount of dropout used in the layers of the target branch. Can be used when asymmetric overfitting between branches is observed. Defaults to 0.

- **momentum** *(float, optional)* – The momentum used by the optimizer. Defaults to 0.9.

- **weighted_loss** *(bool, optional)* – Enables the use of class weights in the loss. Defaults to False.

- **compute_mode** *(str, optional)* – The specific device that will be used during training. The possible values can be one the available gpus or the cpu (please dont). Defaults to ‘cuda:0’.

- **num_workers** *(int, optional)* – The number of sub-processes to use for data loading. Larger values usually improve performance but after a point training speed will become worse. Defaults to 1.

- **train_batchsize** *(int, optional)* – The number of samples that comprise a batch from the training set. Defaults to 512.

- **val_batchsize** *(int, optional)* – The number of samples that comprise a batch from the validation and test sets. Defaults to 512.

- **num_epochs** *(int, optional)* – The max number of epochs allowed for training. Defaults to 100.

- **metrics** *(list, optional)* – The performance metrics that will be calculated. For classification tasks the available metrics are ['hamming_loss', 'auroc', 'f1_score', 'aupr', 'accuracy', 'recall', 'precision'] while for regression tasks the available metrics are ['RMSE', 'MSE', 'MAE', 'R2', 'RRMSE']. Defaults to ['hamming_loss', 'auroc', 'f1_score', 'aupr', 'accuracy', 'recall', 'precision'].

- **metrics_average** *(list, optional)* – The averaging strategy that will be used to calculate the metric. The available options are ['macro', 'micro', 'instance']. Defaults to ['macro', 'micro'].

- **top_k** *(float, optional)* – The number of top predictions used to calculate top_k versions of metrics (frequently used in collaborative filtering problems).

- **patience** *(int, optional)* – The number of epochs that the network is allowed to continue training for while observing worse overall performance. Defaults to 10.

- **delta** *(float, optional)* – The delta used during early stopping

- **evaluate_train** *(bool, optional)* – Whether or not to calculate performance metrics over the training set. Defaults to False.
• **evaluate_val** *(bool, optional)* – Whether or not to calculate performance metrics over the validation set. Defaults to False.

• **verbose** *(bool, optional)* – Whether or not to print useful info about the training process in the terminal. Defaults to False.

• **results_verbose** *(bool, optional)* – Whether or not to print useful info about the calculation of the performance metrics in the terminal. Defaults to False.

• **return_results_per_target** *(bool, optional)* – Whether or not to return metrics per target. Defaults to False.

• **use_early_stopping** *(bool, optional)* – Whether or not to use early stopping while training. Defaults to True.

• **use_tensorboard_logger** *(bool, optional)* – Whether or not to log results in Tensorboard. Defaults to False.

• **wandb_project_name** *(str, optional)* – The name of the wandb project that the results of an experiment will be logged. Defaults to None.

• **wandb_project_entity** *(str, optional)* – The user name of the wandb account. Defaults to None.

• **results_path** *(str, optional)* – The path the all relevant information will be saved to. Defaults to './results/'.

• **experiment_name** *(str, optional)* – The name of the current experiment. This name will be used to local save and the wandb save. Defaults to None.

• **save_model** *(bool, optional)* – Whether or not to save the model of the epoch with the best validation performance. Defaults to True.

• **metric_to_optimize_early_stopping** *(str, optional)* – The metric that will be used for tracking by the early stopping routine. The value can be the loss or one of the available performance metrics. Defaults to ‘loss’.

• **metric_to_optimize_best_epoch_selection** *(str, optional)* – The validation metric that will be used to determine the best configuration. The value can be the loss or one of the available performance metrics. Defaults to ‘loss’.

• **instance_branch_architecture** *(str, optional)* – The type of architecture that will be used in the instance branch. Currently, there are two available options, MLP: a basic fully connected feed-forward neural network is used, CONV: a convolutional neural network is used. Defaults to None.

• **use_instance_features** *(bool, optional)* – Whether or not the instance features will be used. Defaults to False.

• **instance_branch_input_dim** *(int, optional)* – The input dimension of the instance branch. Defaults to None.

• **instance_branch_nodes_reducing_factor** *(int, optional)* – The factor that will be used to create a smooth bottleneck in the instance branch. Not currently implemented. Defaults to 2.

• **instance_branch_nodes_per_layer** *(list, optional)* – Defines the number of nodes in the MLP version of the instance branch. If list, each element defines the number of nodes in the corresponding layer. If int, the same number of nodes is used instance_branch_layers times. Defaults to [10, 10, 10].

• **instance_branch_layers** *(int, optional)* – The number of layers in the MLP version of the instance branch. (Only used if instance_branch_nodes_per_layer is int). Defaults to None.

• **instance_train_transforms** *(str, optional)* – The Pytorch compatible transforms that can be used on the training samples. Useful when using images with convolutional architectures. Defaults to None.
• **instance_inference_transforms** (*type*, *optional*) – The Pytorch compatible transforms that can be used on the validation and test samples. Useful when using images with convolutional architectures. Defaults to None.

• **instance_branch_conv_architecture** (*str*, *optional*) – The type of the convolutional architecture that is used in the instance branch. Defaults to ‘resnet’.

• **instance_branch_conv_architecture_version** (*str*, *optional*) – The version of the specific type of convolutional architecture that is used in the instance branch. Defaults to ‘resnet101’.

• **instance_branch_conv_architecture_dense_layers** (*int*, *optional*) – The number of dense layers that are used at the end of the convolutional architecture of the instance branch. Defaults to 1.

• **instance_branch_conv_architecture_last_layer_trained** (*str*, *optional*) – When using pre-trained architectures, the user can define that last layer that will be frozen during training. Defaults to ‘last’.

• **target_branch_architecture** (*str*, *optional*) – The type of architecture that will be used in the target branch. Currently, there are two available options, MLP: a basic fully connected feed-forward neural network is used, CONV a convolutional neural network is used. Defaults to None.

• **use_target_features** (*bool*, *optional*) – Whether or not the target features will be used. Defaults to False. Defaults to False.

• **target_branch_input_dim** (*int*, *optional*) – The input dimension of the target branch. Defaults to None.

• **target_branch_nodes_reducing_factor** (*int*, *optional*) – The factor that will be used to create a smooth bottleneck in the target branch. Not currently implemented. Defaults to 2.

• **target_branch_nodes_per_layer** (*list, optional*) – Defines the number of nodes in the MLP version of the target branch. If list, each element defines the number of nodes in the corresponding layer. If int, the same number of nodes is used target_branch_layers times. Defaults to [10, 10, 10].

• **target_branch_layers** (*type*, *optional*) – The number of layers in the MLP version of the target branch. (Only used if target_branch_nodes_per_layer is int). Defaults to None.

• **target_train_transforms** (*type*, *optional*) – The Pytorch compatible transforms that can be used on the training samples. Useful when using images with convolutional architectures. Defaults to None.

• **target_inference_transforms** (*type*, *optional*) – The Pytorch compatible transforms that can be used on the validation and test samples. Useful when using images with convolutional architectures. Defaults to None.

• **target_branch_conv_architecture** (*str*, *optional*) – The type of the convolutional architecture that is used in the target branch. Defaults to ‘resnet’.

• **target_branch_conv_architecture_version** (*str*, *optional*) – The version of the specific type of convolutional architecture that is used in the target branch. Defaults to ‘resnet101’.

• **target_branch_conv_architecture_dense_layers** (*int*, *optional*) – The number of dense layers that are used at the end of the convolutional architecture of the target branch. Defaults to 1.

• **target_branch_conv_architecture_last_layer_trained** (*str*, *optional*) – When using pre-trained architectures, the user can define that last layer that will be frozen during training. Defaults to ‘last’.
• **comb_mlp_nodes_reducing_factor** (*int, optional*) – The factor that will be used to create a smooth bottleneck in the combination MLP. (Only used if `general_architecture_version` in “mlp”). Not currently implemented. Defaults to 2.

• **comb_mlp_nodes_per_layer** (*list, optional*) – The number of nodes in the combination branch. If list, each element defines the number of nodes in the corresponding layer. If int, the same number of nodes is used ‘comb_mlp_layers’ times. (Only used if `general_architecture_version` in “mlp” or “kronecker”). Defaults to [10, 10, 10].

• **comb_mlp_layers** (*int, optional*) – The number of layers in the combination branch. (Only used if `general_architecture_version` in “mlp”). Defaults to None.

• **embedding_size** (*int, optional*) – The size of the embeddings outputted by the two branches. (Only used if `general_architecture_version` in “dot_product”). Defaults to 100.

• **eval_every_n_epochs** (*int, optional*) – The interval that indicates when the performance metrics are computed. Defaults to 10.

• **load_pretrained_model** (*bool, optional*) – Whether or not a pretrained model will be loaded. Defaults to False.

• **pretrained_model_path** (*str, optional*) – The path to the .pt file with the pretrained model (Only used if `load_pretrained_model` == True). Defaults to ‘’.  

• **running_hpo** (*bool, optional*) – Whether or not the base model will be used by an hpo method. This is used to adjust the prints. Defaults to False.

• **additional_info** (*dict, optional*) – A dictionary that holds all other relevant info. Can be used as log additional info for an experiment in wandb. Defaults to {}.

• **eval_instance_verbose** (*str, optional*) – Printing of instance-wise warnings when evaluating performance metrics.

• **eval_target_verbose** (*str, optional*) – Printing of target-wise warnings when evaluating performance metrics.

• **hpo_results_path** (*str, optional*) – The directory the HPO results will be stored.

**Returns**
A dictionary with the config that will be used by the model to adjust the architecture and all other training-related information

**Return type**
`dict`

DeepMTP.utils.utils.get_default_batch_norm()

To return the default batch_norm value

**Returns**
The value False

**Return type**
`float`

DeepMTP.utils.utils.get_default_dropout_rate()

To return the default dropout rate

**Returns**
The value 0

**Return type**
`int`

DeepMTP.utils.utils.get_default_inference_transform()

To return the default transformation pipeline for a resnet during inference
DeepMTP Documentation

Returns
A transformation pipeline

Return type
torchvision.transforms

DeepMTP.utils.utils.get_default_train_transform()
To return the default transformation pipeline for a resnet during training

Returns
A transformation pipeline

Return type
torchvision.transforms

DeepMTP.utils.utils.get_optimization_direction(metric_name: str) → str
Determines if the goal is to maximize or minimize based on the name of the metric

Parameters
metric_name (string) – the name of the metric

Returns
Max if the goal is to maximize or min if the goal is to minimize

Return type
string

Module contents

Submodules

DeepMTP.branch_models module

class DeepMTP.branch_models.ConvNet(*args: Any, **kwargs: Any)
    Bases: Sequential
    A convolutional neural network that is based on resnet.
    forward(v)

class DeepMTP.branch_models.MLP(*args: Any, **kwargs: Any)
    Bases: Sequential
    A standard fully connected feed-forward neural network.
    forward(v)

DeepMTP.dataset module

DeepMTP.hpo_worker module

DeepMTP.main module

DeepMTP.main_streamlit module

DeepMTP.random_search module
class DeepMTP.random_search.RandomSearch(base_worker, configspace, budget=1, max_num_epochs=100, direction='min', verbose=False)

Bases: object

Implements the basic Random search HPO method. Nothing fancy, just a for loop over randomly generated configurations.

get_run_summary()

run_optimizer()

DeepMTP.random_search_streamlit module

class DeepMTP.random_search_streamlit.RandomSearch(base_worker, configspace, budget=1, max_num_epochs=100, direction='min', verbose=False)

Bases: object

Implements the basic Random search HPO method. Nothing fancy, just a for loop over randomly generated configurations.

get_norm_val(val, min_val, max_val)

get_run_summary()

run_optimizer()

DeepMTP.simple_hyperband module

class DeepMTP.simple_hyperband.HyperBand(base_worker, configspace, eta=3, max_budget=1, direction='min', verbose=False)

Bases: object

Implements a basic version of the Hyperband HPO method. One cool thing about it is that I reduced the training time by continuing to train later configurations instead of starting from scratch each time.

calculate_hyperband_iters(R, eta, verbose=False)

get_run_summary()

run_optimizer()

DeepMTP.simple_hyperband_streamlit module

class DeepMTP.simple_hyperband_streamlit.HyperBand(base_worker, configspace, eta=3, max_budget=1, direction='min', verbose=False)

Bases: object

Implements a basic version of the Hyperband HPO method. One cool thing about it is that I reduced the training time by continuing to train later configurations instead of starting from scratch each time.

calculate_hyperband_iters(R, eta, verbose=False)

get_norm_val(val, min_val, max_val)

get_run_summary()

run_optimizer()
DeepMTP Documentation

DeepMTP.tests module

Module contents

1.7 Credits

If you use DeepMTP in your work, please cite the following paper:

```latex
@article{iliadis2022multi,
  title={Multi-target prediction for dummies using two-branch neural networks},
  author={Iliadis, Dimitrios and De Baets, Bernard and Waegeman, Willem},
  journal={Machine Learning},
  volume={111},
  number={2},
  pages={651--684},
  year={2022},
  publisher={Springer}
}
```
CHAPTER
TWO

INDICES AND TABLES

• genindex
• modindex
• search
d
DeepMTP, 20
DeepMTP.branch_models, 18
DeepMTP.random_search, 18
DeepMTP.random_search_streamlit, 19
DeepMTP.simple_hyperband, 19
DeepMTP.simple_hyperband_streamlit, 19
DeepMTP.tests, 20
DeepMTP.utils, 18
DeepMTP.utils.utils, 13
INDEX

B
BaseExperimentInfo (class in DeepMTP.utils.utils), 13

calculate_hyperband_iters()
(DeepMTP.simple_hyperband.HyperBand method), 19
calculate_hyperband_iters()
(DeepMTP.simple_hyperband_streamlit.HyperBand method), 19
ConvNet (class in DeepMTP.branch_models), 18

D
DeepMTP
module, 20
DeepMTP.branch_models
module, 18
DeepMTP.random_search
module, 18
DeepMTP.random_search_streamlit
module, 19
DeepMTP.simple_hyperband
module, 19
DeepMTP.simple_hyperband_streamlit
module, 19
DeepMTP.tests
module, 20
DeepMTP.utils
module, 18
DeepMTP.utils.utils
module, 13

F
forward() (DeepMTP.branch_models.ConvNet method), 18
forward() (DeepMTP.branch_models.MLP method), 18

G
generate_config() (in module DeepMTP.utils.utils), 13
get_budget() (DeepMTP.utils.utils.BaseExperimentInfo method), 13
get_config() (DeepMTP.utils.utils.BaseExperimentInfo method), 13
get_default_batch_norm() (in module DeepMTP.utils.utils), 17
get_default_dropout_rate() (in module DeepMTP.utils.utils), 17
get_default_inference_transform() (in module DeepMTP.utils.utils), 17
get_default_train_transform() (in module DeepMTP.utils.utils), 17
get_norm_val() (DeepMTP.random_search_streamlit.RandomSearch method), 19
get_norm_val() (DeepMTP.simple_hyperband_streamlit.HyperBand method), 19
get_optimization_direction() (in module DeepMTP.utils.utils), 18
get_run_summary() (DeepMTP.random_search.RandomSearch method), 19
get_run_summary() (DeepMTP.random_search_streamlit.RandomSearch method), 19
get_run_summary() (DeepMTP.simple_hyperband.HyperBand method), 19
get_run_summary() (DeepMTP.simple_hyperband_streamlit.HyperBand method), 19

H
HyperBand (class in DeepMTP.simple_hyperband), 19
HyperBand (class in DeepMTP.simple_hyperband_streamlit), 19

M
MLP (class in DeepMTP.branch_models), 18
module
DeepMTP, 20
DeepMTP.branch_models, 18
DeepMTP.random_search, 18
DeepMTP.random_search_streamlit, 19
DeepMTP.simple_hyperband, 19
DeepMTP.simple_hyperband_streamlit, 19
DeepMTP.tests, 20
DeepMTP.utils, 18
DeepMTP.utils.utils, 13

R
RandomSearch (class in DeepMTP.random_search), 18
RandomSearch (class in DeepMTP.random_search_streamlit), 19
run_optimizer() (DeepMTP.random_search.RandomSearch method), 19
run_optimizer() (DeepMTP.random_search_streamlit.RandomSearch method), 19
run_optimizer() (DeepMTP.simple_hyperband.HyperBand method), 19
run_optimizer() (DeepMTP.simple_hyperband_streamlit.HyperBand method), 19

update_score() (DeepMTP.utils.utils.BaseExperimentInfo method), 13