MLBench Core Documentation

MLBench development team

Jul 09, 2021

MLBENCH

CHAPTER

ONE

MLBENCH CORE API

1.1 Examples

1.2 mlbench_core.aggregation

1.2.1 pytorch

Aggregation

class mlbench_core.aggregation.pytorch.aggregation.**Aggregation**(*use_cuda=False*)

Aggregate updates / models from different processes.

Parameters **use_cuda** (*bool*) – Whether to use CUDA tensors for communication

abstract **_agg**(*self*, *data*, *op*, *denom=None*)

Aggregate data using *op* operation.

Parameters

- **data** (torch.Tensor) – A Tensor to be aggregated.
- **op** (*str*) – Aggregation methods like *avg*, *sum*, *min*, *max*, etc.
- **denom** (torch.Tensor, optional) – Custom denominator to average by Use with op == *custom_avg*. (default: *None*)

Returns An aggregated tensor.

Return type torch.Tensor

_agg_gradients_by_layer(*self*, *model*, *op*, *denom=None*)

Aggregate models gradients each layer individually

Parameters

- **model** (torch.Module) – Models to be averaged.
- **op** (*str*) – Aggregation method. Should be in *ALLREDUCE_AGGREGATION_OPS*
- **denom** (torch.Tensor, optional) – Custom denominator to average by Use with op == *custom_avg*. (default: *None*)

_agg_gradients_by_model(*self*, *model*, *op*, *denom=None*)

Aggregate models gradients, all layers at once

Parameters

- **model** (torch.Module) – Models to be averaged.

- **op** (*str*) – Aggregation method. Should be in *ALLREDUCE_AGGREGATION_OPS*
- **denom** (torch.Tensor, optional) – Custom denominator to average by Use with op == *custom_avg*. (default: *None*)

_agg_weights_by_layer(*self*, *model*, *op*, *denom=None*)

Aggregate models by model weight, for each layer individually

Parameters

- **model** (torch.Module) – Models to be averaged.
- **op** (*str*) – Aggregation method. Should be in *ALLREDUCE_AGGREGATION_OPS*
- **denom** (torch.Tensor, optional) – Custom denominator to average by Use with op == *custom_avg*. (default: *None*)

_agg_weights_by_model(*self*, *model*, *op*, *denom=None*)

Aggregate models by model weight, all layers at once

Parameters

- **model** (torch.Module) – Models to be averaged.
- **op** (*str*) – Aggregation method. Should be in *ALLREDUCE_AGGREGATION_OPS*
- **denom** (torch.Tensor, optional) – Custom denominator to average by Use with op == *custom_avg*. (default: *None*)

agg_grad(*self*, *by_layer=False*)

agg_model(*self*, *by_layer=False*)

Centralized (Synchronous) aggregation

All-Reduce

class mlbench_core.aggregation.pytorch.centralized.**AllReduceAggregation**(*world_size*, *divide_before=False*, *use_cuda=False*)

Bases: *mlbench_core.aggregation.pytorch.aggregation.Aggregation*

Aggregate weights / models from different processes using all-reduce aggregation

Parameters

- **world_size** (*int*) – Current distributed world size
- **divide_before** (*bool*) – Perform division before reduction (avoid overflow)
- **use_cuda** (*bool*) – Use cuda tensors for reduction

All-Reduce Horovod

class mlbench_core.aggregation.pytorch.centralized.**AllReduceAggregationHVD**(*world_size*, *divide_before=False*, *use_cuda=False*)

Bases: *AllReduceAggregation*

Implements *AllReduceAggregation* using horovod for communication

Sparsified Aggregation

class mlbench_core.aggregation.pytorch.centralized.**SparsifiedAggregation**(*model*, *use_cuda=False*)

Bases: *mlbench_core.aggregation.pytorch.aggregation.Aggregation*

Aggregate sparsified updates.

Power Aggregation

class mlbench_core.aggregation.pytorch.centralized.**PowerAggregation**(*model*, *use_cuda=False*, *reuse_query=False*, *world_size=1*, *rank=1*)

Bases: *mlbench_core.aggregation.pytorch.aggregation.Aggregation*

Aggregate updates using power iteration and error feedback.

Parameters

- **model** (nn.Module) – Model which contains parameters for SGD
- **use_cuda** (*bool*) – Whether to use cuda tensors for aggregation
- **reuse_query** (*bool*) – Whether to use warm start to initialize the power iteration
- **rank** (*int*) – The rank of the gradient approximation

Decentralized (Asynchronous) aggregation

Decentralized Aggregation

class mlbench_core.aggregation.pytorch.decentralized.**DecentralizedAggregation**(*rank*, *neighbors*, *use_cuda=False*)

Bases: *mlbench_core.aggregation.pytorch.aggregation.Aggregation*

Aggregate updates in a decentralized manner.

1.3 mlbench_core.api

MLBench Master/Dashboard API Client Functionality

mlbench_core.api.**MLBENCH_IMAGES**
Dict of official benchmark images

> **Note:** Format: `{name: (image_name, command, run_on_all, GPU_supported)}`

class mlbench_core.api.**ApiClient**(*max_workers=5*, *in_cluster=True*, *label_selector='component=master,app=mlbench'*, *k8s_namespace='default'*, *url=None*, *load_config=True*)
Client for the mlbench Master/Dashboard REST API

When used inside a cluster, will use the API Pod IP directly for communication. When used outside of a cluster, will try to figure out how to access the API depending on the K8s service type, if it's accessible. Endpoint URL can also be set manually.

All requests are executed in a separate process to ensure non-blocking execution. Results are returned as `concurrent.futures.Future` objects wrapping `requests` responses.

Expects K8s credentials to be set correctly (automatic inside a cluster, through kubectl outside of it)

Parameters

- **max_workers** (*int*) – maximum number of processes to run in parallel
- **in_cluster** (*bool*) – Whether the client is run inside the K8s cluster or not
- **label_selector** (*str*) – K8s label selectors to find the master pod when running inside a cluster. Default: `component=master,app=mlbench`
- **k8s_namespace** (*str*) – K8s namespace mlbench is running in. Default: `default`
- **url** (*str*) – ip:port/path or hostname:port/path that overrides automatic endpoint detection, pointing to the root of the master/dashboard node. Default: `None`

create_run(*self*, *name*, *num_workers*, *num_cpus=2.0*, *max_bandwidth=1000*, *image=None*, *backend=None*, *custom_image_name=None*, *custom_image_command=None*, *custom_backend=None*, *run_all_nodes=False*, *gpu_enabled=False*, *light_target=False*)
Create a new benchmark run.

Available official benchmarks can be found in the `mlbench_core.api.MLBENCH_IMAGES` dict.

Parameters

- **name** (*str*) – The name of the run
- **num_workers** (*int*) – The number of worker nodes to use
- **num_cpus** (*float*) – The number of CPU Cores per worker to utilize. Default: `2.0`
- **max_bandwidth** (*int*) – Maximum bandwidth available for communication between worker nodes in mbps. Default: `1000`
- **image** (*str*) – Name of the official benchmark image to use (see `mlbench_core.api.MLBENCH_IMAGES` keys). Default: `None`
- **backend** (*str*) – Name of the backend to use (see `mlbench_core.api.MLBENCH_BACKENDS`) Default: `None`

- **custom_image_name** (*str*) – The name of a custom Docker image to run. Can be a dockerhub or private Docker repository url. Default: None
- **custom_image_command** (*str*) – Command to run on the custom image. Default: None
- **custom_backend** (*str*) – Custom backend to use. Default: None
- **run_all_nodes** (*bool*) – Whether to run custom_image_command on all worker nodes or only the rank 0 node.
- **gpu_enabled** (*bool*) – Enable GPU acceleration. Default: False
- **light_target** (*bool*) – Use light target goal Default: False

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

delete_run(*self*, *run_id*)
Delete a benchmark run.

Args: run_id(str): The id of the run to get

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

download_run_metrics(*self*, *run_id*, *since=None*, *summarize=None*)
Get all metrics for a run as zip.

Parameters

- **run_id** (*str*) – The id of the run to get metrics for
- **since** (*datetime*) – Only get metrics newer than this date Default: None
- **summarize** (*int*) – If set, metrics are summarized to at most this
- **entries by averaging the metrics. Default** (*many*) – None

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

get_all_metrics(*self*)
Get all metrics ever recorded by the master node.

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

get_pod_metrics(*self*, *pod_id*, *since=None*, *summarize=None*)
Get all metrics for a worker pod.

Parameters

- **pod_id** (*str*) – The id of the pod to get metrics for
- **since** (*datetime*) – Only get metrics newer than this date Default: None
- **summarize** (*int*) – If set, metrics are summarized to at most this
- **entries by averaging the metrics. Default** (*many*) – None

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

get_run(*self*, *run_id*)
Get a specific benchmark run

Parameters **run_id** (*str*) – The id of the run to get

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

get_run_metrics(*self*, *run_id*, *since=None*, *summarize=None*, *metric_filter=None*, *last_n=None*)
Get all metrics for a run.

Parameters

- **run_id** (*str*) – The id of the run to get metrics for
- **since** (*datetime*) – Only get metrics newer than this date Default: None
- **summarize** (*int*) – If set, metrics are summarized to at most this
- **entries by averaging the metrics. Default** (*many*) – None

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

get_runs(*self*)
Get all active, finished and failed benchmark runs

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

get_worker_pods(*self*)
Get information on all worker nodes.

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

post_metric(*self*, *run_id*, *name*, *value*, *cumulative=False*, *metadata=''*, *date=None*)
Save a metric to the master node for a run.

Parameters

- **run_id** (*str*) – The id of the run to save a metric for
- **name** (*str*) – The name of the metric, e.g. accuracy
- **value** (*Number*) – The metric value to save
- **cumulative** (*bool, optional*) – Whether this metric is cumulative or not. Cumulative metrics are values that increment over time, i.e. current_calue = previous_value + value_difference. Non-cumulative values or discrete values at a certain time. Default: False
- **metadata** (*dict*) – Optional metadata to attach to a metric. Default: None
- **date** (*datetime*) – The date the metric was gathered. Default: datetime.now

Returns A concurrent.futures.Future objects wrapping requests.response object. Get the result by calling return_value.result().json()

1.4 mlbench_core.controlflow

1.4.1 pytorch

Controlflow

mlbench_core.controlflow.pytorch.**validation_round**(*dataloader*, *model*, *loss_function*, *metrics*, *dtype*, *tracker=None*, *transform_target_type=False*, *use_cuda=False*, *max_batches=None*)

Evaluate the model on the test dataset.

Parameters

- **(obj** (*tracker*) – torch.utils.data.DataLoader): The validation set
- **(obj** – torch.nn.Module): The model to train
- **(obj** – torch.nn.Module): The loss function
- **metrics** (*list*) – List of metrics to track
- **dtype** (*str*) – The datatype to use, one of *fp32 `or `fp64*
- **(obj** – mlbench_core.utils.Tracker | None): Tracker object to use.
- **transform_target_type** (*bool*) – Convert target to *dtype*. Default *False*
- **use_cuda** (*bool*) – Whether to use GPU for training, default: *False*
- **max_batches** (*int | None*) – Maximum number of batches to validate on

Returns Dictionary of average of each metric, and average validation loss

Return type (dict, float)

mlbench_core.controlflow.pytorch.**record_train_batch_stats**(*batch_idx*, *loss*, *output*, *metric_results*, *tracker*, *num_batches_per_device_train*)

Record the stats in a training batch.

Parameters

- **batch_idx** (*int*) – The id of the current batch
- **loss** (*float*) – The loss of the batch
- **output** (torch.Tensor) – The model output
- **metric_results** (*dict*) – of mlbench_core.evaluation.pytorch.metrics.MLBenchMetric: float Metrics and their values
- **tracker** (mlbench_core.utils.Tracker) – Tracker object to use.
- **num_batches_per_device_train** (*int*) – Number of batches per train epoch

mlbench_core.controlflow.pytorch.**record_validation_stats**(*metrics_values*, *loss*, *tracker=None*, *rank=0*)

Records the stats of a previously run validation

Parameters

- **metrics_values** (*dict*) – Dictionary of each metric's average.
- **loss** (*float*) – Validation loss
- **(obj** (*tracker*) – mlbench_core.utils.Tracker, optional): Tracker object to use.

- **rank** (*int*) – Current distributed rank

Returns Whether this validation round is the best

Return type (bool)

CheckpointsEvaluationControlFlow

class mlbench_core.controlflow.pytorch.**CheckpointsEvaluationControlFlow**(*ckpt_dir*, *rank*, *world_size*, *checkpointer*, *model*, *epochs*, *loss_function*, *metrics*, *use_cuda=False*, *dtype=None*, *max_batch_per_epoch=None*)

Evaluate models on training / validation dataset.

Parameters

- **ckpt_dir** (*str*) – Path to checkpoints.
- **rank** (*int*) – The rank of the current process
- **world_size** (*int*) – The total number of workers
- **checkpointer** (Checkpointer) – Used to load checkpoints.
- **model** (torch.optim.Optimizer) – An optimizer for the given model.
- **epochs** (*int*) – Number of epochs to traing.
- **loss_function** (torch.nn.modules.loss._Loss) – loss function.
- **metrics** (list of mlbench_core.evaluation.pytorch.*) – metrics like TopKAccuracy.
- **use_cuda** (*bool*) – Whether to train on GPU or not. Default: *False*
- **dtype** (*str*) – The datatype to use for the dataloader data
- **max_batch_per_epoch** (*int*) – Maximum number of batches per epoch. Whole dataset
- **used if not specified. Default** (*is*) – *None*

evaluate_by_epochs(*self*, *dataloader*)

Evaluate dataset using the averaged models.

In each epoch each process loads models and averages them. The averaged model is used to evaluate train / validation dataset.

Parameters **dataloader** (torch.utils.data.DataLoader) – The dataset to be evaluated.

Returns list of stats of models in each epoch.

Return type list

Helpers

mlbench_core.controlflow.pytorch.helpers.**maybe_range**(*maximum*)
Map an integer or None to an integer iterator starting from 0 with stride 1.

If maximum number of batches per epoch is limited, then return an finite iterator. Otherwise, return an iterator of infinite length.

Parameters **maximum** (*int | None*) – Maximum number of steps in iterator. If none, returns iterator of infinite length

Returns (iterator)

mlbench_core.controlflow.pytorch.helpers.**convert_dtype**(*dtype*, *obj*)
Converts given tensor to given dtype

Parameters

- **dtype** (*str*) – One of *fp32* or *fp64*
- **(obj** (*obj*) – torch.Tensor | *obj*:torch.nn.Module): Module or tensor to convert

Returns torch.Tensor | *obj*:torch.nn.Module): Converted tensor or module

Return type (*obj*

mlbench_core.controlflow.pytorch.helpers.**prepare_batch**(*data*, *target*, *dtype*, *transform_target_dtype=False*, *use_cuda=False*)
Prepares a batch for training by changing the type and sending to cuda if necessary

Parameters

- **(obj** (*target*) – torch.Tensor): The input tensor
- **(obj** – torch.Tensor): The target tensor
- **dtype** (*str*) – One of *fp32* or *fp64*, data type to transform input and/or target
- **transform_target_dtype** (*bool*) – Transform target to *dtype* too
- **use_cuda** (*bool*) – Send tensors to GPU

Returns torch.Tensor, *obj*:torch.Tensor): Input and target tensors

Return type (*obj*

mlbench_core.controlflow.pytorch.helpers.**iterate_dataloader**(*dataloader*, *dtype*, *max_batch_per_epoch=None*, *use_cuda=False*, *transform_target_type=False*)
Function that returns an iterator on the given loader. Can be used to limit the number of batches, converting input and target dtypes and sending to GPU

Parameters

- **(obj** (*dataloader*) – torch.utils.data.DataLoader): The loader
- **dtype** (*str*) – Type to convert to (*fp32* or *fp64*)
- **max_batch_per_epoch** (*int | None*) – Maximum number of batches
- **use_cuda** (*bool*) – Send tensors to GPU
- **transform_target_type** (*bool*) – Transform target dtype as well

Returns An iterator over the data

Return type (iterator)

1.4.2 tensorflow

TrainValidation

class mlbench_core.controlflow.tensorflow.**TrainValidation**(*train_op*, *sess*, *loss*, *metrics*, *max_train_steps*, *train_epochs*, *batch_size*, *num_batches_per_epoch_for_train*, *num_batches_per_epoch_for_validation*, *train_set_init_op*, *validation_set_init_op*, *run_id*, *rank*, *lr_scheduler_level='epoch'*, *tracker=None*)

A control flow to train and evaluate a model.

Parameters

- **train_op** (tf.Operation) – An operation for training models.
- **sess** (tf.Session) – A session which the control flow will communicate.
- **loss** (tf.Tensor) – The loss tensor.
- **metrics** (list of tf.Tensor) – A list of metrics tensors.
- **max_train_steps** (*int*) – Number of steps for training (independent of lr)
- **train_epochs** (*int*) – Number of steps for training (may related to lr).
- **batch_size** (*int*) – Size of a batch.
- **num_batches_per_epoch_for_train** (*int*) – Number of batches in one training epoch
- **num_batches_per_epoch_for_validation** (*int*) – Number of batches in one validation epoch
- **train_set_init_op** (tf.Operation) – Op for initializing training dataset.
- **validation_set_init_op** (tf.Operation) – Op for initializing validation dataset.
- **run_id** (*str*) – the id of the run in the dashboard
- **rank** (*int*) – the rank of the current worker
- **lr_scheduler_level** (*str*) – Learning rate is updated based on *epoch* or *batch*.

train_and_eval(*self*, *initial_epoch=0*, *lr_tensor_name=None*)

Train and evaluate one epoch.

Parameters

- **initial_epoch** (*int, optional*) – Defaults to 0. Initial epoch of training.
- **lr_tensor_name** (tf.Tensor, optional) – Defaults to None. A (scalar) float tensor representing name of learning rate

train_one_epoch(*self*, *lr_tensor_name=None*)

Train a model for an epoch and use tracker to log stats.

Parameters **lr_tensor** (*obj*) – The learningrate schedule tensorflow operation

valid_one_epoch(*self*)
Validate a model for an epoch and use tracker to log stats.

1.5 mlbench_core.dataset

1.5.1 linearmodels

pytorch

Epsilon Logistic Regression

class mlbench_core.dataset.linearmodels.pytorch.dataloader.**LMDBDataset**(*name*, *data_type*, *root*, *target_transform=None*)

LMDB Dataset

Parameters

- **root** (*string*) – Either root directory for the database files, or a absolute path pointing to the file.
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.

class mlbench_core.dataset.linearmodels.pytorch.dataloader.**LMDBPTClass**(*root*, *transform=None*, *target_transform=None*)

LMDB Dataset loader Class

Parameters

- **root** (*string*) – Either root directory for the database files, or a absolute path pointing to the file.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.

1.5.2 imagerecognition

pytorch

CIFAR10V1

class mlbench_core.dataset.imagerecognition.pytorch.dataloader.**CIFAR10V1**(*root*, *train=True*, *download=False*)

CIFAR10V1 Dataset.

Loads CIFAR10V1 images with mean and std-dev normalisation. Performs random crop and random horizontal flip on train and only normalisation on val. Based on *torchvision.datasets.CIFAR10* and Pytorch CIFAR 10 Example.

Parameters

- **root** (*str*) – Root folder for the dataset
- **train** (*bool*) – Whether to get the train or validation set (default=True)
- **download** (*bool*) – Whether to download the dataset if it's not present

Imagenet

class mlbench_core.dataset.imagerecognition.pytorch.dataloader.**Imagenet**(*root*, *train=True*)
Imagenet (ILSVRC2017) Dataset.

Loads Imagenet images with mean and std-dev normalisation. Performs random crop and random horizontal flip on train and resize + center crop on val. Based on *torchvision.datasets.ImageFolder*

Parameters

- **root** (*str*) – Root folder of Imagenet dataset (without *train/* or *val/*)
- **train** (*bool*) – Whether to get the train or validation set (default=True)

tensorflow

DatasetCifar

class mlbench_core.dataset.imagerecognition.tensorflow.**DatasetCifar**(*dataset*, *dataset_root*, *batch_size*, *world_size*, *rank*, *seed*, *tf_dtype=tf.float32*)
This clas is adapted from the following script https://github.com/tensorflow/models/blob/master/tutorials/image/cifar10/cifar10_input.py

Parameters

- **dataset** (*str*) – Name of the dataset e.g. *cifar-10*, *cifar-100*.
- **dataset_root** (*str*) – Root directory to the dataset.
- **batch_size** (*int*) – Size of batch.
- **world_size** (*int*) – Size of the world size.
- **rank** (*int*) – Rank of the process.
- **seed** (*int*) – Seed of random number.
- **tf_dtype** (*tensorflow.python.framework.dtypes.DType, optional*) – Defaults to tf.float32. Datatypes of the tensor.

input_fn(*self*, *is_train*, *repeat_count=- 1*, *num_shards=1*, *shard_index=0*)
Input_fn using the tf.data input pipeline for CIFAR-10 dataset.

In synchronized training, faster nodes may use more batches than the number of batches availble. Thus repeat dataset for engouh times to avoid throwing error.

In the distributed settings, datasets are split into *num_shards* non-overlapping parts and each process takes one shard by its index.

Parameters

- **is_train** (*bool*) – A boolean denoting whether the input is for training.
- **repeat_count** (*int*) – Defaults to -1. Count of dataset repeated times with -1 for infinite.

- **num_shards** (*int*) – Defaults to 1. Number of Shards the dataset is splitted.
- **shard_index** (*int*) – Defaults to 0. Index of shard to use.

Returns tf.data.Dataset object of *((inputs, labels), is_train)*.

maybe_download_and_extract(*self*)
Download and extract the tarball from Alex's website.

parse_record(*self*, *raw_record*)
Parse CIFAR-10/100 image and label from a raw record.

preprocess_image(*self*, *image*, *is_training*)
Preprocess a single image of layout [height, width, depth].

record_dataset(*self*, *filenames*)
Returns an input pipeline Dataset from *filenames*.

1.5.3 NLP

pytorch

Translation WMT16

class mlbench_core.dataset.nlp.pytorch.**WMT16Dataset**(*root*, *lang=('en', 'de')*, *math_precision=None*, *download=True*, *train=False*, *validation=False*, *lazy=False*, *preprocessed=False*, *sort=False*, *min_len=0*, *max_len=None*, *max_size=None*)

Dataset for WMT16 en to de translation

Parameters

- **root** (*str*) – Root folder where to download files
- **lang** (*tuple*) – Language translation pair
- **math_precision** (*str*) – One of *fp16* or *fp32*. The precision used during training
- **download** (*bool*) – Download the dataset from source
- **train** (*bool*) – Load train set
- **validation** (*bool*) – Load validation set
- **lazy** (*bool*) – Load the dataset in a lazy format
- **min_len** (*int*) – Minimum sentence length
- **max_len** (*int | None*) – Maximum sentence length
- **max_size** (*int | None*) – Maximum dataset size

class mlbench_core.dataset.nlp.pytorch.wmt16.wmt16_tokenizer.**WMT16Tokenizer**(*base_dir*, *math_precision=None*, *separator='@@'*)

Tokenizer Class for WMT16 that uses the whole vocabulary

Parameters

- **base_dir** (*str*) – Base directory for files

- **math_precision** (*str*) – Math precision
- **separator** (*str*) – BPE

detokenize(*self*, *inputs*, *delim=' '*)
Detokenizes single sentence and removes token separator characters.

Parameters

- **inputs** – sequence of tokens
- **delim** – tokenization delimiter

returns: string representing detokenized sentence

segment(*self*, *line*)
Tokenizes single sentence and adds special BOS and EOS tokens.

Parameters **line** – sentence

returns: list representing tokenized sentence

Translation WMT17

class mlbench_core.dataset.nlp.pytorch.wmt17.**Dictionary**(*pad='<pad>_'*, *eos='<EOS>_'*)
Dictionary Class for WMT17 Dataset. Essentially a mapping from symbols to consecutive integers

Parameters

- **pad** (*str*) – Padding symbol to use
- **eos** (*str*) – End of String symbol to use

add_symbol(*self*, *word*, *n=1*)
Adds a word to the dictionary

eos(*self*)
Helper to get index of end-of-sentence symbol

index(*self*, *sym*)
Returns the index of the specified symbol

classmethod load(*cls*, *f*, *ignore_utf_errors=False*)
Loads the dictionary from a text file with the format:

` <symbol0> <symbol1> ... `

Parameters

- **f** (*str*) – Dictionary file name
- **ignore_utf_errors** (*bool*) – Ignore UTF-8 related errors

pad(*self*)
Helper to get index of pad symbol

string(*self*, *tensor*, *bpe_symbol=None*)
Helper for converting a tensor of token indices to a string.

Can optionally remove BPE symbols or escape <unk> words.

update(*self*, *new_dict*)
Updates counts from new dictionary.

mlbench_core.dataset.nlp.pytorch.wmt17.**collate_batch**(*samples*, *pad_idx*, *eos_idx*, *left_pad_source=True*, *left_pad_target=False*, *bsz_mult=8*, *seq_len_multiple=1*)

Collate a list of samples into a batch

Parameters

- **samples** (*list[dict]*) – Samples to collate
- **pad_idx** (*int*) – Padding symbol index
- **eos_idx** (*int*) – EOS symbol index
- **left_pad_source** (*bool*) – Pad sources on the left
- **left_pad_target** (*bool*) – Pad sources on the right
- **bsz_mult** (*int*) – Batch size multiple
- **seq_len_multiple** (*int*) – Sequence length multiple

Returns Containing keys *id* (list of indices), *ntokens* (total num tokens), *net_input* and *target*

Return type (dict)

Language Modeling WikiText2

1.6 mlbench_core.evaluation

1.6.1 pytorch

criterion

Customized Loss Functions.

BCELossRegularized

class mlbench_core.evaluation.pytorch.criterion.**BCELossRegularized**(*weight=None*, *size_average=None*, *reduce=None*, *l1=0.0*, *l2=0.0*, *model=None*, *reduction='mean'*)

Binary Cross Entropy (BCE) with l1/l2 regularization.

Parameters

- **weight** (*Tensor, optional*) – a manual rescaling weight given to each class. If given, it has to be a Tensor of size *C*. Otherwise, it is treated as if having all ones.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`).
- **default,** (*By*) – the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By

- **the** (*default,*) – losses are averaged or summed over observations for each minibatch depending on size_average. When reduce is False, returns a loss per batch element instead and ignores size_average. Default: True
- **l1** (*float, optional*) – The scale of the L1 regularization. Default:
- **0.0** –
- **l2** (*float, optional*) – The scale of the L2 regularization. Default:
- **0.0** –
- **model** (torch.nn.Module) – a pytorch model to be trained and
- **validated.** –
- **reduction** (*string, optional*) – Specifies the reduction to apply to
- **output** (*the*) – 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Note: size_average and reduce are in the process of being deprecated, and in the meantime, specifying either of those two args will override reduction. Default: 'mean'

MSELossRegularized

class mlbench_core.evaluation.pytorch.criterion.**MSELossRegularized**(*weight=None*, *size_average=None*, *reduce=None*, *l1=0.0*, *l2=0.0*, *model=None*, *reduction='mean'*)

Mean Squared Error (MSE) with l1/l2 regularization.

Parameters

- **weight** (*Tensor, optional*) – a manual rescaling weight given to each class. If given, it has to be a Tensor of size *C*. Otherwise, it is treated as if having all ones.
- **size_average** (*bool, optional*) – Deprecated (see reduction).
- **default,** (*By*) – the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field size_average is set to False, the losses are instead summed for each minibatch. Ignored when reduce is False. Default: True
- **reduce** (*bool, optional*) – Deprecated (see reduction). By
- **the** (*default,*) – losses are averaged or summed over observations for each minibatch depending on size_average. When reduce is False, returns a loss per batch element instead and ignores size_average. Default: True
- **l1** (*float, optional*) – The scale of the L1 regularization. Default:
- **0.0** –
- **l2** (*float, optional*) – The scale of the L2 regularization. Default:
- **0.0** –
- **model** (torch.nn.Module) – a pytorch model to be trained and
- **validated.** –

- **reduction** (*string, optional*) – Specifies the reduction to apply to
- **output** (*the*) – 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: 'mean'

class mlbench_core.evaluation.pytorch.criterion.**LabelSmoothing**(*padding_idx*, *smoothing=0.0*, *fast_xentropy=False*)

NLL loss with label smoothing.

Parameters

- **padding_idx** (*int*) – Code for padding char
- **smoothing** (*float*) – Smoothing value
- **fast_xentropy** (*bool*) – Use *apex.contrib.xentropy.SoftmaxCrossEntropyLoss*

metrics

Utilities for measuring the performance of a model.

TopKAccuracy

class mlbench_core.evaluation.pytorch.metrics.**TopKAccuracy**(*topk=1*)

Top K accuracy of an output.

Counts a prediction as correct if the target value is in the top `k` predictions, false otherwise, and returns the number of correct instances relative to total instances (0.0 to 100.0).

Parameters **topk** (*int, optional*) – The number of top predictions to consider. Default: 1

__call__(*self*, *output*, *target*)

Computes the precision@k for the specified values of k

Parameters

- **output** (`torch.Tensor`) – Predictions of a model
- **target** (`torch.Tensor`) – Target labels

Example

```
>>> m = nn.Softmax()
>>> input = torch.randn(10, 50)
>>> preds = m(input)
>>> targets = torch.randint(0, 1, (10,50))
>>> topk = TopKAccuracy(5)
>>> precision = topk(preds, targets)
```

Returns float

property name(*self*)

str: Name of this metric.

1.6.2 tensorflow

criterion

Define loss functions.

softmax_cross_entropy_with_logits_v2_l2_regularized

mlbench_core.evaluation.tensorflow.criterion.**softmax_cross_entropy_with_logits_v2_l2_regularized**(*logits*, *labels*, *l2*, *loss_filte*

Return an op for computing cross entropy with weight decay.

The *labels* are assumed to be one-hot encoded. The loss filter function excludes some tensors from computing weight decay.

Parameters

- **logits** (tf.Tensor) – input logits tensor.
- **labels** (tf.Tensor) – input one-hot encoded tensor.
- **l2** (float) – size of weight decay
- **loss_filter_fn** (callable) – filter function.

Returns a scalar tensor

Return type tf.Tensor

metrics

Define tensorflow metrics.

topk_accuracy

mlbench_core.evaluation.tensorflow.metrics.**topk_accuracy_with_logits**(*logits*, *labels*, *k=1*)

Compute the top-k accuracy of logits.

Parameters

- **logits** (tf.Tensor) – input tensor
- **labels** (tf.Tensor) – input one-hot encoded tensor.
- **k** (int, optional) – Defaults to 1. top k accuracy.

Returns a scalar tensor of the accuracy (between 0 and 1).

Return type tf.Tensor

1.7 mlbench_core.lr_scheduler

1.7.1 pytorch

LRLinearWarmUp

class mlbench_core.lr_scheduler.pytorch.lr.**LRLinearWarmUp**(*optimizer*, *init_lr*, *scaled_lr*, *warmup_duration*)

Applies linear warmup to learning rate.

At the first iteration, lr will be *initial_lr*, and will linearly increase to *scaled_lr* at iteration *warmup_duration* + *1* (i.e *warmup_duration* steps of warm-up)

In [GDollarG+17], warmup is used in order to apply the `Linear Scaling Rule`. Starting from the `base_lr`, lr gradually increases to `base_lr * scaling_factor`.

Parameters

- **init_lr** (*float*) – Initial LR at beginning of warmup
- **scaled_lr** (*float*) – LR at end of warmup
- **warmup_duration** (*float*) – Duration of warmup

MultiStepLRLinearWarmUp

class mlbench_core.lr_scheduler.pytorch.lr.**MultiStepLRLinearWarmUp**(*optimizer*, *gamma*, *milestones*, *scaled_lr*, *warmup_init_lr=0*, *warmup_duration=0*)

Multi-step Learning rate Scheduler with Linear Warm-up Period

Parameters

- **optimizer** (`torch.optim.Optimizer`) – an optimizer for the given model.
- **gamma** (*float*) – Decay factor for learning rate
- **milestones** (`list` of `int`) – The epochs/steps at which to reduce the learning rate
- **scaled_lr** (*float*) – The LR to reach after warmup
- **warmup_init_lr** (*float*) – The initial learning rate to use for the warmup epochs. Default: 0
- **warmup_duration** (*int*) – The number of epochs to perform warmup before regular lr scaling starts. Default: 0

ReduceLROnPlateauWithWarmup

class mlbench_core.lr_scheduler.pytorch.lr.**ReduceLROnPlateauWithWarmup**(*optimizer*, *warmup_init_lr*, *scaled_lr*, *warmup_epochs*, *batches_per_epoch=None*, ***kwargs*)

ReduceLROnPlateau but with a linear warm-up period.

Parameters

- **optimizer** (torch.optim.Optimizer) – an optimizer for the given model.
- **warmup_init_lr** (*float*) – LR at beginning of warm-up
- **scaled_lr** (*float*) – LR at end of warm-up
- **warmup_epochs** (*int*) – Number of epochs for warm-up
- **batches_per_epoch** (*int, optional*) – Number of batches per epoch if we want a warm-up per batch
- ****kwargs** – Arguments for ReduceLROnPlateau

batch_step(*self*)

Function to call when the warm-up is per batch.

This function will change the learning rate to `` progress = batch_idx / warmup_duration new_lr = progress * scaled_lr + (1 - progress) * warmup_init_lr ``

step(*self*, *metrics*, *epoch=None*)

Scheduler step at end of epoch.

This function will pass the arguments to ReduceLROnPlateau if the warmup is done, and call *self.batch_step* if the warm-up is per epoch, to update the LR.

Parameters **metrics** (*float*) – Current loss

SparsifiedSGDLR

class mlbench_core.lr_scheduler.pytorch.lr.**SparsifiedSGDLR**(*optimizer*, *gamma*, *l2_coef*, *shifting_param*)

Learning rate schedule for sparsifiedSGD (gamma / l2_coef * (t + shifting_param))

Parameters

- **optimizer** (torch.optim.Optimizer) – an optimizer for the given model.
- **gamma** (*float*) – The constant value in the numerator of the learning rate schedule formula
- **l2_coef** (*float*) – The regularization rate which is used in the denominator of the learning rate schedule formula
- **shifting_param** (*float*) – The constant value in the denominator of the learning rate schedule formula

TimeDecayLR

class mlbench_core.lr_scheduler.pytorch.lr.**TimeDecayLR**(*optimizer*, *beta*)

Time based decay learning rate schedule for SGD (alpha / (t + beta))

Parameters

- **optimizer** (torch.optim.Optimizer) – an optimizer for the given model.
- **beta** (*float*) – The constant value in the denominator of the learning rate schedule formula

Returns A learning rate scheduler (torch.optim.lr_scheduler.LambdaLR)

SQRTTimeDecayLR

class mlbench_core.lr_scheduler.pytorch.lr.**SQRTTimeDecayLR**(*optimizer*)

Time based decay learning rate schedule for SGD (alpha / sqrt(t))

Returns A learning rate scheduler (`torch.optim.lr_scheduler.LambdaLR`)

ExponentialWarmupMultiStepLR

class mlbench_core.lr_scheduler.pytorch.lr.**ExponentialWarmupMultiStepLR**(*optimizer*, *iterations*, *warmup_steps=0*, *remain_steps=1.0*, *decay_interval=None*, *decay_steps=4*, *decay_factor=0.5*)

Learning rate scheduler with exponential warmup and step decay.

Parameters: warmup_steps, remain_steps and decay_interval accept both integers and floats as an input. Integer input is interpreted as absolute index of iteration, float input is interpreted as a fraction of total training iterations (epochs * steps_per_epoch).

If decay_interval is None then the decay will happen at regulary spaced intervals ('decay_steps' decays between iteration indices 'remain_steps' and 'iterations').

Parameters

- **optimizer** – instance of optimizer
- **iterations** (*int*) – total number of training iterations
- **warmup_steps** (*int*) – number of warmup iterations
- **remain_steps** (*int*|*float*) – start decay at 'remain_steps' iteration
- **decay_interval** (*int*|*float*) – interval between LR decay steps
- **decay_steps** (*int*) – max number of decay steps
- **decay_factor** (*float*) – decay factor

SQRTTimeDecayLRWithWarmup

class mlbench_core.lr_scheduler.pytorch.lr.**SQRTTimeDecayLRWithWarmup**(*optimizer*, *base_lr*, *warmup_init_lr*, *warmup_steps*)

SQRT learning rate scheduler with Linear warm-up steps

During warmup: ` lrs = torch.linspace(warmup_init_lr, base_lr, warmup_steps)
lr = lrs[update_num] `

After warmup: ` lr = base_lr * decay_factor `

where `decay_factor = sqrt(warmup_steps / current_iteration)`

Parameters

- **optimizer** (`torch.optim`) – The optimizer
- **base_lr** (*float*) – The base LR after warm-up
- **warmup_init_lr** (*float*) – LR at start of training

- **warmup_steps** (*int*) – Number of warm-up steps

1.7.2 tensorflow

manual_stepping

mlbench_core.lr_scheduler.tensorflow.**manual_stepping**(*global_step*, *boundaries*, *rates*, *warmup=False*)
Manually stepped learning rate schedule.

This function provides fine grained control over learning rates. One must specify a sequence of learning rates as well as a set of integer steps at which the current learning rate must transition to the next. For example, if boundaries = [5, 10] and rates = [.1, .01, .001], then the learning rate returned by this function is .1 for global_step=0,. . . ,4, .01 for global_step=5. . . 9, and .001 for global_step=10 and onward.

Parameters

- **global_step** (tf.Tensor) – int64 (scalar) tensor representing global step.
- **boundaries** (*list*) – a list of global steps at which to switch learning
- **rates** (*list*) – a list of (float) learning rates corresponding to intervals between the boundaries. The length of this list must be exactly len(boundaries) + 1.
- **warmup** (*bool, optional*) – Defaults to False. Whether to linearly interpolate learning rate for steps in [0, boundaries[0]].

Raises

- **ValueError** – boundaries is a strictly increasing list of positive integers
- **ValueError** – len(rates) == len(boundaries) + 1
- **ValueError** – boundaries[0] != 0

Returns a (scalar) float tensor representing learning rate

Return type tf.Tensor

References

1.8 mlbench_core.models

1.8.1 pytorch

Since *Kuang Liu<https://github.com/kuangliu/pytorch-cifar>* has already included many classical neural network models. We use their implementation direclty for

- VGG

linear_models

LogisticRegression

class mlbench_core.models.pytorch.linear_models.**LogisticRegression**(*n_features*)
Logistic regression implementation

Parameters **n_features** (*int*) – Number of features

LinearRegression

class mlbench_core.models.pytorch.linear_models.**LinearRegression**(*n_features*)
Ridge regression implementation

Parameters **n_features** (*int*) – Number of features

resnet

Contains definitions for Residual Networks.

Residual networks were originally proposed in [HZRS16a] . Then they improve the [HZRS16b] Here we refer to the settings in [HZRS16a] as *v1* and [HZRS16b] as *v2*.

Since torchvision resnet has already implemented.

- ResNet-18
- ResNet-34
- ResNet-50
- ResNet-101
- ResNet-152

for image net. Here we only implemented the remaining models

- ResNet-20
- ResNet-32
- ResNet-44
- ResNet-56

for CIFAR-10 dataset. Besides, their implementation uses projection shortcut by default.

ResNetCIFAR

class mlbench_core.models.pytorch.resnet.**ResNetCIFAR**(*resnet_size*, *bottleneck*, *num_classes*, *version=_DEFAULT_RESNETCIFAR_VERSION*)
Basic ResNet implementation.

Parameters

- **resnet_size** (*int*) – Number of layers
- **bottleneck** (*bool*) – Whether to use a bottleneck layer (`Not Implemented`)
- **num_classes** (*int*) – Number of output classes

- **version** (*int*) – Resnet version (1 or 2). Default: 1

RNN

—

Google Neural Machine Translation

Model

class mlbench_core.models.pytorch.gnmt.**GNMT**(*vocab_size*, *hidden_size=1024*, *num_layers=4*, *dropout=0.2*, *share_embedding=True*, *fusion=True*)

GNMT v2 model

Parameters

- **vocab_size** (*int*) – size of vocabulary (number of tokens)
- **hidden_size** (*int*) – internal hidden size of the model
- **num_layers** (*int*) – number of layers, applies to both encoder and decoder
- **dropout** (*float*) – probability of dropout (in encoder and decoder) tensors, if false the model uses (seq, batch, feature)
- **share_embedding** (*bool*) – if True embeddings are shared between encoder and decoder

decode(*self*, *inputs*, *context*, *inference=False*)

Applies the decoder to inputs, given the context from the encoder.

Parameters

- **inputs** (*torch.tensor*) – tensor with inputs (seq_len, batch)
- **context** – context from the encoder
- **inference** – if True inference mode, if False training mode

Returns torch.tensor

encode(*self*, *inputs*, *lengths*)

Applies the encoder to inputs with a given input sequence lengths.

Parameters

- **inputs** (*torch.tensor*) – tensor with inputs (seq_len, batch)
- **lengths** – vector with sequence lengths (excluding padding)

Returns torch.tensor

generate(*self*, *inputs*, *context*, *beam_size*)

Autoregressive generator, works with SequenceGenerator class. Executes decoder (in inference mode), applies log_softmax and topK for inference with beam search decoding.

Parameters

- **inputs** – tensor with inputs to the decoder
- **context** – context from the encoder
- **beam_size** – beam size for the generator

Returns (words, logprobs, scores, new_context) words: indices of topK tokens logprobs: log probabilities of topK tokens scores: scores from the attention module (for coverage penalty) new_context: new decoder context, includes new hidden states for decoder RNN cells

BahdanauAttention

Encoder

class mlbench_core.models.pytorch.gnmt.encoder.**ResidualRecurrentEncoder**(*vocab_size*, *hidden_size=1024*, *num_layers=4*, *dropout=0.2*, *embedder=None*, *init_weight=0.1*)

Encoder with Embedding, LSTM layers, residual connections and optional dropout.

The first LSTM layer is bidirectional and uses variable sequence length API, the remaining (num_layers-1) layers are unidirectional. Residual connections are enabled after third LSTM layer, dropout is applied on inputs to LSTM layers.

Parameters

- **vocab_size** – size of vocabulary
- **hidden_size** – hidden size for LSTM layers
- **num_layers** – number of LSTM layers, 1st layer is bidirectional
- **dropout** – probability of dropout (on input to LSTM layers)
- **embedder** – instance of nn.Embedding, if None constructor will create new embedding layer
- **init_weight** – range for the uniform initializer

forward(*self*, *inputs*, *lengths*)

Execute the encoder.

Parameters

- **inputs** – tensor with indices from the vocabulary
- **lengths** – vector with sequence lengths (excluding padding)

Returns tensor with encoded sequences

Decoder

class mlbench_core.models.pytorch.gnmt.decoder.**RecurrentAttention**(*input_size=1024*, *context_size=1024*, *hidden_size=1024*, *num_layers=1*, *dropout=0.2*, *init_weight=0.1*, *fusion=True*)

LSTM wrapped with an attention module.

Parameters

- **input_size** (*int*) – number of features in input tensor
- **context_size** (*int*) – number of features in output from encoder

- **hidden_size** (*int*) – internal hidden size
- **num_layers** (*int*) – number of layers in LSTM
- **dropout** (*float*) – probability of dropout (on input to LSTM layer)
- **init_weight** (*float*) – range for the uniform initializer

forward(*self*, *inputs*, *hidden*, *context*, *context_len*)
Execute RecurrentAttention.

Parameters

- **inputs** (*int*) – tensor with inputs
- **hidden** (*int*) – hidden state for LSTM layer
- **context** – context tensor from encoder
- **context_len** – vector of encoder sequence lengths

Returns (rnn_outputs, hidden, attn_output, attn_scores)

class mlbench_core.models.pytorch.gnmt.decoder.**Classifier**(*in_features*, *out_features*, *init_weight=0.1*)

Fully-connected classifier

Parameters

- **in_features** (*int*) – number of input features
- **out_features** (*int*) – number of output features (size of vocabulary)
- **init_weight** (*float*) – range for the uniform initializer

forward(*self*, *x*)
Execute the classifier.

Parameters **x** (*torch.tensor*) –

Returns torch.tensor

class mlbench_core.models.pytorch.gnmt.decoder.**ResidualRecurrentDecoder**(*vocab_size*, *hidden_size=1024*, *num_layers=4*, *dropout=0.2*, *embedder=None*, *init_weight=0.1*, *fusion=True*)

Decoder with Embedding, LSTM layers, attention, residual connections and optinal dropout.

Attention implemented in this module is different than the attention discussed in the GNMT arxiv paper. In this model the output from the first LSTM layer of the decoder goes into the attention module, then the re-weighted context is concatenated with inputs to all subsequent LSTM layers in the decoder at the current timestep.

Residual connections are enabled after 3rd LSTM layer, dropout is applied on inputs to LSTM layers.

Parameters

- **vocab_size** (*int*) – size of vocabulary
- **hidden_size** (*int*) – hidden size for LSMT layers
- **num_layers** (*int*) – number of LSTM layers
- **dropout** (*float*) – probability of dropout (on input to LSTM layers)

- **embedder** (*nn.Embedding*) – if None constructor will create new embedding layer
- **init_weight** (*float*) – range for the uniform initializer

append_hidden(*self*, *h*)
Appends the hidden vector h to the list of internal hidden states.

Parameters **h** – hidden vector

forward(*self*, *inputs*, *context*, *inference=False*)
Execute the decoder.

Parameters

- **inputs** – tensor with inputs to the decoder
- **context** – state of encoder, encoder sequence lengths and hidden state of decoder's LSTM layers
- **inference** – if True stores and repackages hidden state

Returns:

init_hidden(*self*, *hidden*)
Converts flattened hidden state (from sequence generator) into a tuple of hidden states. :param hidden: None or flattened hidden state for decoder RNN layers

package_hidden(*self*)
Flattens the hidden state from all LSTM layers into one tensor (for the sequence generator).

Transformer Model for Translation

Model

class mlbench_core.models.pytorch.transformer.**TransformerModel**(*args*, *src_dict*, *trg_dict*)
Transformer model

This model uses MultiHeadAttention as described in [VSP+17]

Parameters

- **args** – Arguments of model. All arguments should be accessible via *__getattribute__* method
- **src_dict** (*mlbench_core.dataset.nlp.pytorch.wmt17.Dictionary*) – Source dictionary
- **trg_dict** (*mlbench_core.dataset.nlp.pytorch.wmt17.Dictionary*) – Target dictionary

forward(*self*, *src_tokens*, *src_lengths*, *prev_output_tokens*)
Run the forward pass of the transformer model.

Parameters

- **src_tokens** (torch.Tensor) – Source tokens
- **src_lengths** (torch.Tensor) – Source sentence lengths
- **prev_output_tokens** (torch.Tensor) – Previous output tokens

Returns The model output, and attention weights if needed

Return type (torch.Tensor, Optional[torch.Tensor])

Encoder

class mlbench_core.models.pytorch.transformer.encoder.**TransformerEncoder**(*args*, *dictionary*, *embed_tokens*, *left_pad=True*)

Transformer encoder consisting of *args.encoder_layers* layers. Each layer is a `TransformerEncoderLayer`.

Parameters

- **args** – Arguments of model. All arguments should be accessible via *__getattribute__* method
- **dictionary** (*mlbench_core.dataset.nlp.pytorch.wmt17.Dictionary*) – encoding dictionary
- **embed_tokens** (*torch.nn.Embedding*) – input embedding
- **left_pad** (*bool*) – Pad sources to the left (*True*) or right (*False*). Default: *True*

forward(*self*, *src_tokens*)

Forward function of encoder

Parameters **src_tokens** (`torch.Tensor`) – Source tokens

Returns {*encoder:out* (`torch.Tensor`), *encoder_padding_mask* (`torch.Tensor`)}

Return type (dict)

Decoder

class mlbench_core.models.pytorch.transformer.decoder.**TransformerDecoder**(*args*, *dictionary*, *embed_tokens*, *no_encoder_attn=False*, *left_pad=False*)

Transformer decoder consisting of *args.decoder_layers* layers. Each layer is a `TransformerDecoderLayer`.

Parameters

- **args** – Arguments of model. All arguments should be accessible via *__getattribute__* method
- **dictionary** (*mlbench_core.dataset.nlp.pytorch.wmt17.Dictionary*) – decoding dictionary
- **embed_tokens** (*torch.nn.Embedding*) – output embedding
- **no_encoder_attn** (*bool, optional*) – whether to attend to encoder outputs (default: False).
- **left_pad** (*bool*) – Pad targets to the left (*True*) or right (*False*). Default: *False*

Layers

class mlbench_core.models.pytorch.transformer.modules.**TransformerEncoderLayer**(*args*)

Encoder layer block.

In the original paper each operation (multi-head attention or FFN) is postprocessed with: *dropout -> add residual -> layernorm*. In the tensor2tensor code they suggest that learning is more robust when preprocessing each layer with layernorm and postprocessing with: *dropout -> add residual*. We default to the approach in the paper, but the tensor2tensor approach can be enabled by setting *args.encoder_normalize_before* to `True`.

Parameters **args** (*argparse.Namespace*) – parsed command-line arguments

class mlbench_core.models.pytorch.transformer.modules.**TransformerDecoderLayer**(*args*, *no_encoder_attn=False*)

Decoder layer block.

In the original paper each operation (multi-head attention, encoder attention or FFN) is postprocessed with: *dropout -> add residual -> layernorm*. In the tensor2tensor code they suggest that learning is more robust when preprocessing each layer with layernorm and postprocessing with: *dropout -> add residual*. We default to the approach in the paper, but the tensor2tensor approach can be enabled by setting *args.decoder_normalize_before* to `True`.

Parameters

- **args** (*argparse.Namespace*) – parsed command-line arguments
- **no_encoder_attn** (*bool, optional*) – whether to attend to encoder outputs (default: False).

SequenceGenerator

class mlbench_core.models.pytorch.transformer.sequence_generator.**SequenceGenerator**(*model*, *src_dict*, *trg_dict*, *beam_size=1*, *minlen=1*, *maxlen=None*, *stop_early=True*, *normalize_scores=True*, *len_penalty=1*, *retain_dropout=False*, *sampling=False*, *sampling_topk=-1*, *sampling_temperature=1*)

Generates translations of a given source sentence.

Parameters

- **model** (`torch.nn.Module`) – The model to predict on. Should be instance of *TransformerModel*
- **src_dict** (*mlbench_core.dataset.nlp.pytorch.wmt17.Dictionary*) – Source dictionary

- **trg_dict** (*mlbench_core.dataset.nlp.pytorch.wmt17.Dictionary*) – Target dictionary
- **beam_size** (*int*) – Size of the beam. Default 1
- **minlen** (*int*) – Minimum generation length. Default 1
- **maxlen** (*int*) – Maximum generation length. If *None*, takes value of model.max_decoder_positions(). Default *None*
- **stop_early** (*bool*) – Stop generation immediately after we finalize beam_size hypotheses, even though longer hypotheses might have better normalized scores. Default *True*
- **normalize_scores** (*bool*) – Normalize scores by the length of the output. Default *True*
- **len_penalty** (*float*) – length penalty: <1.0 favors shorter, >1.0 favors longer sentences. Default 1
- **retain_dropout** (*bool*) – Keep dropout layers. Default *False*
- **sampling** (*bool*) – sample hypotheses instead of using beam search. Default *False*
- **sampling_topk** (*int*) – sample from top K likely next words instead of all words. Default -1
- **sampling_temperature** (*int*) – temperature for random sampling. Default 1

generate(*self*, *src_tokens*, *src_lengths*, *maxlen=None*, *prefix_tokens=None*)
Generate a batch of translations.

generate_batch_translations(*self*, *batch*, *maxlen_a=0.0*, *maxlen_b=None*, *prefix_size=0*)
Yield individual translations of a batch.

Parameters

- **batch** (*dict*) – The model input batch. Must have keys *net_input*, *target* and *ntokens*
- **maxlen_a** (*float*) –
- **maxlen_b** (*Optional[int]*) – Generate sequences of max lengths *maxlen_a*x* + *maxlen_b* where *x* = *input sentence length*
- **prefix_size** (*int*) – Prefix size

translate_batch(*self*, *batch*, *maxlen_a=1.0*, *maxlen_b=50*, *prefix_size=0*, *remove_bpe=None*, *nbest=1*, *ignore_case=True*)

Parameters

- **batch** (*dict*) – The model input batch. Must have keys *net_input*, *target* and *ntokens*
- **maxlen_a** (*float*) – Default 1.0
- **maxlen_b** (*Optional[int]*) – Generate sequences of max lengths *maxlen_a*x* + *maxlen_b* where *x* = *input sentence length*. Default 50
- **prefix_size** (*int*) – Prefix size. Default 0
- **remove_bpe** (*Optional[str]*) – BPE token. Default *None*
- **nbest** (*int*) – Number of hypotheses to output. Default 1
- **ignore_case** (*bool*) – Ignore case druing online eval. Default *True*

Returns The translations and their targets for the given batch

Return type (list[str], list[str])

References

NLP

LSTM Language Model

1.8.2 tensorflow

resnet

Contains definitions for Residual Networks. Residual networks ('v1' ResNets) were originally proposed in [HZRS16a]. The full preactivation 'v2' ResNet variant was introduced by [HZRS16b]. The key difference of the full preactivation 'v2' variant compared to the 'v1' variant in [1] is the use of batch normalization before every weight layer rather than after.

mlbench_core.models.tensorflow.resnet_model.**fixed_padding**(*inputs*, *kernel_size*, *data_format*)

Pads the input along the spatial dimensions independently of input size.

Parameters

- **inputs** (tf.Tensor) – A tensor of size [batch, channels, height_in, width_in] or [batch, height_in, width_in, channels] depending on data_format.
- **kernel_size** (*int*) – The kernel to be used in the conv2d or max_pool2d operation. Should be a positive integer.
- **data_format** (*str*) – The input format ('channels_last' or 'channels_first').

Returns A tensor with the same format as the input with the data either intact (if kernel_size == 1) or padded (if kernel_size > 1).

mlbench_core.models.tensorflow.resnet_model.**conv2d_fixed_padding**(*inputs*, *filters*, *kernel_size*, *strides*, *data_format*)

Strided 2-D convolution with explicit padding.

mlbench_core.models.tensorflow.resnet_model.**block_layer**(*inputs*, *filters*, *bottleneck*, *block_fn*, *blocks*, *strides*, *training*, *name*, *data_format*)

Creates one layer of blocks for the ResNet model.

Parameters

- **inputs** (tf.Tensor) – A tensor of size [batch, channels, height_in, width_in] or [batch, height_in, width_in, channels] depending on data_format.
- **filters** (*int*) – The number of filters for the first convolution of the layer.
- **bottleneck** (*bool*) – Is the block created a bottleneck block.
- **block_fn** (*callable*) – The block to use within the model, either *building_block* or *bottleneck_block*.
- **blocks** (*int*) – The number of blocks contained in the layer.
- **strides** (*int*) – The stride to use for the first convolution of the layer. If greater than 1, this layer will ultimately downsample the input.
- **training** (*bool*) – Either True or False, whether we are currently training the model. Needed for batch norm.

- **name** (*str*) – A string name for the tensor output of the block layer.
- **data_format** (*str*) – The input format ('channels_last' or 'channels_first').

Returns The output tensor of the block layer.

mlbench_core.models.tensorflow.resnet_model.**batch_norm**(*inputs*, *training*, *data_format*)
Performs a batch normalization using a standard set of parameters.

Model

class mlbench_core.models.tensorflow.resnet_model.**Model**(*resnet_size*, *bottleneck*, *num_classes*, *num_filters*, *kernel_size*, *conv_stride*, *first_pool_size*, *first_pool_stride*, *block_sizes*, *block_strides*, *resnet_version=DEFAULT_VERSION*, *data_format=None*, *dtype=DEFAULT_DTYPE*)
Base class for building the Resnet Model.

Cifar10Model

class mlbench_core.models.tensorflow.resnet_model.**Cifar10Model**(*resnet_size*, *data_format=None*, *num_classes=10*, *resnet_version=2*, *dtype=tf.float32*)
Model class with appropriate defaults for CIFAR-10 data.

1.9 mlbench_core.optim

1.9.1 pytorch

Optimizers

The optimizers in this module are not distributed. Their purpose is to implement logic that can be inherited by distributed optimizers.

SparsifiedSGD

class mlbench_core.optim.pytorch.optim.**SparsifiedSGD**(*params*, *lr=required*, *weight_decay=0*, *sparse_grad_size=10*)
Implements sparsified version of stochastic gradient descent.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **sparse_grad_size** (*int*) – Size of the sparsified gradients vector (default: 10).

get_estimated_weights(*self*)
Returns the weighted average parameter tensor

sparsify_gradients(*self*, *param*, *lr*)
Calls one of the sparsification functions (random or blockwise)

Parameters

- **random_sparse** (*bool*) – Indicates the way we want to make the gradients sparse (random or blockwise) (default: False)
- **(** (*param*) – obj: *torch.nn.Parameter*): Model parameter

step(*self*, *closure=None*)
Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

update_estimated_weights(*self*, *iteration*, *sparse_vector_size*)
Updates the estimated parameters

Parameters

- **iteration** (*int*) – Current global iteration
- **sparse_vector_size** (*int*) – Size of the sparse gradients vector

SignSGD

class mlbench_core.optim.pytorch.optim.**SignSGD**(*params*, *lr*, *momentum=0*, *weight_decay=0*, *dampening=0*, *nesterov=False*)
Implements sign stochastic gradient descent (optionally with momentum).

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

step(*self*, *closure=None*)
Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

Centralized (Synchronous) Optimizers

The optimizers in this module are all distributed and synchronous: workers advance in a synchronous manner. All workers communicate with each other using *all_reduce* or *all_gather* operations.

Generic Centralized Optimizer

class mlbench_core.optim.pytorch.centralized.**GenericCentralizedOptimizer**(*world_size*, *model*, *use_cuda=False*, *by_layer=False*, *divide_before=False*, *agg_grad=True*)

Implements a generic centralized (synchronous) optimizer with *AllReduceAggregation*. Averages the reduced parameters over the world size, after aggregation. Can aggregate gradients or weights, by layers or all at once.

Parameters

- **world_size** (*int*) – Size of the network
- **model** (nn.Module) – Model which contains parameters for SGD
- **use_cuda** (*bool*) – Whether to use cuda tensors for aggregation
- **by_layer** (*bool*) – Aggregate by layer instead of all layers at once
- **agg_grad** (*bool*) – Aggregate the gradients before updating weights. If *False*, weights will be updated and then reduced across all workers. (default: *True*)

step(*self*, *closure=None*, *tracker=None*)

Aggregates the gradients and performs a single optimization step.

Parameters

- **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.
- **tracker** (mlbench_core.utils.Tracker, optional) –

CentralizedSGD

class mlbench_core.optim.pytorch.centralized.**CentralizedSGD**(*world_size=None*, *model=None*, *lr=required*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *use_cuda=False*, *by_layer=False*, *agg_grad=True*)

Bases: *GenericCentralizedOptimizer*

Implements centralized stochastic gradient descent (optionally with momentum). Averages the reduced parameters over the world size.

Parameters

- **world_size** (*int*) – Size of the network
- **model** (nn.Module) – Model which contains parameters for SGD
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)
- **use_cuda** (*bool*) – Whether to use cuda tensors for aggregation
- **by_layer** (*bool*) – Aggregate by layer instead of all layers at once
- **agg_grad** (*bool*) – Aggregate the gradients before updating weights. If *False*, weights will be updated and then reduced across all workers. (default: *True*)

CentralizedAdam

class mlbench_core.optim.pytorch.centralized.**CentralizedAdam**(*world_size=None*, *model=None*, *lr=required*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, *use_cuda=False*, *by_layer=False*, *agg_grad=True*)

Bases: *GenericCentralizedOptimizer*

Implements centralized Adam algorithm. Averages the reduced parameters over the world size

Parameters

- **world_size** (*int*) – Size of the network
- **model** (nn.Module) – Model which contains parameters for Adam
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **amsgrad** (*boolean, optional*) – whether to use the AMSGrad variant of this algorithm from the paper [RKK18] (default: False)
- **use_cuda** (*bool*) – Whether to use cuda tensors for aggregation
- **by_layer** (*bool*) – Aggregate by layer instead of all layers at once

CustomCentralizedOptimizer

class mlbench_core.optim.pytorch.centralized.**CustomCentralizedOptimizer**(*model*, *world_size*, *optimizer*, *use_cuda=False*, *by_layer=False*, *agg_grad=True*, *grad_clip=float('inf')*, *average_world=False*, *average_custom=False*, *divide_before=False*)

Bases: *GenericCentralizedOptimizer*

Custom Centralized Optimizer. Can be used with any optimizer passed as argument. Adds the gradient clipping option, as well as custom average

Parameters

- **model** (torch.nn.Module) – model
- **world_size** (*int*) – Distributed world size
- **use_cuda** (*bool*) – Use cuda tensors for aggregation
- **by_layer** (*bool*) – Aggregate by layer
- **agg_grad** (*bool*) – Aggregate the gradients before updating weights. If *False*, weights will be updated and then reduced across all workers. (default: *True*)
- **grad_clip** (*float*) – coefficient for gradient clipping, max L2 norm of the gradients
- **average_world** (*bool*) – Average the gradients by world size
- **average_custom** (*bool*) – Divide gradients by given denominator at each step, instead of *world_size*
- **divide_before** (*bool*) – Divide gradients before reduction (default: False)

step(*self*, *closure=None*, *tracker=None*, *denom=None*)

Performs one step of the optimizer.

Parameters

- **closure** (*callable*) – Optimizer closure argument
- **tracker** (mlbench_core.utils.Tracker, optional) –
- **denom** (Optional[torch.Tensor]) – Custom denominator to reduce by

CentralizedSparsifiedSGD

class mlbench_core.optim.pytorch.centralized.**CentralizedSparsifiedSGD**(*params=None*, *lr=required*, *weight_decay=0*, *sparse_grad_size=10*, *random_sparse=False*, *world_size=1*, *average_world=True*)

Implements centralized sparsified version of stochastic gradient descent.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – Learning rate
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **sparse_grad_size** (*int*) – Size of the sparsified gradients vector (default: 10)
- **random_sparse** (*bool*) – Whether select random sparsification (default: *False*)
- **average_world** (*bool*) – Whether to average models on the world_size (default: *True*)

step(*self*, *closure=None*)

Aggregates the gradients and performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

PowerSGD

class mlbench_core.optim.pytorch.centralized.**PowerSGD**(*model=None*, *lr=required*, *momentum=0*, *weight_decay=0*, *dampening=0*, *nesterov=False*, *average_world=True*, *use_cuda=False*, *by_layer=False*, *reuse_query=False*, *world_size=1*, *rank=1*)

Implements PowerSGD with error feedback (optionally with momentum).

Parameters

- **model** (nn.Module) – Model which contains parameters for SGD
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)
- **average_world** (*bool*) – Whether to average models on the world_size (default: *True*)
- **use_cuda** (*bool*) – Whether to use cuda tensors for aggregation
- **by_layer** (*bool*) – Aggregate by layer instead of all layers at once
- **reuse_query** (*bool*) – Whether to use warm start to initialize the power iteration
- **rank** (*int*) – The rank of the gradient approximation

step(*self*, *closure=None*, *tracker=None*)

Performs a single optimization step.

Parameters

- **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.
- **tracker** (mlbench_core.utils.Tracker, optional) –

Decentralized (Asynchronous) Optimizers

The optimizers in this module are all distributed and asynchronous: workers advance independently from each other, and communication patterns follow an arbitrary graph.

DecentralizedSGD

class mlbench_core.optim.pytorch.decentralized.**DecentralizedSGD**(*rank=None*, *neighbors=None*, *model=None*, *lr=required*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *average_world=True*, *use_cuda=False*, *by_layer=False*)

Implements decentralized stochastic gradient descent (optionally with momentum).

Parameters

- **rank** (*int*) – rank of current process in the network
- **neighbors** (*list*) – list of ranks of the neighbors of current process
- **model** (nn.Module) – model which contains parameters for SGD
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)
- **average_world** (*bool*) – Whether to average models on the world_size (default: *True*)
- **use_cuda** (*bool*) – Whether to use cuda tensors for aggregation
- **by_layer** (*bool*) – Aggregate by layer instead of all layers at once

step(*self*, *closure=None*, *tracker=None*)

Aggregates the gradients and performs a single optimization step.

Parameters

- **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.
- **tracker** (mlbench_core.utils.Tracker, optional) –

References

Mixed Precision Optimizers

FP16Optimizer

class mlbench_core.optim.pytorch.fp_optimizers.**FP16Optimizer**(*fp16_model*, *world_size*, *use_cuda=False*, *use_horovod=False*, *by_layer=False*, *grad_clip=float('inf')*, *init_scale=1024*, *scale_factor=2*, *scale_window=128*, *max_scale=None*, *min_scale=0.0001*, *average_world=False*, *average_custom=False*, *divide_before=False*)

Mixed precision optimizer with dynamic loss scaling and backoff. https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#scalefactor

Parameters

- **fp16_model** (*torch.nn.Module*) – model (previously casted to half)
- **world_size** (*int*) – Distributed world size
- **use_cuda** (*bool*) – Use cuda tensors for aggregation
- **use_horovod** (*bool*) – Use Horovod for aggregation
- **by_layer** (*bool*) – Aggregate by layer
- **grad_clip** (*float*) – coefficient for gradient clipping, max L2 norm of the gradients
- **init_scale** (*int*) – initial loss scale
- **scale_factor** (*float*) – Factor for upscale/dowscale
- **scale_window** (*int*) – interval for loss scale upscaling
- **average_world** (*bool*) – Average the gradients by world size
- **average_custom** (*bool*) – Divide gradients by given denominator at each step, instead of *world_size*
- **divide_before** (*bool*) – Divide gradients before reduction (default: False)

backward_loss(*self*, *loss*)

Scales and performs backward on the given loss

Parameters **loss** (*torch.nn.Module*) – The loss

static fp16_to_fp32_flat_grad(*fp32_params*, *fp16_model*)

Copies the parameters in *fp16_model* into *fp32_params* in-place

Parameters

- **fp32_params** (*torch.Tensor*) – Parameters in fp32
- **fp16_model** (*torch.nn.Module*) – Model in fp16

static fp32_to_fp16_weights(*fp16_model*, *fp32_params*)

Copies the parameters in *fp32_params* into *fp16_model* in-place

Parameters

- **fp16_model** (*torch.nn.Module*) – Model in fp16
- **fp32_params** (*torch.Tensor*) – Parameters in fp32

initialize_flat_fp32_weight(*self*)
Initializes the model's parameters in fp32

Returns The Parameters in fp32

Return type (torch.Tensor)

step(*self*, *closure=None*, *tracker=None*, *multiplier=1*, *denom=None*)
Performs one step of the optimizer. Applies loss scaling, computes gradients in fp16, converts gradients to fp32, inverts scaling and applies optional gradient norm clipping. If gradients are finite, it applies update to fp32 master weights and copies updated parameters to fp16 model for the next iteration. If gradients are not finite, it skips the batch and adjusts scaling factor for the next iteration.

Parameters

- **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.
- **tracker** (mlbench_core.utils.Tracker, optional) –
- **multiplier** (*float*) – Multiplier for gradient scaling. Gradient will be scaled using *scaled_grad = reduced_grad / (loss_scaler * multiplier)*
- **denom** (Optional[torch.Tensor]) – Custom denominator to average by Use with *average_batch*. (default: *None*)

zero_grad(*self*)
Resets the gradients of the optimizer and fp16_model

1.10 mlbench_core.utils

1.10.1 pytorch

FCGraph

class mlbench_core.utils.pytorch.**FCGraph**(*rank*, *world_size*, *use_cuda=False*)
Fully-Connected Network Graph

Parameters **config** (*dict*) – a global object containing all of the config.

initialize_backends

mlbench_core.utils.pytorch.**initialize_backends**(*comm_backend='mpi'*, *hosts=None*, *rank=- 1*, *logging_level='INFO'*, *logging_file='/mlbench.log'*, *use_cuda=False*, *seed=None*, *cudnn_deterministic=False*, *ckpt_run_dir='/checkpoints'*, *delete_existing_ckpts=False*)
Initializes the backends.

Sets up logging, sets up pytorch and configures paths correctly.

Parameters **config** (types.SimpleNamespace) – a global object containing all of the config.

Returns a global object containing all of the config.

Return type (types.SimpleNamespace)

Checkpointer

class mlbench_core.utils.pytorch.checkpoint.**Checkpointer**(*ckpt_run_dir*, *rank*, *freq=CheckpointFreq.BEST*, *save_stats=True*)

A class for handling checkpoint saving and loading.

Parameters

- **ckpt_run_dir** (*str*) – The path of the checkpoint directory.
- **rank** (*int*) – The rank of the eurrent worker.
- **freq** (*int*) – The frequency of checkpointing. Default: *CheckpointFreq.BEST*
- **save_stats** (*bool*) – Save stats to additional text files. Default: *True*

helpers

Helper functions.

mlbench_core.utils.pytorch.helpers.**config_logging**(*logging_level='INFO'*, *logging_file='/mlbench.log'*)

Setup logging modules. A stream handler and file handler are added to default logger *mlbench*.

Parameters

- **logging_level** (*str*) – Log level
- **logging_file** (*str*) – Log file

mlbench_core.utils.pytorch.helpers.**config_pytorch**(*use_cuda=False*, *seed=None*, *cudnn_deterministic=False*)

Config pytorch packages.

Fix random number for packages and initialize distributed environment for pytorch. Setup cuda environment for pytorch.

Parameters

- **use_cuda** (*bool*) – Use CUDA acceleration
- **seed** (*int | None*) – Random seed to use
- **cudnn_deterministic** (*bool*) – Set *cudnn.determenistic=True*

Returns FCGraph): The rank, world size, and network graph

Return type (int, int, *obj*

mlbench_core.utils.pytorch.helpers.**config_path**(*ckpt_run_dir*, *delete_existing_ckpts=False*)

Config the path used during the experiments.

utils

mlbench_core.utils.pytorch.utils.**pack_tensors**(*tensors*, *use_cuda=False*)
Packs a list of tensors into one 1-dimensional tensor.

Parameters

- **tensors** (*list[torch.Tensor]*) – The tensors to pack
- **use_cuda** (*bool*) – Whether the resulting tensor should be on cuda

Returns

The flattened tensors, the list start indices of each packed tensor, and the original shape of each tensor.

Those values are used to then unpack the tensor

Return type (torch.Tensor, list[int], list[(int, int)])

mlbench_core.utils.pytorch.utils.**unpack_tensors**(*aggregated*, *indices*, *sizes*)
Unpacks a 1-dimensional tensor into a list of tensors

Parameters

- **aggregated** (*torch.Tensor*) – The 1-dimensional tensor
- **indices** (*List[Int]*) – The start index of each tensor
- **sizes** (*List[(Int, Int)]*) – The size of each resulting tensor

Returns The unpacked tensors

Return type List[torch.Tensor]

mlbench_core.utils.pytorch.utils.**orthogonalize**(*matrix*, *eps=torch.FloatTensor([1e-16])*)
Function used to orthogonalize a matrix.

Parameters

- **matrix** (*torch.Tensor*) – Matrix to orthogonalize
- **eps** (*torch.FloatTensor*) – Used to avoid division by zero (default: 1e-16)

Inference

Translator

BeamSearch

class mlbench_core.utils.pytorch.inference.beam_search.**SequenceGenerator**(*model*, *beam_size=5*, *max_seq_len=100*, *len_norm_factor=0.6*, *len_norm_const=5*, *cov_penalty_factor=0.1*)
Generator for the autoregressive inference with beam search decoding.

Beam search decoding supports coverage penalty and length normalization. For details, refer to Section 7 of the GNMT paper (https://arxiv.org/pdf/1609.08144.pdf).

Parameters

- **model** – model which implements generate method

- **beam_size** (*int*) – decoder beam size
- **max_seq_len** (*int*) – maximum decoder sequence length
- **len_norm_factor** (*float*) – length normalization factor
- **len_norm_const** (*float*) – length normalization constant
- **cov_penalty_factor** (*float*) – coverage penalty factor

beam_search(*self*, *batch_size*, *initial_input*, *initial_context=None*)
Beam Search decoder.

Parameters

- **batch_size** (*int*) – decoder batch size
- **initial_input** (*torch.tensor*) – initial input, usually tensor of BOS tokens
- **initial_context** (*torch.tensor*) – initial context, usually [encoder_context, src_seq_lengths, None]

Returns: (translation, lengths, counter) translation: (batch_size, max_seq_len) - indices of target tokens lengths: (batch_size) - lengths of generated translations counter: number of iterations of the decoding loop

greedy_search(*self*, *batch_size*, *initial_input*, *initial_context=None*)
Greedy decoder.

Parameters

- **batch_size** (*int*) – decoder batch size
- **initial_input** (*torch.tensor*) – initial input, usually tensor of BOS tokens
- **initial_context** (*torch.tensor*) – initial context, usually [encoder_context, src_seq_lengths, None]

Returns: (translation, lengths, counter) translation: (batch_size, max_seq_len) - indices of target tokens lengths: (batch_size) - lengths of generated translations counter: number of iterations of the decoding loop

1.10.2 tensorflow

Initialize environment for pytorch.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[GDollarG+17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[HZRS16a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778. 2016.

[HZRS16b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, 630–645. Springer, 2016.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017. URL: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

[RKK18] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=ryQu7f-RZ.

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

Z