PyTorch Wrapper

Release v1.0.4

Dec 08, 2019

Docs

1 Installation **3**
- 1.1 Via PyPI . . . 3
- 1.2 Via Git . . . 3

2 Examples **5**

3 System **7**

4 Modules **11**
- 4.1 Dynamic Self Attention Encoder . . . 11
- 4.2 Embedding Layer . . . 12
- 4.3 Layer Norm . . . 12
- 4.4 MLP . . . 13
- 4.5 Multi-Head Attention . . . 14
- 4.6 Residual . . . 15
- 4.7 Sequence Basic CNN Block . . . 16
- 4.8 Sequence Basic CNN Encoder . . . 17
- 4.9 Sequence Dense CNN . . . 18
- 4.10 Sinusoidal Positional Embedding Layer . . . 18
- 4.11 Softmax Attention Layer . . . 19
- 4.12 Softmax Self Attention Layer . . . 19
- 4.13 Transformer Encoder . . . 20
- 4.14 Transformer Encoder Block . . . 21

5 Functional **23**

6 Loss Wrappers **27**

7 Evaluators **29**

8 Samplers **39**

9 Training Callbacks **41**

10 Tuner **45**

Python Module Index **47**

Index 49

PyTorchWrapper is a library that provides a systematic and extensible way to build, train, evaluate, and tune deep learning models using PyTorch.

It also provides several ready to use modules and functions for fast model development.

GitHub Link

CHAPTER 1

Installation

1.1 Via PyPI

```
pip install pytorch-wrapper
```

1.2 Via Git

```
git clone https://github.com/jkoutsikakis/pytorch-wrapper.git
cd pytorch-wrapper
pip install .
```

CHAPTER 2

Examples

1. Two Spiral Task
2. Image Classification Task
3. Tuning Image Classifier
4. Text Classification Task
5. Sequence Classification Task
6. Custom Callback
7. Custom Loss Wrapper
8. Custom Evaluator

CHAPTER 3

System

class pytorch_wrapper.system.**System**(*model*, *last_activation=None*, *device=<sphinx.ext.autodoc.importer._MockObject object>*)

Bases: object

A system contains the usual methods needed for a deep learning model (train, evaluate, predict, save, load, etc).

Parameters

- **model** – An nn.Module object that represents the whole model. The module's forward method must return a Tensor or a Dict of Tensors.
- **last_activation** – Callable that needs to be called at non train time. Some losses work with logits and as such the last activation might not be performed inside the model's forward method. If the last activation is performed inside the model then pass None.
- **device** – Device on which the model should reside.

device

evaluate(*data_loader*, *evaluators*, *batch_input_key='input'*)

Evaluates the model on a dataset.

Parameters

- **data_loader** – DataLoader object that generates batches of the evaluation dataset. Each batch must be a Dict that contains the input of the model (key='batch_input_key') as well as the information needed by the evaluators.
- **evaluators** – Dictionary containing objects derived from AbstractEvaluator. The keys are the evaluators' names.
- **batch_input_key** – The key of the batches returned by the data_loader that contains the input of the model.

Returns Dict containing an object derived from AbstractEvaluatorResults for each evaluator.

static load(*f*)

Loads a System from a file. The model will reside in the CPU initially.

Parameters **f** – a file-like object (has to implement write and flush) or a string containing a file name.

load_model_state(*f*, *strict=True*)
Loads the model's state from a file.

Parameters

- **f** – a file-like object (has to implement write and flush) or a string containing a file name.
- **strict** – Whether the file must contain exactly the same weight keys as the model.

Returns NamedTuple with two lists (*missing_keys* and *unexpected_keys*).

predict(*data_loader*, *perform_last_activation=True*, *batch_id_key=None*, *batch_input_key='input'*, *model_output_key=None*)
Computes the outputs of the model on a dataset.

Parameters

- **data_loader** – DataLoader object that generates batches of data. Each batch must be a Dict that contains at least a Tensor or a list/tuple of Tensors containing the input(s) of the model(key='batch_input_key').
- **perform_last_activation** – Whether to perform the last_activation.
- **batch_id_key** – Key where the dict returned by the dataloader contains the ids of the examples. Leave None if there are no ids.
- **batch_input_key** – Key where the dict returned by the dataloader contains the input of the model.
- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.

Returns Dict containing a list of predictions (key='outputs') and a list of ids (key='batch_id_key') if provided by the dataloader.

predict_batch(*single_batch_input*)
Computes the output of the model for a single batch.

Parameters **single_batch_input** – Tensor or list of Tensors [tensor_1, tensor_2, ...] that correspond to the input of the model.

Returns The output of the model.

pure_predict(*data_loader*, *batch_input_key='input'*, *keep_batches=True*)
Computes the output of the model on a dataset.

Parameters

- **data_loader** – DataLoader object that generates batches of data. Each batch must be a Dict that contains at least a Tensor or a list/tuple of Tensors containing the input(s) of the model(key='batch_input_key').
- **batch_input_key** – The key of the batches returned by the data_loader that contains the input of the model.
- **keep_batches** – If set to True then the method also returns a list of the batches returned by the dataloader.

Returns Dict containing a list of batched model outputs (key='output_list') and a list of batches as returned by the dataloader (key='batch_list') if keep_batches is set to True.

save(*f*)
Saves the System to a file.

Parameters **f** – a file-like object (has to implement write and flush) or a string containing a file name.

save_model_state(*f*)
Saves the model's state to a file.

Parameters **f** – a file-like object (has to implement write and flush) or a string containing a file name.

to(*device*)
Transfers the model to the specified device.

Parameters **device** – Device to be transferred to.

Returns Returns the model after moving it to the device (inplace).

train(*loss_wrapper*, *optimizer*, *train_data_loader*, *evaluation_data_loaders=None*, *batch_input_key='input'*, *evaluators=None*, *callbacks=None*, *gradient_accumulation_steps=1*)
Trains the model on a dataset.

Parameters

- **loss_wrapper** – Object derived from AbstractLossWrapper that wraps the calculation of the loss.
- **optimizer** – Optimizer object.
- **train_data_loader** – DataLoader object that generates batches of the train dataset. Each batch must be a Dict that contains at least a Tensor or a list/tuple of Tensors containing the input(s) of the model (key='batch_input_key') as well as all the information needed by the loss_wrapper.
- **evaluation_data_loaders** – Dictionary containing the evaluation data-loaders. The keys are the datasets' names. Each batch generated by the dataloaders must be a Dict that contains the input of the model (key='batch_input_key') as well as the information needed by the evaluators.
- **batch_input_key** – Key of the Dicts returned by the Dataloader objects that corresponds to the input of the model.
- **evaluators** – Dictionary containing objects derived from AbstractEvaluator. The keys are the evaluators' names.
- **callbacks** – List containing TrainingCallback objects. They are used in order to inject functionality at several points of the training process. Default is NumberOfEpochsStoppingCriterionCallback(10) that stops training after the 10th iteration (counting from 0).
- **gradient_accumulation_steps** – Number of backward calls before an optimization step. Used in order to simulate a larger batch size).

Returns List containing the results for each epoch.

train_on_multi_gpus(*loss_wrapper*, *optimizer*, *train_data_loader*, *evaluation_data_loaders=None*, *batch_input_key='input'*, *evaluators=None*, *callbacks=None*, *gradient_accumulation_steps=1*, *multi_gpu_device_ids=None*, *multi_gpu_output_device=None*, *multi_gpu_dim=0*)
Trains the model on a dataset using multiple GPUs. At the end of training the model is moved back to the device it was on at the beginning.

Parameters

- **loss_wrapper** – Object derived from AbstractLossWrapper that wraps the calculation of the loss.

- **optimizer** – Optimizer object.
- **train_data_loader** – DataLoader object that generates batches of the train dataset. Each batch must be a Dict that contains at least a Tensor or a list/tuple of Tensors containing the input(s) of the model (key='batch_input_key').
- **evaluation_data_loaders** – Dictionary containing the evaluation data-loaders. The keys are the datasets' names. Each batch generated by the dataloaders must be a Dict that contains the input of the model (key='batch_input_key') as well as the information needed by the evaluators.
- **batch_input_key** – Key of the Dicts returned by the Dataloader objects that corresponds to the input of the model.
- **evaluators** – Dictionary containing objects derived from AbstractEvaluator. The keys are the evaluators' names.
- **callbacks** – List containing TrainingCallback objects. They are used in order to inject functionality at several points of the training process. Default is NumberOfEpochsStoppingCriterionCallback(10) that stops training after the 10th iteration (counting from 0).
- **gradient_accumulation_steps** – Number of backward calls before an optimization step. Used in order to simulate a larger batch size).
- **multi_gpu_device_ids** – CUDA devices used during training (default: all devices).
- **multi_gpu_output_device** – Device location of output (default: device_ids[0]).
- **multi_gpu_dim** – Int dimension on which to split each batch.

Returns List containing the results for each epoch.

CHAPTER 4

Modules

4.1 Dynamic Self Attention Encoder

class pytorch_wrapper.modules.dynamic_self_attention_encoder.**DynamicSelfAttentionEncoder**(*ti*
at
at
pr
je
ti
pr
je
ti
ob
je
at
te
ob
je
is

Bases: sphinx.ext.autodoc.importer._MockObject

Dynamic Self Attention Encoder (https://arxiv.org/abs/1808.07383).

Parameters

- **time_step_size** – Time step size.
- **att_scores_nb** – Number of attended representations.
- **att_iterations** – Number of iterations of the dynamic self-attention algorithm.
- **projection_size** – Size of the projection layer.
- **projection_activation** – Callable that creates the activation of the projection layer.

- **attended_representation_activation** – Callable that creates the activation used on the attended representations after each iteration.
- **is_end_padded** – Whether to mask at the end.

forward(*batch_sequences*, *batch_sequence_lengths*)

Parameters

- **batch_sequences** – 3D Tensor (batch_size, sequence_length, time_step_size).
- **batch_sequence_lengths** – 1D Tensor (batch_size) containing the lengths of the sequences.

Returns 2D Tensor (batch_size, projection_size * att_scores_nb) containing the encodings.

4.2 Embedding Layer

class pytorch_wrapper.modules.embedding_layer.**EmbeddingLayer**(*vocab_size*, *emb_size*, *trainable*, *padding_idx=None*)

Bases: sphinx.ext.autodoc.importer._MockObject

Embedding Layer.

Parameters

- **vocab_size** – Size of the vocabulary.
- **emb_size** – Size of the embeddings.
- **trainable** – Whether the embeddings should be altered during training.
- **padding_idx** – Index of the vector to be initialized with zeros.

forward(*x*)

load_embeddings(*embeddings*)
Loads pre-trained embeddings.

Parameters **embeddings** – Numpy array of the appropriate size containing the pre-trained embeddings.

4.3 Layer Norm

class pytorch_wrapper.modules.layer_norm.**LayerNorm**(*last_dim_size*, *eps=1e-06*)

Bases: sphinx.ext.autodoc.importer._MockObject

Layer Normalization (https://arxiv.org/pdf/1607.06450.pdf).

Parameters

- **last_dim_size** – Size of last dimension.
- **eps** – Small number for numerical stability (avoid division by zero).

forward(*x*)

Parameters **x** – Tensor to be layer normalized.

Returns Layer normalized Tensor.

4.4 MLP

class pytorch_wrapper.modules.mlp.**MLP**(*input_size*, *input_activation=None*, *input_dp=None*, *input_pre_activation_bn=False*, *input_post_activation_bn=False*, *input_pre_activation_ln=False*, *input_post_activation_ln=False*, *num_hidden_layers=1*, *hidden_layer_size=128*, *hidden_layer_bias=True*, *hidden_layer_init=None*, *hidden_layer_bias_init=None*, *hidden_activation=<sphinx.ext.autodoc.importer._MockObject object>*, *hidden_dp=None*, *hidden_layer_pre_activation_bn=False*, *hidden_layer_post_activation_bn=False*, *hidden_layer_pre_activation_ln=False*, *hidden_layer_post_activation_ln=False*, *output_layer_init=None*, *output_layer_bias_init=None*, *output_size=1*, *output_layer_bias=True*, *output_activation=None*, *output_dp=None*, *output_layer_pre_activation_bn=False*, *output_layer_post_activation_bn=False*, *output_layer_pre_activation_ln=False*, *output_layer_post_activation_ln=False*)

Bases: sphinx.ext.autodoc.importer._MockObject

Multi Layer Perceptron.

Parameters

- **input_size** – Size of the last dimension of the input.
- **input_activation** – Callable that creates the activation used on the input.
- **input_dp** – Callable that creates the activation used on the input.
- **input_pre_activation_bn** – Whether to use batch normalization before the activation of the input layer.
- **input_post_activation_bn** – Whether to use batch normalization after the activation of the input layer.
- **input_pre_activation_ln** – Whether to use layer normalization before the activation of the input layer.
- **input_post_activation_ln** – Whether to use layer normalization after the activation of the input layer.
- **num_hidden_layers** – Number of hidden layers.
- **hidden_layer_size** – Size of hidden layers. It is also possible to provide a list containing a different size for each hidden layer.
- **hidden_layer_bias** – Whether to use bias. It is also possible to provide a list containing a different option for each hidden layer.
- **hidden_layer_init** – Callable that initializes inplace the weights of the hidden layers.
- **hidden_layer_bias_init** – Callable that initializes inplace the bias of the hidden layers.

- **hidden_activation** – Callable that creates the activation used after each hidden layer. It is also possible to provide a list containing num_hidden_layers callables.
- **hidden_dp** – Dropout probability for the hidden layers. It is also possible to provide a list containing num_hidden_layers probabilities.
- **hidden_layer_pre_activation_bn** – Whether to use batch normalization before the activation of each hidden layer.
- **hidden_layer_post_activation_bn** – Whether to use batch normalization after the activation of each hidden layer.
- **hidden_layer_pre_activation_ln** – Whether to use layer normalization before the activation of each hidden layer.
- **hidden_layer_post_activation_ln** – Whether to use layer normalization after the activation of each hidden layer.
- **output_layer_init** – Callable that initializes inplace the weights of the output layer.
- **output_layer_bias_init** – Callable that initializes inplace the bias of the output layer.
- **output_size** – Output size.
- **output_layer_bias** – Whether to use bias.
- **output_activation** – Callable that creates the activation used after the output layer.
- **output_dp** – Dropout probability for the output layer.
- **output_layer_pre_activation_bn** – Whether to use batch normalization before the activation of the output layer.
- **output_layer_post_activation_bn** – Whether to use batch normalization before the activation of the output layer.
- **output_layer_pre_activation_ln** – Whether to use layer normalization before the activation of the output layer.
- **output_layer_post_activation_ln** – Whether to use layer normalization before the activation of the output layer.

forward(*x*)

Parameters **x** – Tensor having its last dimension being of size input_size.

Returns Tensor with the same shape as x except the last dimension which is of size output_size.

4.5 Multi-Head Attention

class pytorch_wrapper.modules.multi_head_attention.**MultiHeadAttention**(*q_time_step_size*, *k_time_step_size*, *v_time_step_size*, *heads*, *attention_type='dot'*, *dp=0*, *is_end_padded=True*)

Bases: sphinx.ext.autodoc.importer._MockObject

Multi Head Attention (https://arxiv.org/pdf/1706.03762.pdf).

Parameters

- **q_time_step_size** – Query time step size.
- **k_time_step_size** – Key time step size.
- **v_time_step_size** – Value time step size.
- **heads** – Number of attention heads.
- **attention_type** – Attention type ['dot', 'multiplicative', 'additive'].
- **dp** – Dropout probability.
- **is_end_padded** – Whether to mask at the end.

forward(*q*, *k*, *v*, *q_sequence_lengths*, *k_sequence_lengths*)

Parameters

- **q** – 3D Tensor (batch_size, q_sequence_length, time_step_size) containing the queries.
- **k** – 3D Tensor (batch_size, k_sequence_length, time_step_size) containing the keys.
- **v** – 3D Tensor (batch_size, k_sequence_length, time_step_size) containing the values.
- **q_sequence_lengths** – 1D Tensor (batch_size) containing the lengths of the query sequences.
- **k_sequence_lengths** – 1D Tensor (batch_size) containing the lengths of the key sequences.

Returns 3D Tensor (batch_size, q_sequence_length, time_step_size).

4.6 Residual

class pytorch_wrapper.modules.residual.**Residual**(*module*, *residual_index=None*, *model_output_key=None*)

Bases: sphinx.ext.autodoc.importer._MockObject

Adds the input of a module to it's output.

Parameters

- **module** – The module to wrap.
- **residual_index** – The index of the input to be added. Leave None if it is not a multi-input module.
- **model_output_key** – The key of the output of the model to be added. Leave None if it is not a multi-output module.

forward(**x*)

Parameters **x** – The input of the wrapped module.

Returns The output of the wrapped module added to it's input.

4.7 Sequence Basic CNN Block

class pytorch_wrapper.modules.sequence_basic_cnn_block.**SequenceBasicCNNBlock**(*time_step_size*, *kernel_height=3*, *out_channels=300*, *activation=<sphinx.ext.autodoc.importer._MockObject object>*, *dp=0*)

Bases: sphinx.ext.autodoc.importer._MockObject

Sequence Basic CNN Block.

Parameters

- **time_step_size** – Time step size.
- **kernel_height** – Filter height.
- **out_channels** – Number of filters.
- **activation** – Callable that creates the activation function.
- **dp** – Dropout probability.

forward(*batch_sequences*)

Parameters **batch_sequences** – 3D Tensor (batch_size, sequence_length, time_step_size) containing the sequence.

Returns 2D Tensor (batch_size, sequence_length, out_channels) containing the encodings.

4.8 Sequence Basic CNN Encoder

class pytorch_wrapper.modules.sequence_basic_cnn_encoder.**SequenceBasicCNNEncoder**(*time_step_size*, *input_activation*, *kernel_heights=(2, 3, 4, 5)*, *out_channels*, *pre_pooling_activation=<object>*, *pooling_function=<object>*, *post_pooling_activation*, *post_pooling_dp*)

Bases: sphinx.ext.autodoc.importer._MockObject

Basic CNN Encoder for sequences (https://arxiv.org/abs/1408.5882).

Parameters

- **time_step_size** – Time step size.
- **input_activation** – Callable that creates the activation used on the input.
- **kernel_heights** – Tuple containing filter heights.
- **out_channels** – Number of filters for each filter height.
- **pre_pooling_activation** – Callable that creates the activation used before pooling.
- **pooling_function** – Callable that performs a pooling function before the activation.
- **post_pooling_activation** – Callable that creates the activation used after pooling.
- **post_pooling_dp** – Callable that performs a pooling function before the activation.

forward(*batch_sequences*)

Parameters **batch_sequences** – 3D Tensor (batch_size, sequence_length, time_step_size) containing the sequence.

Returns 2D Tensor (batch_size, len(kernel_heights) * out_channels) containing the encodings.

4.9 Sequence Dense CNN

class pytorch_wrapper.modules.sequence_dense_cnn.**SequenceDenseCNN**(*input_size*, *projection_layer_size=150*, *kernel_heights=(3, 5)*, *feature_map_increase=75*, *cnn_depth=3*, *output_projection_layer_size=300*, *activation=<sphinx.ext.autodoc.importer._MockObject object>*, *dp=0*, *normalize_output=True*)

Bases: sphinx.ext.autodoc.importer._MockObject

Dense CNN for sequences (https://arxiv.org/abs/1808.07383).

Parameters

- **input_size** – Time step size.
- **projection_layer_size** – Size of projection_layer.
- **kernel_heights** – Kernel height of the filters.
- **feature_map_increase** – Number of filters of each convolutional layer.
- **cnn_depth** – Number of convolutional layers per kernel height.
- **output_projection_layer_size** – Size of the output time_steps.
- **activation** – Callable that creates the activation used after each layer.
- **dp** – Dropout probability.
- **normalize_output** – Whether to perform l2 normalization on the output.

forward(*batch_sequences*)

Parameters **batch_sequences** – 3D Tensor (batch_size, sequence_length, time_step_size).

Returns 3D Tensor (batch_size, sequence_length, output_projection_layer_size).

4.10 Sinusoidal Positional Embedding Layer

class pytorch_wrapper.modules.sinusoidal_positional_embedding_layer.**SinusoidalPositionalEmb**

Bases: sphinx.ext.autodoc.importer._MockObject

Sinusoidal Positional Embeddings (https://arxiv.org/pdf/1706.03762.pdf).

Parameters

- **emb_size** – Size of the positional embeddings.

- **pad_at_end** – Whether to pad at the end.
- **init_max_sentence_length** – Initial maximum length of sentence.

create_embeddings(*num_embeddings*)

forward(*length_tensor*, *max_sequence_length*)

Parameters

- **length_tensor** – ND Tensor containing the real lengths.
- **max_sequence_length** – Int that corresponds to the size of (N+1)D dimension.

Returns (N+2)D Tensor with the positional embeddings.

4.11 Softmax Attention Layer

class pytorch_wrapper.modules.softmax_attention_encoder.**SoftmaxAttentionEncoder**(*attention_mlp*, *is_end_padded*

Bases: sphinx.ext.autodoc.importer._MockObject

Encodes a sequence using context based soft-max attention.

Parameters

- **attention_mlp** – MLP object used to generate unnormalized attention score(s). If the last dimension of the tensor returned by the MLP is larger than 1 then multi-attention is applied.
- **is_end_padded** – Whether to mask at the end.

forward(*batch_sequences*, *batch_context_vector*, *batch_sequence_lengths*)

Parameters

- **batch_sequences** – 3D Tensor (batch_size, sequence_length, time_step_size).
- **batch_context_vector** – 2D Tensor (batch_size, context_vector_size).
- **batch_sequence_lengths** – 1D Tensor (batch_size) containing the lengths of the sequences.

Returns Dict with a 2D Tensor (batch_size, time_step_size) or a 3D Tensor in case of multi-attention (batch_size, nb_attentions, time_step_size) containing the encodings (key='output') and a 2D Tensor (batch_size, sequence_length) or a 3D Tensor (batch_size, sequence_length, nb_attentions) containing the attention scores (key='att_scores').

4.12 Softmax Self Attention Layer

class pytorch_wrapper.modules.softmax_self_attention_encoder.**SoftmaxSelfAttentionEncoder**(*at*
*is_*

Bases: sphinx.ext.autodoc.importer._MockObject

Encodes a sequence using soft-max self-attention.

Parameters

- **attention_mlp** – MLP object used to generate unnormalized attention score(s). If the last dimension of the tensor returned by the MLP is larger than 1 then multi-attention is applied.

- **is_end_padded** – Whether to mask at the end.

forward(*batch_sequences*, *batch_sequence_lengths*)

Parameters

- **batch_sequences** – 3D Tensor (batch_size, sequence_length, time_step_size).
- **batch_sequence_lengths** – 1D Tensor (batch_size) containing the lengths of the sequences.

Returns Dict with a 2D Tensor (batch_size, time_step_size) or a 3D Tensor in case of multi-attention (batch_size, nb_attentions, time_step_size) containing the encodings (key='output') and a 2D Tensor (batch_size, sequence_length) or a 3D Tensor (batch_size, sequence_length, nb_attentions) containing the attention scores (key='att_scores').

4.13 Transformer Encoder

class pytorch_wrapper.modules.transformer_encoder.**TransformerEncoder**(*time_step_size*, *heads*, *depth*, *dp=0*, *use_positional_embeddings=Tr*
is_end_padded=True)

Bases: sphinx.ext.autodoc.importer._MockObject

Transformer Encoder (https://arxiv.org/pdf/1706.03762.pdf).

Parameters

- **time_step_size** – Time step size.
- **heads** – Number of attention heads.
- **depth** – Number of transformer blocks.
- **dp** – Dropout probability.
- **use_positional_embeddings** – Whether to use positional embeddings.
- **is_end_padded** – Whether to mask at the end.

forward(*batch_sequences*, *batch_sequence_lengths*)

Parameters

- **batch_sequences** – batch_sequences: 3D Tensor (batch_size, sequence_length, time_step_size).
- **batch_sequence_lengths** – 1D Tensor (batch_size) containing the lengths of the sequences.

Returns 3D Tensor (batch_size, sequence_length, time_step_size).

4.14 Transformer Encoder Block

class pytorch_wrapper.modules.transformer_encoder_block.**TransformerEncoderBlock**(*time_step_size*, *heads*, *out_mlp*, *dp=0*, *is_end_padded*

Bases: sphinx.ext.autodoc.importer._MockObject

Transformer Encoder Block (https://arxiv.org/pdf/1706.03762.pdf).

Parameters

- **time_step_size** – Time step size.
- **heads** – Number of attention heads.
- **out_mlp** – MLP that will be performed after the attended sequence is generated.
- **dp** – Dropout probability.
- **is_end_padded** – Whether to mask at the end.

forward(*batch_sequences*, *batch_sequence_lengths*)

Parameters

- **batch_sequences** – batch_sequences: 3D Tensor (batch_size, sequence_length, time_step_size).
- **batch_sequence_lengths** – 1D Tensor (batch_size) containing the lengths of the sequences.

Returns 3D Tensor (batch_size, sequence_length, time_step_size).

CHAPTER 5

Functional

pytorch_wrapper.functional.**create_mask_from_length**(*length_tensor*, *mask_size*, *zeros_at_end=True*)

Creates a binary mask based on length.

Parameters

- **length_tensor** – ND Tensor containing the lengths.
- **mask_size** – Int specifying the mask size. Usually the largest length.
- **zeros_at_end** – Whether to put the zeros of the mask at the end.

Returns (N+1)D Int Tensor (..., mask_size).

pytorch_wrapper.functional.**get_first_non_masked_element**(*data_tensor*, *lengths_tensor*, *dim*, *is_end_padded=True*)

Returns the first non masked element of a Tensor along the specified dimension.

Parameters

- **data_tensor** – ND Tensor.
- **lengths_tensor** – (dim)D Tensor containing lengths.
- **dim** – Int that corresponds to the dimension.
- **is_end_padded** – Whether the Tensor is padded at the end.

Returns (N-1)D Tensor containing the first non-masked elements along the specified dimension.

pytorch_wrapper.functional.**get_last_non_masked_element**(*data_tensor*, *lengths_tensor*, *dim*, *is_end_padded=True*)

Returns the last non masked element of a Tensor along the specified dimension.

Parameters

- **data_tensor** – ND Tensor.
- **lengths_tensor** – (dim)D Tensor containing lengths.

- **dim** – Int that corresponds to the dimension.
- **is_end_padded** – Whether the Tensor is padded at the end.

Returns (N-1)D Tensor containing the last non-masked elements along the specified dimension.

pytorch_wrapper.functional.**get_last_state_of_rnn**(*rnn_out*, *batch_sequence_lengths*, *is_bidirectional*, *is_end_padded=True*)

Returns the last state(s) of the output of an RNN.

Parameters

- **rnn_out** – 3D Tensor (batch_size, sequence_length, time_step_size).
- **batch_sequence_lengths** – 1D Tensor (batch_size) containing the lengths of the sequences.
- **is_bidirectional** – Whether the RNN is bidirectional or not.
- **is_end_padded** – Whether the Tensor is padded at the end.

Returns 2D Tensor (batch_size, time_step_size or time_step_size * 2) containing the last state(s) of the RNN.

pytorch_wrapper.functional.**masked_max_pooling**(*data_tensor*, *mask*, *dim*)

Performs masked max-pooling across the specified dimension of a Tensor.

Parameters

- **data_tensor** – ND Tensor.
- **mask** – Tensor containing a binary mask that can be broad-casted to the shape of data_tensor.
- **dim** – Int that corresponds to the dimension.

Returns (N-1)D Tensor containing the result of the max-pooling operation.

pytorch_wrapper.functional.**masked_mean_pooling**(*data_tensor*, *mask*, *dim*)

Performs masked mean-pooling across the specified dimension of a Tensor.

Parameters

- **data_tensor** – ND Tensor.
- **mask** – Tensor containing a binary mask that can be broad-casted to the shape of data_tensor.
- **dim** – Int that corresponds to the dimension.

Returns (N-1)D Tensor containing the result of the mean-pooling operation.

pytorch_wrapper.functional.**masked_min_pooling**(*data_tensor*, *mask*, *dim*)

Performs masked min-pooling across the specified dimension of a Tensor.

Parameters

- **data_tensor** – ND Tensor.
- **mask** – Tensor containing a binary mask that can be broad-casted to the shape of data_tensor.
- **dim** – Int that corresponds to the dimension.

Returns (N-1)D Tensor containing the result of the min-pooling operation.

pytorch_wrapper.functional.**pad**(*data_tensor*, *pad_size*, *dim*, *pad_at_end=True*)
Pads a Tensor with zeros along a dimension.

Parameters

- **data_tensor** – Tensor to pad.
- **pad_size** – How many zeros to append.
- **dim** – The dimension to pad.
- **pad_at_end** – Whether to pad at the end.

Returns Padded Tensor.

pytorch_wrapper.functional.**same_dropout**(*data_tensor*, *dropout_p*, *dim*, *is_model_training*)
Drops the same random elements of a Tensor across the specified dimension, during training.

Parameters

- **data_tensor** – ND Tensor.
- **dropout_p** – The dropout probability.
- **dim** – Int that corresponds to the dimension.
- **is_model_training** – Whether the model is currently training.

Returns ND Tensor.

pytorch_wrapper.functional.**sub_tensor_dropout**(*data_tensor*, *dropout_p*, *dim*, *is_model_training*)
Drops (zeroes-out) random sub-Tensors of a Tensor across the specified dimension, during training.

Parameters

- **data_tensor** – ND Tensor.
- **dropout_p** – The dropout probability.
- **dim** – Int that corresponds to the dimension.
- **is_model_training** – Whether the model is currently training.

Returns ND Tensor.

CHAPTER 6

Loss Wrappers

class pytorch_wrapper.loss_wrappers.**AbstractLossWrapper**

Bases: abc.ABC

Objects of derived classes are used to wrap a loss module providing an interface used by the System class.

calculate_loss(*batch*, *output*, *training_context*, *last_activation=None*)

Calculates the loss for a single batch.

Parameters

- **batch** – Dict that contains all information needed by the loss wrapper.
- **output** – Output of the model.
- **training_context** – Dict containing information regarding the training process.
- **last_activation** – Last activation provided to the System.

Returns Output of the loss function/module.

class pytorch_wrapper.loss_wrappers.**GenericPointWiseLossWrapper**(*loss*, *model_output_key=None*, *batch_target_key='target'*, *perform_last_activation=False*)

Bases: *pytorch_wrapper.loss_wrappers.AbstractLossWrapper*

Adapter that wraps a pointwise loss module.

Parameters

- **loss** – Loss module.
- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **perform_last_activation** – Whether to perform the last_activation.

calculate_loss(*output*, *batch*, *training_context*, *last_activation=None*)
Calculates the loss for a single batch.

Parameters

- **batch** – Dict that contains all information needed by the loss wrapper.
- **output** – Output of the model.
- **training_context** – Dict containing information regarding the training process.
- **last_activation** – Last activation provided to the System.

Returns Output of the loss function/module.

class pytorch_wrapper.loss_wrappers.**SequenceLabelingGenericPointWiseLossWrapper**(*loss*, *batch_input_se* *batch_input_ke* *model_output_k* *batch_target_ke* *per-* *form_last_activ* *end_padded=Tr*

Bases: *pytorch_wrapper.loss_wrappers.AbstractLossWrapper*

Adapter that wraps a pointwise loss module. It is used in sequence labeling tasks in order to flat the output and target while discarding invalid values due to padding.

Parameters

- **loss** – Loss module.
- **batch_input_sequence_length_idx** – The index of the input list where the lengths of the sequences can be found.
- **batch_input_key** – Key of the Dicts returned by the Dataloader objects that corresponds to the input of the model.
- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **perform_last_activation** – Whether to perform the last_activation.
- **end_padded** – Whether the sequences are end-padded.

calculate_loss(*output*, *batch*, *training_context*, *last_activation=None*)
Calculates the loss for a single batch.

Parameters

- **batch** – Dict that contains all information needed by the loss wrapper.
- **output** – Output of the model.
- **training_context** – Dict containing information regarding the training process.
- **last_activation** – Last activation provided to the System.

Returns Output of the loss function/module.

CHAPTER 7

Evaluators

class pytorch_wrapper.evaluators.**AUROCEvaluator**(*model_output_key=None*, *batch_target_key='target'*, *average='macro'*, *target_threshold=0.5*)

Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

AUROC evaluator.

Parameters

- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **average** – Type ['macro' or 'micro'] of averaging performed on the results in case of multi-label task.

calculate()

Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()

(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)

Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**AbstractEvaluator**
Bases: abc.ABC

Objects of derived classes are used to evaluate a model on a dataset using a specific metric.

calculate()
Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

calculate_at_once(*output*, *dataset*, *last_activation=None*)
Calculates the metric at once for the whole dataset.

Parameters

- **output** – Output of the model.
- **dataset** – Dict that contains all information needed for a dataset by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

Returns AbstractEvaluatorResults object.

reset()
(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)
Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**AbstractEvaluatorResults**
Bases: abc.ABC

Objects of derives classes encapsulate results of an evaluation metric.

compare_to(*other_results_object*)
Compares these results with the results of another object.

Parameters **other_results_object** – Object of the same class.

is_better_than(*other_results_object*)
Compares these results with the results of another object.

Parameters **other_results_object** – Object of the same class.

class pytorch_wrapper.evaluators.**AccuracyEvaluator**(*threshold=0.5*, *model_output_key=None*, *batch_target_key='target'*)
Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Accuracy evaluator.

Parameters

- **threshold** – Threshold above which an example is considered positive.

- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.

calculate()
Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()
(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)
Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**F1Evaluator**(*threshold=0.5*, *model_output_key=None*, *batch_target_key='target'*, *average='binary'*)
Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

F1 evaluator.

Parameters

- **threshold** – Threshold above which an example is considered positive.
- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **average** – Type ['binary', 'macro' or 'micro'] of averaging performed on the results.

calculate()
Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()
(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)
Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**GenericEvaluatorResults**(*score*, *label='score'*, *score_format='%f'*, *is_max_better=True*)

Bases: *pytorch_wrapper.evaluators.AbstractEvaluatorResults*

Generic evaluator results.

Parameters

- **score** – Numeric value that represents the score.
- **label** – String used in the str representation.
- **score_format** – Format String used in the str representation.
- **is_max_better** – Flag that signifies if larger means better.

compare_to(*other_results_object*)

Compares these results with the results of another object.

Parameters **other_results_object** – Object of the same class.

is_better_than(*other_results_object*)

Compares these results with the results of another object.

Parameters **other_results_object** – Object of the same class.

is_max_better

score

class pytorch_wrapper.evaluators.**GenericPointWiseLossEvaluator**(*loss_wrapper*, *label='loss'*, *score_format='%f'*, *batch_target_key='target'*)

Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Adapter that uses an object of a class derived from AbstractLossWrapper to calculate the loss during evaluation.

Parameters

- **loss_wrapper** – AbstractLossWrapper object that calculates the loss.
- **label** – Str used as label during printing of the loss.
- **score_format** – Format used for str representation of the loss.
- **batch_target_key** – Key where the dict (batch) contains the target values.

calculate()

Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()

(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)

Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.

- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**MultiClassAccuracyEvaluator**(*model_output_key=None*, *batch_target_key='target'*)

Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Multi-Class Accuracy evaluator.

Parameters

- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.

calculate()

Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()

(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)

Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**MultiClassF1Evaluator**(*model_output_key=None*, *batch_target_key='target'*, *average='macro'*)

Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Multi-Class F1 evaluator.

Parameters

- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **average** – Type ['macro' or 'micro'] of averaging performed on the results.

calculate()

Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()

(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)

Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**MultiClassPrecisionEvaluator**(*model_output_key=None*, *batch_target_key='target'*, *average='macro'*)

Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Multi-Class Precision evaluator.

Parameters

- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **average** – Type ['macro' or 'micro'] of averaging performed on the results.

calculate()

Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()

(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)

Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**MultiClassRecallEvaluator**(*model_output_key=None*, *batch_target_key='target'*, *average='macro'*)

Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Multi-Class Recall evaluator.

Parameters

- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **average** – Type ['macro' or 'micro'] of averaging performed on the results.

calculate()

Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()
(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)
Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**PrecisionEvaluator**(*threshold=0.5*, *model_output_key=None*, *batch_target_key='target'*, *average='binary'*)
Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Precision evaluator.

Parameters

- **threshold** – Threshold above which an example is considered positive.
- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **average** – Type ['binary', 'macro' or 'micro'] of averaging performed on the results.

calculate()
Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()
(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)
Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**RecallEvaluator**(*threshold=0.5*, *model_output_key=None*, *batch_target_key='target'*, *average='binary'*)
Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Recall evaluator.

Parameters

- **threshold** – Threshold above which an example is considered positive.
- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **average** – Type ['binary', 'macro' or 'micro'] of averaging performed on the results.

calculate()
Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()
(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)
Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

class pytorch_wrapper.evaluators.**SequenceLabelingEvaluatorWrapper**(*evaluator*, *batch_input_sequence_length_idx*, *batch_input_key='input'*, *model_output_key=None*, *batch_target_key='target'*, *end_padded=True*)

Bases: *pytorch_wrapper.evaluators.AbstractEvaluator*

Adapter that wraps a pointwise loss module. It is used in sequence labeling tasks in order to flat the output and target while discarding invalid values due to padding.

Parameters

- **evaluator** – The evaluator.
- **batch_input_sequence_length_idx** – The index of the input list where the lengths of the sequences can be found.
- **batch_input_key** – Key of the Dicts returned by the Dataloader objects that corresponds to the input of the model.
- **model_output_key** – Key where the dict returned by the model contains the actual predictions. Leave None if the model returns only the predictions.
- **batch_target_key** – Key where the dict (batch) contains the target values.
- **end_padded** – Whether the sequences are end-padded.

calculate()
Called after all batches have been processed. Calculates the metric.

Returns AbstractEvaluatorResults object.

reset()
(Re)initializes the object. Called at the beginning of the evaluation step.

step(*output*, *batch*, *last_activation=None*)

Gathers information needed for performance measurement about a single batch. Called after each batch in the evaluation step.

Parameters

- **output** – Output of the model.
- **batch** – Dict that contains all information needed for a single batch by the evaluator.
- **last_activation** – The last activation of the model. Some losses work with logits and as such the last activation might not be performed inside the model's forward method.

CHAPTER 8

Samplers

class pytorch_wrapper.samplers.**OrderedBatchWiseRandomSampler**(*data_source*, *get_order_value_callable*, *batch_size*, *seed=1234*)

Bases: sphinx.ext.autodoc.importer._MockObject

Semi-randomly samples indexes from a dataset ensuring that the corresponding examples will have similar values. Values are returned by a callable.

Parameters

- **data_source** – a data source (usually a dataset object).
- **get_order_value_callable** – a callable that takes as input the example's index and returns the ordering value.
- **batch_size** – the batch size.
- **seed** – the initial seed.

class pytorch_wrapper.samplers.**OrderedSequentialSampler**(*data_source*, *get_order_value_callable*)

Bases: sphinx.ext.autodoc.importer._MockObject

Samples elements from a dataset ordered by a value returned by a callable for each example.

Parameters

- **data_source** – a data source (usually a dataset object).
- **get_order_value_callable** – a callable that takes as input the example's index and returns the ordering value.

class pytorch_wrapper.samplers.**SubsetOrderedBatchWiseRandomSampler**(*indexes*, *get_order_value_callable*, *batch_size*, *seed=1234*)

Bases: sphinx.ext.autodoc.importer._MockObject

Semi-randomly samples indexes from a list ensuring that the corresponding examples will have similar values. Values are returned by a callable.

Parameters

- **indexes** – a list of indexes.
- **get_order_value_callable** – a callable that takes as input the example's index and returns the ordering value.
- **batch_size** – the batch size.
- **seed** – the initial seed.

class pytorch_wrapper.samplers.**SubsetOrderedSequentialSampler**(*indexes*, *get_order_value_callable*)

Bases: sphinx.ext.autodoc.importer._MockObject

Samples elements from a list of indexes ordered by a value returned by a callable for each example.

Parameters

- **indexes** – a list of indexes.
- **get_order_value_callable** – a callable that takes as input the example's index and returns the ordering value.

class pytorch_wrapper.samplers.**SubsetSequentialSampler**(*indexes*)

Bases: sphinx.ext.autodoc.importer._MockObject

Samples elements sequentially based on a list of indexes.

Parameters **indexes** – a list of indexes.

CHAPTER 9

Training Callbacks

class pytorch_wrapper.training_callbacks.**AbstractCallback**

Bases: abc.ABC

Objects of derived classes inject functionality in several points of the training process.

on_batch_end(*training_context*)

Called after a batch has been processed.

Parameters **training_context** – Dict containing information regarding the training process.

on_batch_start(*training_context*)

Called just before processing a new batch.

Parameters **training_context** – Dict containing information regarding the training process.

on_epoch_end(*training_context*)

Called at the end of an epoch.

Parameters **training_context** – Dict containing information regarding the training process.

on_epoch_start(*training_context*)

Called at the beginning of a new epoch.

Parameters **training_context** – Dict containing information regarding the training process.

on_evaluation_end(*training_context*)

Called at the end of the evaluation step.

Parameters **training_context** – Dict containing information regarding the training process.

on_evaluation_start(*training_context*)

Called at the beginning of the evaluation step.

Parameters **training_context** – Dict containing information regarding the training process.

on_training_end(*training_context*)
Called at the end of the training process.

Parameters **training_context** – Dict containing information regarding the training process.

on_training_start(*training_context*)
Called at the beginning of the training process.

Parameters **training_context** – Dict containing information regarding the training process.

post_backward_calculation(*training_context*)
Called just after backward is called.

Parameters **training_context** – Dict containing information regarding the training process.

post_loss_calculation(*training_context*)
Called just after loss calculation.

Parameters **training_context** – Dict containing information regarding the training process.

post_predict(*training_context*)
Called just after prediction during training time.

Parameters **training_context** – Dict containing information regarding the training process.

pre_optimization_step(*training_context*)
Called just before the optimization step.

Parameters **training_context** – Dict containing information regarding the training process.

class pytorch_wrapper.training_callbacks.**EarlyStoppingCriterionCallback**(*patience*, *evaluation_data_loader_key*, *evaluator_key*, *tmp_best_state_filepath*)

Bases: *pytorch_wrapper.training_callbacks.StoppingCriterionCallback*

Stops the training process if the results do not get better for a number of epochs.

Parameters

- **patience** – How many epochs to forgive deteriorating results.
- **evaluation_data_loader_key** – Key of the data-loader dict (provided as an argument to the train method of System) that corresponds to the data-set that the early stopping method considers.
- **evaluator_key** – Key of the evaluators dict (provided as an argument to the train method of System) that corresponds to the evaluator that the early stopping method considers.

- **tmp_best_state_filepath** – Path where the state of the best so far model will be saved.

on_evaluation_end(*training_context*)
Called at the end of the evaluation step.

Parameters **training_context** – Dict containing information regarding the training process.

on_training_end(*training_context*)
Called at the end of the training process.

Parameters **training_context** – Dict containing information regarding the training process.

on_training_start(*training_context*)
Called at the beginning of the training process.

Parameters **training_context** – Dict containing information regarding the training process.

class pytorch_wrapper.training_callbacks.**NumberOfEpochsStoppingCriterionCallback**(*nb_of_epochs*
Bases: *pytorch_wrapper.training_callbacks.StoppingCriterionCallback*

Stops the training process after a number of epochs.

Parameters **nb_of_epochs** – Number of epochs to train.

on_epoch_end(*training_context*)
Called at the end of an epoch.

Parameters **training_context** – Dict containing information regarding the training process.

class pytorch_wrapper.training_callbacks.**StoppingCriterionCallback**
Bases: *pytorch_wrapper.training_callbacks.AbstractCallback*

CHAPTER 10

Tuner

class pytorch_wrapper.tuner.**AbstractTuner**(*hyper_parameter_generators*, *algorithm*, *fit_iterations*)

Bases: abc.ABC

Objects of derived classes are used to tune a model using the Hyperopt library.

Parameters

- **hyper_parameter_generators** – Dict containing a hyperopt hyper-parameter generator for each hyper-parameter (e.g. {'batch_size': hp.choice('batch_size', [32, 64])})
- **algorithm** – Hyperopt's tuning algorithm (e.g. hyperopt.rand.suggest, hyperopt.tpe.suggest).
- **fit_iterations** – Number of trials.

run(*trials_load_path=None*, *trials_save_path=None*)

Initiates the tuning algorithm.

Parameters

- **trials_load_path** – Path of a Trials object to load at the beginning of the tuning algorithm. If None the tuning algorithm will start from scratch.
- **trials_save_path** – Path where to save the Trials object after each iteration. If None the Trials object will not be saved.

Returns A sorted list of tuples [(loss, {parameters}), ...].

class pytorch_wrapper.tuner.**Tuner**(*hyper_parameter_generators*, *step_function*, *algorithm*, *fit_iterations*)

Bases: *pytorch_wrapper.tuner.AbstractTuner*

Objects of this class are used to tune a model using the Hyperopt library.

Parameters

- **hyper_parameter_generators** – Dict containing a hyperopt hyper-parameter generator for each hyper-parameter (e.g. {'batch_size': hp.choice('batch_size', [32, 64])})

- **step_function** – callable that creates and evaluates a model using the provided hyper-parameters. A dict will be provided as an argument containing the chosen hyper-parameters for the current iteration. The key for each hyper-parameter is the same as its corresponding generator. It must return a numeric value representing the loss of the current iteration.
- **algorithm** – Hyperopt's tuning algorithm (e.g. hyperopt.rand.suggest, hyperopt.tpe.suggest).
- **fit_iterations** – Number of trials.

Python Module Index

p

pytorch_wrapper.evaluators, 29
pytorch_wrapper.functional, 23
pytorch_wrapper.loss_wrappers, 27
pytorch_wrapper.modules.dynamic_self_attention_encoder, 11
pytorch_wrapper.modules.embedding_layer, 12
pytorch_wrapper.modules.layer_norm, 12
pytorch_wrapper.modules.mlp, 13
pytorch_wrapper.modules.multi_head_attention, 14
pytorch_wrapper.modules.residual, 15
pytorch_wrapper.modules.sequence_basic_cnn_block, 16
pytorch_wrapper.modules.sequence_basic_cnn_encoder, 17
pytorch_wrapper.modules.sequence_dense_cnn, 18
pytorch_wrapper.modules.sinusoidal_positional_embedding_layer, 18
pytorch_wrapper.modules.softmax_attention_encoder, 19
pytorch_wrapper.modules.softmax_self_attention_encoder, 19
pytorch_wrapper.modules.transformer_encoder, 20
pytorch_wrapper.modules.transformer_encoder_block, 21
pytorch_wrapper.samplers, 39
pytorch_wrapper.system, 7
pytorch_wrapper.training_callbacks, 41
pytorch_wrapper.tuner, 45

Index

A

AbstractCallback (*class in pytorch_wrapper.training_callbacks*), 41
AbstractEvaluator (*class in pytorch_wrapper.evaluators*), 29
AbstractEvaluatorResults (*class in pytorch_wrapper.evaluators*), 30
AbstractLossWrapper (*class in pytorch_wrapper.loss_wrappers*), 27
AbstractTuner (*class in pytorch_wrapper.tuner*), 45
AccuracyEvaluator (*class in pytorch_wrapper.evaluators*), 30
AUROCEvaluator (*class in pytorch_wrapper.evaluators*), 29

C

calculate() (*pytorch_wrapper.evaluators.AbstractEvaluator method*), 30
calculate() (*pytorch_wrapper.evaluators.AccuracyEvaluator method*), 31
calculate() (*pytorch_wrapper.evaluators.AUROCEvaluator method*), 29
calculate() (*pytorch_wrapper.evaluators.F1Evaluator method*), 31
calculate() (*pytorch_wrapper.evaluators.GenericPointWiseLossEvaluator method*), 32
calculate() (*pytorch_wrapper.evaluators.MultiClassAccuracyEvaluator method*), 33
calculate() (*pytorch_wrapper.evaluators.MultiClassF1Evaluator method*), 33
calculate() (*pytorch_wrapper.evaluators.MultiClassPrecisionEvaluator method*), 34
calculate() (*pytorch_wrapper.evaluators.MultiClassRecallEvaluator method*), 34
calculate() (*pytorch_wrapper.evaluators.PrecisionEvaluator method*), 35
calculate() (*pytorch_wrapper.evaluators.RecallEvaluator method*), 36
calculate() (*pytorch_wrapper.evaluators.SequenceLabelingEvaluatorWrapper method*), 36
calculate_at_once() (*pytorch_wrapper.evaluators.AbstractEvaluator method*), 30
calculate_loss() (*pytorch_wrapper.loss_wrappers.AbstractLossWrapper method*), 27
calculate_loss() (*pytorch_wrapper.loss_wrappers.GenericPointWiseLossWrapper method*), 27
calculate_loss() (*pytorch_wrapper.loss_wrappers.SequenceLabelingGenericPointWis method*), 28
compare_to() (*pytorch_wrapper.evaluators.AbstractEvaluatorResults method*), 30
compare_to() (*pytorch_wrapper.evaluators.GenericEvaluatorResults method*), 32
create_embeddings() (*pytorch_wrapper.modules.sinusoidal_positional_embedding_layer.S method*), 19
create_mask_from_length() (*in module pytorch_wrapper.functional*), 23

D

device (*pytorch_wrapper.system.System attribute*), 7
DynamicSelfAttentionEncoder (*class in pytorch_wrapper.modules.dynamic_self_attention_encoder*), 11

E

EarlyStoppingCriterionCallback (*class in pytorch_wrapper.training_callbacks*), 42
EmbeddingLayer (*class in pytorch_wrapper.modules.embedding_layer*), 12
evaluate() (*pytorch_wrapper.system.System method*), 7

F

F1Evaluator (*class in pytorch_wrapper.evaluators*),

31
forward() (pytorch_wrapper.modules.dynamic_self_attention_encoder.DynamicSelfAttentionEncoder method), 12
forward() (pytorch_wrapper.modules.embedding_layer.EmbeddingLayer method), 12
forward() (pytorch_wrapper.modules.layer_norm.LayerNorm method), 12
forward() (pytorch_wrapper.modules.mlp.MLP method), 14
forward() (pytorch_wrapper.modules.multi_head_attention.MultiHeadAttention method), 15
forward() (pytorch_wrapper.modules.residual.Residual method), 15
forward() (pytorch_wrapper.modules.sequence_basic_cnn_block.SequenceBasicCNNBlock method), 16
forward() (pytorch_wrapper.modules.sequence_basic_cnn_encoder.SequenceBasicCNNEncoder method), 17
forward() (pytorch_wrapper.modules.sequence_dense_cnn.SequenceDenseCNN method), 18
forward() (pytorch_wrapper.modules.sinusoidal_positional_embedding_layer.SinusoidalPositionalEmbeddingLayer method), 19
forward() (pytorch_wrapper.modules.softmax_attention_encoder.SoftmaxAttentionEncoder method), 19
forward() (pytorch_wrapper.modules.softmax_self_attention_encoder.SoftmaxSelfAttentionEncoder method), 20
forward() (pytorch_wrapper.modules.transformer_encoder.TransformerEncoder method), 20
forward() (pytorch_wrapper.modules.transformer_encoder_block.TransformerEncoderBlock method), 21

G

GenericEvaluatorResults (class in pytorch_wrapper.evaluators), 31
GenericPointWiseLossEvaluator (class in pytorch_wrapper.evaluators), 32
GenericPointWiseLossWrapper (class in pytorch_wrapper.loss_wrappers), 27
get_first_non_masked_element() (in module pytorch_wrapper.functional), 23
get_last_non_masked_element() (in module pytorch_wrapper.functional), 23
get_last_state_of_rnn() (in module pytorch_wrapper.functional), 24

I

is_better_than() (pytorch_wrapper.evaluators.AbstractEvaluatorResults method), 30
is_better_than() (pytorch_wrapper.evaluators.GenericEvaluatorResults method), 32
is_max_better (pytorch_wrapper.evaluators.GenericEvaluatorResults attribute), 32

L

LayerNorm (class in pytorch_wrapper.modules.layer_norm), 12
load() (pytorch_wrapper.system.System static method), 7
load_embeddings() (pytorch_wrapper.modules.embedding_layer.EmbeddingLayer method), 12
load_model_state() (pytorch_wrapper.system.System method), 8

M

masked_max_pooling() (in module pytorch_wrapper.functional), 24
masked_mean_pooling() (in module pytorch_wrapper.functional), 24
masked_min_pooling() (in module pytorch_wrapper.functional), 24
MLP (class in pytorch_wrapper.modules.mlp), 13
MultiClassAccuracyEvaluator (class in pytorch_wrapper.evaluators), 33
MultiClassF1Evaluator (class in pytorch_wrapper.evaluators), 33
MultiClassPrecisionEvaluator (class in pytorch_wrapper.evaluators), 34
MultiClassRecallEvaluator (class in pytorch_wrapper.evaluators), 34
MultiHeadAttention (class in pytorch_wrapper.modules.multi_head_attention), 14

N

NumberOfEpochsStoppingCriterionCallback (class in pytorch_wrapper.training_callbacks), 43

O

on_batch_end() (pytorch_wrapper.training_callbacks.AbstractCallback method), 41
on_batch_start() (pytorch_wrapper.training_callbacks.AbstractCallback method), 41
on_epoch_end() (pytorch_wrapper.training_callbacks.AbstractCallback method), 41
on_epoch_end() (pytorch_wrapper.training_callbacks.NumberOfEpochsStoppingCrit method), 43
on_epoch_start() (pytorch_wrapper.training_callbacks.AbstractCallback method), 41

on_evaluation_end() (pytorch_wrapper.training_callbacks.AbstractCallback method), 41
on_evaluation_end() (pytorch_wrapper.training_callbacks.EarlyStoppingCriterionCallback method), 43
on_evaluation_start() (pytorch_wrapper.training_callbacks.AbstractCallback method), 41
on_training_end() (pytorch_wrapper.training_callbacks.AbstractCallback method), 42
on_training_end() (pytorch_wrapper.training_callbacks.EarlyStoppingCriterionCallback method), 43
on_training_start() (pytorch_wrapper.training_callbacks.AbstractCallback method), 42
on_training_start() (pytorch_wrapper.training_callbacks.EarlyStoppingCriterionCallback method), 43
OrderedBatchWiseRandomSampler (class in pytorch_wrapper.samplers), 39
OrderedSequentialSampler (class in pytorch_wrapper.samplers), 39

P

pad() (in module pytorch_wrapper.functional), 24
post_backward_calculation() (pytorch_wrapper.training_callbacks.AbstractCallback method), 42
post_loss_calculation() (pytorch_wrapper.training_callbacks.AbstractCallback method), 42
post_predict() (pytorch_wrapper.training_callbacks.AbstractCallback method), 42
pre_optimization_step() (pytorch_wrapper.training_callbacks.AbstractCallback method), 42
PrecisionEvaluator (class in pytorch_wrapper.evaluators), 35
predict() (pytorch_wrapper.system.System method), 8
predict_batch() (pytorch_wrapper.system.System method), 8
pure_predict() (pytorch_wrapper.system.System method), 8
pytorch_wrapper.evaluators (module), 29
pytorch_wrapper.functional (module), 23
pytorch_wrapper.loss_wrappers (module), 27
pytorch_wrapper.modules.dynamic_self_attention (module), 11
pytorch_wrapper.modules.embedding_layer (module), 12
pytorch_wrapper.modules.layer_norm (module), 12
pytorch_wrapper.modules.mlp (module), 13
pytorch_wrapper.modules.multi_head_attention (module), 14
pytorch_wrapper.modules.residual (module), 15
pytorch_wrapper.modules.sequence_basic_cnn_block (module), 16
pytorch_wrapper.modules.sequence_basic_cnn_encoder (module), 17
pytorch_wrapper.modules.sequence_dense_cnn (module), 18
pytorch_wrapper.modules.sinusoidal_positional_embed (module), 18
pytorch_wrapper.modules.softmax_attention_encoder (module), 19
pytorch_wrapper.modules.softmax_self_attention_enco (module), 19
pytorch_wrapper.modules.transformer_encoder (module), 20
pytorch_wrapper.modules.transformer_encoder_block (module), 21
pytorch_wrapper.samplers (module), 39
pytorch_wrapper.system (module), 7
pytorch_wrapper.training_callbacks (module), 41
pytorch_wrapper.tuner (module), 45

R

RecallEvaluator (class in pytorch_wrapper.evaluators), 35
reset() (pytorch_wrapper.evaluators.AbstractEvaluator method), 30
reset() (pytorch_wrapper.evaluators.AccuracyEvaluator method), 31
reset() (pytorch_wrapper.evaluators.AUROCEvaluator method), 29
reset() (pytorch_wrapper.evaluators.F1Evaluator method), 31
reset() (pytorch_wrapper.evaluators.GenericPointWiseLossEvaluator method), 32
reset() (pytorch_wrapper.evaluators.MultiClassAccuracyEvaluator method), 33
reset() (pytorch_wrapper.evaluators.MultiClassF1Evaluator method), 33
reset() (pytorch_wrapper.evaluators.MultiClassPrecisionEvaluator method), 34
reset() (pytorch_wrapper.evaluators.MultiClassRecallEvaluator method), 34
reset() (pytorch_wrapper.evaluators.PrecisionEvaluator method), 35

reset() (*pytorch_wrapper.evaluators.RecallEvaluator method*), 36
reset() (*pytorch_wrapper.evaluators.SequenceLabelingEvaluatorWrapper method*), 36
Residual (*class in pytorch_wrapper.modules.residual*), 15
run() (*pytorch_wrapper.tuner.AbstractTuner method*), 45

S

same_dropout() (*in module pytorch_wrapper.functional*), 25
save() (*pytorch_wrapper.system.System method*), 8
save_model_state() (*pytorch_wrapper.system.System method*), 9
score (*pytorch_wrapper.evaluators.GenericEvaluatorResults attribute*), 32
SequenceBasicCNNBlock (*class in pytorch_wrapper.modules.sequence_basic_cnn_block*), 16
SequenceBasicCNNEncoder (*class in pytorch_wrapper.modules.sequence_basic_cnn_encoder*), 17
SequenceDenseCNN (*class in pytorch_wrapper.modules.sequence_dense_cnn*), 18
SequenceLabelingEvaluatorWrapper (*class in pytorch_wrapper.evaluators*), 36
SequenceLabelingGenericPointWiseLossWrapper (*class in pytorch_wrapper.loss_wrappers*), 28
SinusoidalPositionalEmbeddingLayer (*class in pytorch_wrapper.modules.sinusoidal_positional_embedding_layer*), 18
SoftmaxAttentionEncoder (*class in pytorch_wrapper.modules.softmax_attention_encoder*), 19
SoftmaxSelfAttentionEncoder (*class in pytorch_wrapper.modules.softmax_self_attention_encoder*), 19
step() (*pytorch_wrapper.evaluators.AbstractEvaluator method*), 30
step() (*pytorch_wrapper.evaluators.AccuracyEvaluator method*), 31
step() (*pytorch_wrapper.evaluators.AUROCEvaluator method*), 29
step() (*pytorch_wrapper.evaluators.F1Evaluator method*), 31
step() (*pytorch_wrapper.evaluators.GenericPointWiseLossEvaluator method*), 32
step() (*pytorch_wrapper.evaluators.MultiClassAccuracyEvaluator method*), 33
step() (*pytorch_wrapper.evaluators.MultiClassF1Evaluator method*), 33
step() (*pytorch_wrapper.evaluators.MultiClassPrecisionEvaluator method*), 34
step() (*pytorch_wrapper.evaluators.MultiClassRecallEvaluator method*), 35
step() (*pytorch_wrapper.evaluators.PrecisionEvaluator method*), 35
step() (*pytorch_wrapper.evaluators.RecallEvaluator method*), 36
step() (*pytorch_wrapper.evaluators.SequenceLabelingEvaluatorWrapper method*), 36
StoppingCriterionCallback (*class in pytorch_wrapper.training_callbacks*), 43
sub_tensor_dropout() (*in module pytorch_wrapper.functional*), 25
SubsetOrderedBatchWiseRandomSampler (*class in pytorch_wrapper.samplers*), 39
SubsetOrderedSequentialSampler (*class in pytorch_wrapper.samplers*), 40
SubsetSequentialSampler (*class in pytorch_wrapper.samplers*), 40
System (*class in pytorch_wrapper.system*), 7

T

to() (*pytorch_wrapper.system.System method*), 9
train() (*pytorch_wrapper.system.System method*), 9
train_on_multi_gpus() (*pytorch_wrapper.system.System method*), 9
TransformerEncoder (*class in pytorch_wrapper.modules.transformer_encoder*), 20
TransformerEncoderBlock (*class in pytorch_wrapper.modules.transformer_encoder_block*), 21
Tuner (*class in pytorch_wrapper.tuner*), 45