DeepOBS Documentation

Release 1.2.0-beta0

Frank Schneider

Oct 29, 2019

User Guide

Warning: This DeepOBS version is under continious development and a beta of DeepOBS 1.2.0.

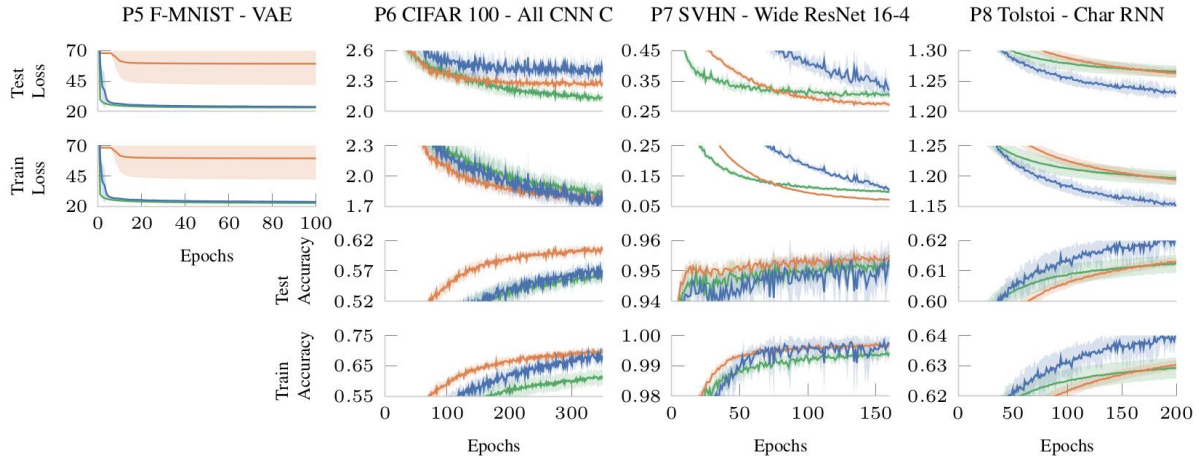
Many thanks to Aaron Bahde for spearheading the developement of DeepOBS 1.2.0.

DeepOBS is a benchmarking suite that drastically simplifies, automates and improves the evaluation of deep learning optimizers.

It can evaluate the performance of new optimizers on a variety of **real-world test problems** and automatically compare them with **realistic baselines**.

DeepOBS automates several steps when benchmarking deep learning optimizers:

- Downloading and preparing data sets.
- Setting up test problems consisting of contemporary data sets and realistic deep learning architectures.
- Running the optimizers on multiple test problems and logging relevant metrics.
- Automatic tuning of optimizer hyperparameters.
- Reporting and visualization the results of the optimizer benchmark.



The code for the current implementation working with **TensorFlow** and **PyTorch** can be found on GitHub.

CHAPTER 1

Quick Start

DeepOBS is a Python package to benchmark deep learning optimizers. It supports TensorFlow and PyTorch.

We tested the package with Python 3.6, TensorFlow version 1.12 and Torch version 1.1.0. Other versions of Python and TensorFlow (>= 1.4.0) might work, and we plan to expand compatibility in the future.

1.1 Installation

You can install the latest try-out version of DeepOBS using *pip*:

```
pip install -e git+https://github.com/abahde/DeepOBS.git@master#egg=DeepOBS
```

Note: The package requires the following packages:

- argparse
- numpy
- pandas
- matplotlib
- tikzplotlib
- seaborn

TensorFlow is not a required package to allow for both the CPU and GPU version. Make sure that one of those is installed. Additionally, you have to install torch/torchvision if you want to use the PyTorch framework.

Hint: We do not specify the exact version of the required package. However, if any problems occur while using DeepOBS, it might be a good idea to upgrade those packages to the newest release (especially matplotlib and numpy).

1.2 Set-Up Data Sets

If you use TensorFlow, you have to download the data sets for the test problems. This can be done by simply running the *Prepare Data* script:

```
deepobs_prepare_data.sh
```

This will automatically download, sort and prepare all the data sets (except ImageNet) in a folder called `data_deepobs` in the current directory. It can take a while, as it will download roughly 1 GB.

Note: The ImageNet data set is currently excluded from this automatic downloading and preprocessing. ImageNet requires a registration to do this and has a total size of hundreds of GBs. You can download it and add it to the `imagenet` folder by yourself if you wish to use the ImageNet data set.

Hint: If you already have some of the data sets on your computer, you can only download the rest. If you have all data sets, you can skip this step, and always tell DeepOBS where the data sets are located. However, the DeepOBS package requires the data sets to be organized in a specific way.

If you use PyTorch, the data downloading will be handled automatically by torchvision.

You are now ready to run different optimizers on various test problems. We provide a *Simple Example* for this, as well as our *Suggested Protocol* for benchmarking deep learning optimizers.

1.3 Contributing to DeepOBS

If you want to see a certain data set or test problem added to DeepOBS, you can just fork DeepOBS, and implemented following the structure of the existing modules and create a pull-request. We are very happy to expand DeepOBS with more data sets and models.

We also invite the authors of other optimization algorithms to add their own method to the benchmark. Just edit a run script to include the new optimization method and create a pull-request.

Provided that this new optimizer produces competitive results, we will add the results to the set of provided baselines.

CHAPTER 2

Simple Example

This tutorial will show you an example of how DeepOBS can be used to benchmark the performance of a new optimization method for deep learning.

This simple example aims to show you some basic functions of DeepOBS, by creating a run script for a new optimizer (we will use the Momentum optimizer as an example here) and running it on a very simple test problem.

We show this example for **TensorFlow** and **PyTorch** respectively.

2.1 Create new Run Script

The easiest way to use DeepOBS with a new optimizer is to write a run script for it. This run script will import the optimizer and list its hyperparameters. For the Momentum optimizer in **TensorFlow** this is

```
"""Example run script using StandardRunner."""

import tensorflow as tf
from deepobs import tensorflow as tfobs

optimizer_class = tf.train.MomentumOptimizer
hyperparams = {"learning_rate": {"type": float},
               "momentum": {"type": float, "default": 0.99},
               "use_nesterov": {"type": bool, "default": False}}

runner = tfobs.runners.StandardRunner(optimizer_class, hyperparams)
runner.run(testproblem='quadratic_deep', hyperparams={'learning_rate': 1e-2}, num_
↪epochs=10)
```

You can download this `example run script tensorflow` and use it as a template.

For the Momentum optimizer in **PyTorch** it is

```
"""Example run script using StandardRunner."""
```

(continues on next page)

(continued from previous page)

```
from torch.optim import SGD
from deepobs import pytorch as pt

optimizer_class = SGD
hyperparams = {"lr": {"type": float},
               "momentum": {"type": float, "default": 0.99},
               "nesterov": {"type": bool, "default": False}}

runner = pt.runners.StandardRunner(optimizer_class, hyperparams)
runner.run(testproblem='quadratic_deep', hyperparams={'lr': 1e-2}, num_epochs=10)
```

You can download this `example run script pytorch` and use it as a template.

The DeepOBS runner needs access to an optimizer class with the same API as the TensorFlow/PyTorch optimizers and a list of additional hyperparameters for this new optimizers.

2.2 Run new Optimizer

You can now just execute the above mentioned script to run Momentum on the `quadratic_deep` test problem. You can change the arguments in the `run()` method to run other test problems, other hyperparameter settings, different number of epochs, etc.. If you want to make the script command line based, you can simply remove all arguments in the `run()` method an parse them from the command line. For **TensorFlow** this would look like this:

```
python runner_momentum_tensorflow.py quadratic_deep --bs 128 --learning_rate 1e-2 --
↪momentum 0.99 --num_epochs 10
```

We will run it a couple times more this time with different `learning_rates`

```
python runner_momentum_tensorflow.py quadratic_deep --bs 128 --learning_rate 1e-3 --
↪momentum 0.99 --num_epochs 10
python runner_momentum_tensorflow.py quadratic_deep --bs 128 --learning_rate 1e-4 --
↪momentum 0.99 --num_epochs 10
python runner_momentum_tensorflow.py quadratic_deep --bs 128 --learning_rate 1e-5 --
↪momentum 0.99 --num_epochs 10
```

For **PyTorch** this would look like this:

```
python runner_momentum_pytorch.py quadratic_deep --bs 128 --lr 1e-2 --momentum 0.99 --
↪num_epochs 10
```

We will run it a couple times more this time with different `lr`

```
python runner_momentum_pytorch.py quadratic_deep --bs 128 --lr 1e-3 --momentum 0.99 --
↪num_epochs 10
python runner_momentum_pytorch.py quadratic_deep --bs 128 --lr 1e-4 --momentum 0.99 --
↪num_epochs 10
python runner_momentum_pytorch.py quadratic_deep --bs 128 --lr 1e-5 --momentum 0.99 --
↪num_epochs 10
```

2.3 Analyzing the Runs

We can use DeepOBS's `analyzer` module to automatically find the best hyperparameter setting. First note, that the runner writes the output in a directory tree like:

<results_name>/<testproblem>/<optimizer>/<hyperparameter_setting>/

In the above example, the directory of the run outputs for **TensorFlow** would be:

./results/quadratic_deep/MomentumOptimizer/...

And for **PyTorch**:

./results/quadratic_deep/SGD/...

We pass the path to the optimizer directory to the analyzer functions. This way, we can get the best performance setting, a plot for the corresponding training curve and a plot that visualizes the hyperparameter sensitivity.

For **TensorFlow** and **PyTorch**:

```
from deepobs import analyzer

# get the overall best performance of the MomentumOptimizer on the quadratic_deep␣
↪testproblem
performance_dic = analyzer.get_performance_dictionary('./results/quadratic_deep/SGD')
print(performance_dic)

# plot the training curve for the best performance
analyzer.plot_optimizer_performance('./results/quadratic_deep/SGD')

# plot again, but this time compare to the Adam baseline
analyzer.plot_optimizer_performance('./results/quadratic_deep/SGD',
                                    reference_path='../DeepOBS_Baselines/baselines_
↪tensorflow/quadratic_deep/MomentumOptimizer')
```

You need to change the results directory accordingly, i.e. in our example it would be

```
'./results/quadratic_deep/SGD'
```

for TensorFlow (as in the example above) and

```
'./results/quadratic_deep/MomentumOptimizer'
```

for PyTorch.

You can download the script and use it as a template for further analysis: `example analyze script tensorflow`.

Note that you can also select a reference path (here the deepobs baselines for **TensorFlow**) to plot reference results as well. You can download the latest baselines from GitHub

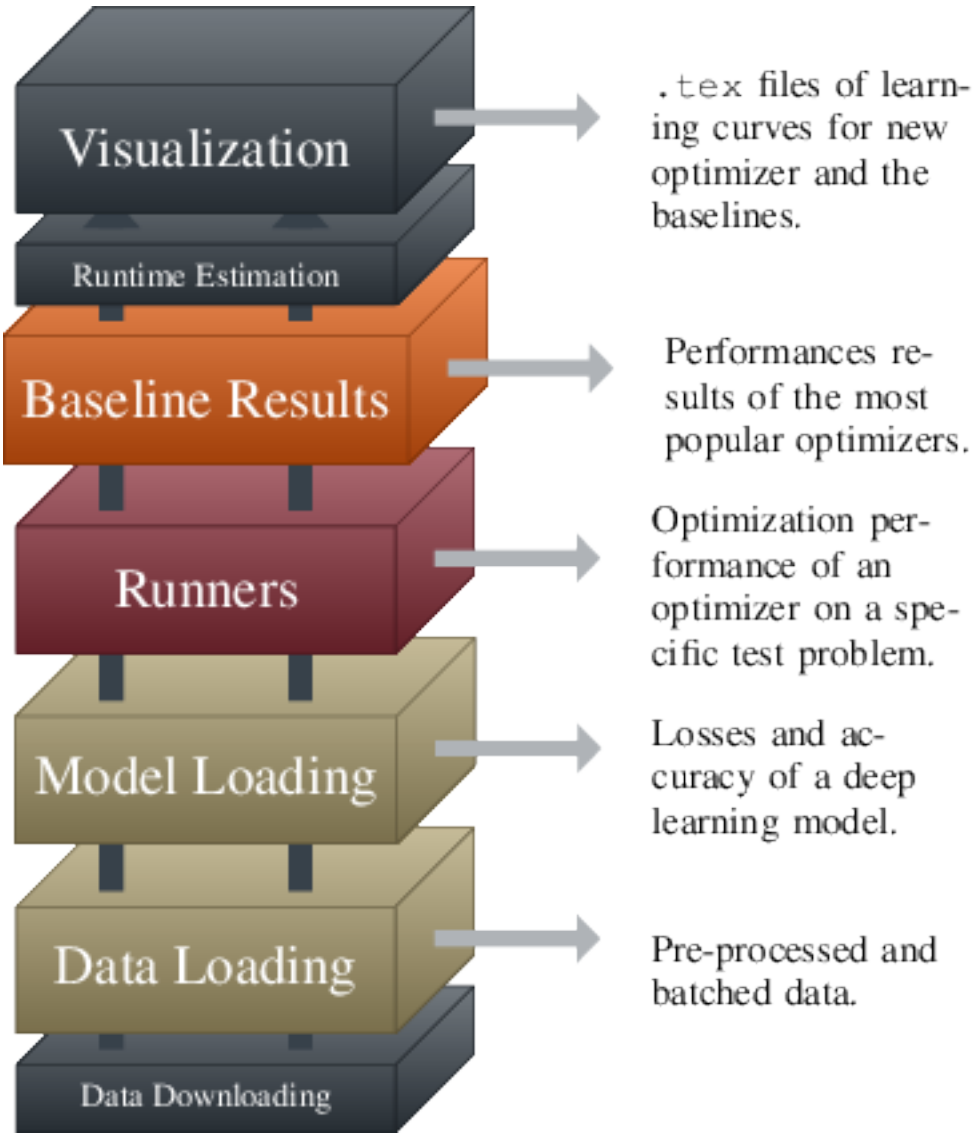
CHAPTER 3

Overview

DeepOBS provides modules and scripts for the full stack required to rapidly, reliably and reproducibly benchmark deep learning optimizers.

Here we briefly described the different levels of automation that DeepOBS provides. While, they are built hierarchically, they can be used separately. For example, one can use just the data loading capabilities of DeepOBS and built a new test problem on top of it.

A more detailed description of the modules and scripts can be found in the API reference section.



3.1 Data Downloading

DeepOBS can automatically download and pre-process all necessary data sets. This includes

- MNIST
- Fashion-MNIST (FMNIST)
- CIFAR-10
- CIFAR-100
- Street View House Numbers (SVHN)
- Leo Tolstoi's War and Peace

Note: While ImageNet is part of DeepOBS, it is currently not part of the automatic data downloading pipeline mechanic. Downloading the *ImageNet* data set requires an account and can take a lot of time to

download. Additonally, it requires quite a large amount of memory. The best way currently is to download and preprocess the *ImageNet* data set separately if needed and move it into the DeepOBS data folder.

The automatic data preparation for the **TensorFlow** version script can be run using

```
deepobs_prepare_data.sh
```

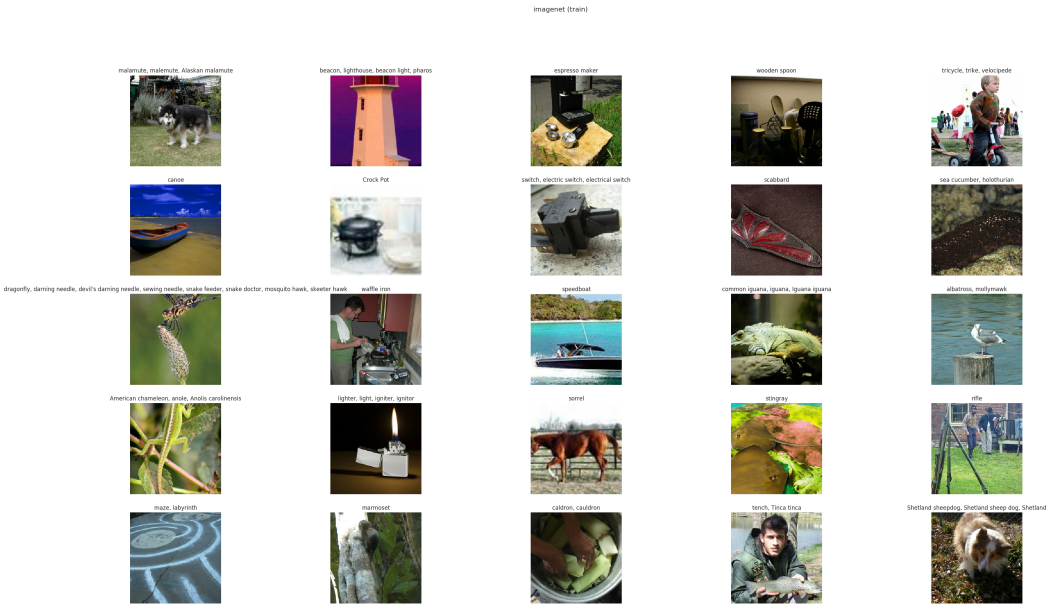
and is described in the API section under *Prepare Data*.

For the **PyTorch** version the data preparation is mostly done automatically by Torchvision. If you use a test problem where the data set is not available in Torchvision (e.g. Tolstoi's War and Peace) you can execute the above mentioned script for the PyTorch version as well.

3.2 Data Loading

The DeepOBS data loading module then performs all necessary processing of the data sets to return inputs and outputs for the deep learning model (e.g. images and labels for image classification). This processing includes splitting, shuffling, batching and data augmentation. The data loading module can also be used to build new deep learning models that are not (yet) part of DeepOBS.

The outputs of the data loading module is illustrated in the figure below and is further described in the API section for TensorFlow (*Data Sets*) and PyTorch (*Data Sets*) respectively.



3.3 Model Loading

Together, data set and model define a loss function and thus an optimization problem. We selected problems for diversity of task as well as the difficulty of the optimization problem itself. The list of test problems of DeepOBS includes popular image classification models on data sets like `MNIST`, `CIFAR-10` or `ImageNet`, but also models for natural language processing and generative models.

Additionally, three two-dimensional problems and an ill-conditioned quadratic problem are included. These simple tests can be used as illustrative toy problems to highlight properties of an algorithm and perform sanity-checks.

Over time, we plan to expand this list when hardware and research progress renders small problems out of date, and introduces new research directions and more challenging problems.

The implementation of the models is described in the API section under *Test Problems* and *Test Problems* respectively.

3.4 Runners

The runners of the DeepOBS package handle training and the logging of statistics measuring the optimizer's performance. For optimizers following the standard TensorFlow or PyTorch optimizer API it is enough to provide the runners with a list of the optimizer's hyperparameters. We provide a template for this, as well as an example of including a more sophisticated optimizer that can't be described as a subclass of the TensorFlow or PyTorch optimizer API.

In the API section, we describe the runners for *TensorFlow* and *PyTorch* and in the *Simple Example* we show an example of creating a run script for a new optimizer.

3.5 Baseline Results

DeepOBS also provides realistic baselines results for, currently, the three most popular optimizers in deep learning, `SGD`, `Momentum`, and `Adam`. These allow comparing a newly developed algorithm to the competition without computational overhead, and without risk of conscious or unconscious bias against the competition.

Baselines for further optimizers will be added when authors provide the optimizer's code, assuming the method perform competitively. Currently, baselines are available for all test problems in the small and large benchmark set.

The current baselines can be downloaded from github.

3.6 Runtime Estimation

In the current DeepOBS version, runtime estimation is not yet available.

3.7 Visualization

The DeepOBS analyzer module reduces the overhead for the preparation of results, and simultaneously standardizes the presentation, making it possible to include a comparably large amount of information in limited space. A more detailed description can be found in its API reference: *Analyzer*. We also procide an example: *Simple Example*

CHAPTER 4

Suggested Protocol

Here we provide a suggested protocol for more rigorously benchmarking deep learning optimizer. Some of the steps were discussed in the DeepOBS paper. Others were derived in the Master's thesis of Aaron Bahde.

4.1 Decide for a Framework

DeepOBS versions >= 1.2.0 support TensorFlow and PyTorch. We ran some basic experiments to check whether these two frameworks can be used interchangeably. So far, we strongly recommend to NOT compare benchmarks (with DeepOBS) across these frameworks. Currently, we only provide baselines for PyTorch.

You can choose between PyTorch and TensorFlow by switching the import statements:

```
# for example import the standard runner from the pytorch submodule
from deepobs.pytorch.runners import StandardRunner

# or from the tensorflow submodule
from deepobs.tensorflow.runners import StandardRunner
```

4.2 Create new Run Script

In order to benchmark a new optimization method a new run script has to be written. A more detailed description can be found in the *Simple Example* and the API section for TensorFlow (*Standard Runner*) and PyTorch (*Standard Runner*). Essentially, all which is needed is the optimizer itself and a list of its hyperparameters. For example for the Momentum optimizer in **Tensorlow** this will be:

```
"""Example run script using StandardRunner."""

import tensorflow as tf
from deepobs import tensorflow as tfobs

```

(continues on next page)

(continued from previous page)

```
optimizer_class = tf.train.MomentumOptimizer
hyperparams = {"learning_rate": {"type": float},
               "momentum": {"type": float, "default": 0.99},
               "use_nesterov": {"type": bool, "default": False}}

runner = tfobs.runners.StandardRunner(optimizer_class, hyperparams)
runner.run(testproblem='quadratic_deep', hyperparams={'learning_rate': 1e-2}, num_
↪epochs=10)
```

And in **PyTorch**:

```
"""Example run script using StandardRunner."""

from torch.optim import SGD
from deepobs import pytorch as pt

optimizer_class = SGD
hyperparams = {"lr": {"type": float},
               "momentum": {"type": float, "default": 0.99},
               "nesterov": {"type": bool, "default": False}}

runner = pt.runners.StandardRunner(optimizer_class, hyperparams)
runner.run(testproblem='quadratic_deep', hyperparams={'lr': 1e-2}, num_epochs=10)
```

4.3 (Possibly) Write Your Own Runner

You should at first try to execute your optimizer with one of the implemented runner classes. If this does not work out, because your optimizer needs additional access to the training loop, you have to write your own runner class. We provide a description how to do this: *How to Write Customized Runner*

4.4 Identify Tunable Hyperparameters

We suggest that you decide which hyperparameters of your optimizer needs to be tuned *before* starting the benchmark. For every test problem you should tune exactly the same hyperparameters with the same resources and the same tuning method. This avoids overfitting of hyperparameters on specific test problems.

4.5 Decide for a Tuning Method

We provide three tuning classes in DeepOBS. You should use one of them:

```
# Grid Search
from deepobs.tuner import GridSearch

# Random Search
from deepobs.tuner import RandomSearch

# Bayesian optimization with a Gaussian process surrogate
from deepobs.tuner import GP
```

Ideally, you use the same tuning method that we used for the baselines. At the moment this is grid search.

4.6 Specify the Tuning Domain

Prospective users of your optimizer expect you to provide information about how to tune your optimizer in *any* application. Therefore, you should provide promising search domains. They should be the same for *all* test problems since the users do not know the link between your optimizer's hyperparameters and the application. In DeepOBS you can user the tuning specifications of each tuner class. This is an example for the Momentum optimizer in PyTorch:

```
from deepobs.tuner import GridSearch, RandomSearch, GP
from torch.optim import SGD
import numpy as np
from deepobs.pytorch.runners import StandardRunner
from deepobs.tuner.tuner_utils import log_uniform
from scipy.stats.distributions import uniform, binom

# define optimizer
optimizer_class = SGD
hyperparams = {"lr": {"type": float},
               "momentum": {"type": float},
               "nesterov": {"type": bool}}

### Grid Search ###
# The discrete values to construct a grid for.
grid = {'lr': np.logspace(-5, 2, 6),
        'momentum': [0.5, 0.7, 0.9],
        'nesterov': [False, True]}

# Make sure to set the amount of resources to the grid size. For grid search, this is␣
↪just a sanity check.
tuner = GridSearch(optimizer_class, hyperparams, grid, runner=StandardRunner,␣
↪ressources=6*3*2)

### Random Search ###
# Define the distributions to sample from
distributions = {'lr': log_uniform(-5, 2),
        'momentum': uniform(0.5, 0.5),
        'nesterov': binom(1, 0.5)}

# Allow 36 random evaluations.
tuner = RandomSearch(optimizer_class, hyperparams, distributions,␣
↪runner=StandardRunner, ressources=36)

### Bayesian Optimization ###
# The bounds for the suggestions
bounds = {'lr': (-5, 2),
        'momentum': (0.5, 1),
        'nesterov': (0, 1)}


# Corresponds to rescaling the kernel in log space.
def lr_transform(lr):
    return 10**lr


# Nesterov is discrete but will be suggested continious.
def nesterov_transform(nesterov):
    return bool(round(nesterov))
```

(continues on next page)

(continued from previous page)

```
# The transformations of the search space. The momentum parameter does not need a␣
↪transformation.
transformations = {'lr': lr_transform,
                   'nesterov': nesterov_transform}

tuner = GP(optimizer_class, hyperparams, bounds, runner=StandardRunner, ressources=36,
↪ transformations=transformations)
```

4.7 Bound the Tuning Resources

The tuning of your optimizer's hyperparameters should *never* exceed the number of instances that were used for the baselines. Less is always better. For our current baselines we used 20 instances for each optimizer on each test problem. Use the `ressources` argument in the tuner class instantiation to limit them.

4.8 Report Stochasticity

To get an understanding of the robustness of the optimizer against training noise we recommend to rerun the best hyperparameter instance of your optimizer with 10 different random seeds. The tuning classes can automatically take care of it:

```
from deepobs.tuner import GridSearch
from torch.optim import SGD
import numpy as np
from deepobs.pytorch.runners import StandardRunner

# define optimizer
optimizer_class = SGD
hyperparams = {"lr": {"type": float}}

### Grid Search ###
# The discrete values to construct a grid for.
grid = {'lr': np.logspace(-5, 2, 6)}

# init tuner class
tuner = GridSearch(optimizer_class, hyperparams, grid, runner=StandardRunner,␣
↪ressources=6)

# tune on quadratic test problem and automatically rerun the best instance with 10␣
↪different seeds.
tuner.tune('quadratic_deep', rerun_best_setting=True)
```

4.9 Run on a Variety of Test Problems

Benchmark results might vary a lot for different test problems. We recommend to run your optimizer on as many test problems as possible but (of course) focus on the ones we use for the baselines. We provide a 'small' test set and a 'large' test set that, in our opinion, reflects a good variety of test problems. They are accessible as global variables in DeepOBS. One way to use them is to automatically tune your optimizer on the recommendations:

```
from deepobs.tuner import GridSearch
from torch.optim import SGD
import numpy as np
from deepobs.pytorch.runners import StandardRunner
from deepobs.config import get_small_test_set

# define optimizer
optimizer_class = SGD
hyperparams = {"lr": {"type": float}}

### Grid Search ###
# The discrete values to construct a grid for.
grid = {'lr': np.logspace(-5, 2, 6)}

# init tuner class
tuner = GridSearch(optimizer_class, hyperparams, grid, runner=StandardRunner,␣
↪ressources=6)

# get the small test set and automatically tune on each of the contained test problems
small_testset = get_small_test_set()
tuner.tune_on_testset(small_testset, rerun_best_setting=True)    # kwargs are parsed␣
↪to the tune() method
```

4.10 Plot Results

To visualize the final results, the user can use the *Analyzer* API. We recommend to include a plot about the hyperparameter sensitivity and to plot your optimizer performance against the baselines:

```
from deepobs.analyzer.analyze import plot_optimizer_performance, plot_hyperparameter_
↪sensitivity

# plot your optimizer against baselines
plot_optimizer_performance('/<path to your results folder>/<test problem>/<your␣
↪optimizer>',
                           reference_path='<path to the baselines>/<test problem>/SGD
↪')

# plot the hyperparameter sensitivity (here we use the learning rate sensitivity of␣
↪the SGD baseline)
plot_hyperparameter_sensitivity('<path to the baselines>/<test problem>/SGD',
                                hyperparam='lr',
                                xscale='log',
                                plot_std=True)
```

4.11 Report Measures for Speed

DeepOBS calculates the speed of your optimizer as a fraction of epochs that it needs to reach the convergence performance of the baselines. This measure is included automatically in the overview table generated by the Analyzer. Additionally, you can calculate an estimate for wall-clock time performance in comparison to SGD. More details can be found in the DeepOBS paper

```
from deepobs.analyzer.analyze import plot_results_table, estimate_runtime
from deepobs.pytorch.runners import StandardRunner
from torch.optim import Adam

# plot the overview table which contains the speed measure for iterations
plot_results_table('<path to your results>', conv_perf_file='<path to the convergence␣
↪performance file of the baselines>')

# briefly run your optimizer against SGD to estimate wall-clock time overhead, here␣
↪we use Adam as an example
estimate_runtime(framework='pytorch',
                 runner_cls=StandardRunner,
                 optimizer_cls=Adam,
                 optimizer_hp={"lr": {"type": float}},
                 optimizer_hyperparams={'lr': 0.1})
```

CHAPTER 5

How to Write Customized Runner

Some optimizers have special requirements. For example, they need access to the training loop and, therefore, cannot use DeepOBS as a black box function. Or, the hyperparameters of the optimizer are somewhat special (e.g. other optimizer instances). For these cases, we give the users the possibility to write their own Runner class. Here, we describe in more detail what you have to do for that.

5.1 Decide for a Framework

Since the latest DeepOBS version comes with TensorFlow and PyTorch implementations you first have to decide on the framework to use. If you decide for TensorFlow your Runner must inherite from the `TFRunner` class. It can be found in the API section *TF Runner*.

If you decide for PyTorch your Runner must inherite from the `PTRunner` class. It can be found in the API section *PT Runner*

5.2 Implement the Training Loop

The most import implementation for your customized runner is the method `training` which runs the training loop on the testproblem. Its basic signature can be found in the API section for *TF Runner* and *PT Runner* respectively. Concrete example implementations can be found in the Runner classes that come with DeepOBS. We recommend copying one of those and adapt it to your needs. In principle, simply make sure that the output dictionary is filled with the metrices `test_accuracies`, `valid_accuracies`, `train_accuracies`, `test_losses`, `valid_losses` and `tain_losses` during training. Additionally, we distinguish between `hyperparameters` (which are the parameters that are used to initialize the optimizer) and `training parameters` (which are used as additional keyword arguments in the training loop).

For the PyTorch version we would like to give some useful hints:

1. A `deepobs.pytorch.testproblems.testproblem` instance holds the attribute `net` which is the model that is to be trained. This way, you have full access to the model parameters during training.

2. Somewhat counterintuitively, we implemented a method `get_batch_loss_and_accuracy` for each testproblem. This method gets the next batch of the training set and evaluates the forward path. We implemented a closure such that you can call the forward path several times within the trainig loop (e.g. a second time after a parameter update). For this, simply set the argument `return_forward_func = True` of `get_batch_loss_and_accuracy`.

3. A `deepobs.pytorch.testproblems.testproblem` instance holds the attribute `regularization_groups`. It can be used to modify the way your optimizer deals with the regularization.

5.3 Read in Hyperparameters and Training Parameters from the Command Line

To use your Runner scripts from the command line, you have to specify the way the hyper and training parameters should be read in by argparse. For that, you can overwrite the methods `_add_training_params_to_argparse` and `_add_hyperparams_to_argparse`. For both frameworks, examples can be found in the `LearningRateScheduleRunner`.

5.4 Specify How the Hyperparameters and Training Parameters Should Be Added to the Run Name

Each individual run ends with writing the output to a well structured directory tree. This is important for later analysis of the results. To specify how your hyper and training parameters should be used for the naming of the setting directories, you have to overwrite the methods `_add_training_params_to_output_dir_name` and `_add_hyperparams_to_output_dir_name`. For both frameworks, examples can be found in the `LearningRateScheduleRunner`.

CHAPTER 6

Tuning Automation

To address the unfairness that arises from the tuning procedure, we implemented a tuning automation in DeepOBS. Here, we describe how to use it. We also provide some basic functionalities to monitor the tuning process. These are not explained here, but can be found in the API section of the *Tuner*. We further describe a comperative and fair usage of the tuning automation in the *Suggested Protocol*.

We provide three different Tuner classes: `GridSearch`, `RandomSearch` and `GP` (which is a Bayesian optimization method with a Gaussian Process surrogate). You can find detailed information about them in the API section *Tuner*. We will show all examples in this section for the PyTorch framework.

6.1 Grid Search

To perform an automated grid search you first have to create the Tuner instance. The optimizer class and its hyperparameters have to be specified in the same way like for Runners. Additionally, you have to give a dictionary that holds the discrete values of each hyperparameter. By default, calling `tune` will execute the whole tuning process in a sequential way on the given hardware.

If you want to parallelize the tuning process you can use the method `generate_commands_script`. It generates commands than can be send to different nodes. If the format of the command string is not correct for your training or hyper parameters you have to overwrite the methods `_generate_kwargs_format_for_command_line` and `_generate_hyperparams_format_for_command_line` of the `ParallelizedTuner` accordingly. Note that the generated commands refer to a run script that you have to specify on your own. Here, as an example, the generated commands refer to `a standard SGD script`

```
from deepobs.tuner import GridSearch
from torch.optim import SGD
import numpy as np
from deepobs.pytorch.runners import StandardRunner

optimizer_class = SGD
hyperparams = {
    "lr": {"type": float},
    "momentum": {"type": float},
```

(continues on next page)

(continued from previous page)

```
    "nesterov": {"type": bool},
}

# The discrete values to construct a grid for.
grid = {
    "lr": np.logspace(-5, 2, 6),
    "momentum": [0.5, 0.7, 0.9],
    "nesterov": [False, True],
}

# Make sure to set the amount of ressources to the grid size. For grid search, this␣
↪is just a sanity check.
tuner = GridSearch(
    optimizer_class,
    hyperparams,
    grid,
    runner=StandardRunner,
    ressources=6 * 3 * 2,
)

# Tune (i.e. evaluate every grid point) and rerun the best setting with 10 different␣
↪seeds.
# tuner.tune('quadratic_deep', rerun_best_setting=True, num_epochs=2, output_dir='./
↪grid_search')

# Optionally, generate commands for a parallelized execution
tuner.generate_commands_script(
    "quadratic_deep",
    run_script="../runner_momentum_pytorch.py",
    num_epochs=2,
    output_dir="./grid_search",
    generation_dir="./grid_search_commands",
)
```

You can download this `example` and use it as a template.

6.2 Random Search

For the random search, you have to give a dictionary that holds the distributions for each hyperparameter:

```
from deepobs.tuner import RandomSearch
from torch.optim import SGD
from deepobs.tuner.tuner_utils import log_uniform
from scipy.stats.distributions import uniform, binom
from deepobs import config
from deepobs.pytorch.runners import StandardRunner

optimizer_class = SGD
hyperparams = {
    "lr": {"type": float},
    "momentum": {"type": float},
    "nesterov": {"type": bool},
}
```

(continues on next page)

(continued from previous page)

```
# Define the distributions to sample from
distributions = {
    "lr": log_uniform(-5, 2),
    "momentum": uniform(0.5, 0.5),
    "nesterov": binom(1, 0.5),
}

# Allow 36 random evaluations.
tuner = RandomSearch(
    optimizer_class,
    hyperparams,
    distributions,
    runner=StandardRunner,
    ressources=36,
)

# Tune (i.e. evaluate 36 different random samples) and rerun the best setting with 10␣
↪different seeds.
tuner.tune(
    "quadratic_deep",
    rerun_best_setting=True,
    num_epochs=2,
    output_dir="./random_search",
)

# Optionally, generate commands for a parallelized execution
tuner.generate_commands_script(
    "quadratic_deep",
    run_script="../runner_momentum_pytorch.py",
    num_epochs=2,
    output_dir="./random_search",
    generation_dir="./random_search_commands",
)
```

You can download this `example` and use it as a template.

6.3 Bayesian Optimization (GP)

The Bayesian optimization method with a Gaussian process surrogate is more complex. At first, you have to specify the bounds of the suggestions. Additionally, you can set the transformation of the search space. In combination with the bounds, this can be used for a rescaling of the kernel or for optimization of discrete values:

```
from deepobs.tuner import GP
from torch.optim import SGD
from sklearn.gaussian_process.kernels import Matern
from deepobs import config
from deepobs.pytorch.runners import StandardRunner

optimizer_class = SGD
hyperparams = {"lr": {"type": float},
               "momentum": {"type": float},
               "nesterov": {"type": bool}}
```

(continues on next page)

(continued from previous page)

```
# The bounds for the suggestions
bounds = {'lr': (-5, 2),
          'momentum': (0.5, 1),
          'nesterov': (0, 1)}


# Corresponds to rescaling the kernel in log space.
def lr_transform(lr):
    return 10**lr


# Nesterov is discrete but will be suggested continious.
def nesterov_transform(nesterov):
    return bool(round(nesterov))


# The transformations of the search space. Momentum does not need a transformation.
transformations = {'lr': lr_transform,
                   'nesterov': nesterov_transform}

tuner = GP(optimizer_class, hyperparams, bounds, runner=StandardRunner, ressources=36,
↪ transformations=transformations)

# Tune with a Matern kernel and rerun the best setting with 10 different seeds.
tuner.tune('quadratic_deep', kernel=Matern(nu=2.5), rerun_best_setting=True, num_
↪epochs=2, output_dir='./gp_tuner')
```

You can download this `example` and use it as a template. Since Bayesian optimization is sequential by nature, we do not offer a parallelized version of it.

CHAPTER 7

Analyzer

DeepOBS uses the analyzer module to get meaning full outputs from the results created by the runners. This includes:

- Getting the best settings (e.g. best `learning rate`) for an optimizer on a specific test problem.
- Plotting the hyperparameter (e.g. `learning_rate`) sensitivity for multiple optimizers on a test problem.
- Plotting all performance metrics of the whole benchmark set.
- Returning the overall performance table for multiple optimizers.

The analyzer can return those outputs as matplotlib plots for customization.

7.1 Validate Output

`deepobs.analyzer.`**`check_output`**(*results_path*)

Iterates through the results folder an checks all outputs for format and completeness. It checks for some basic format in every json file and looks for setting folders which are empty. It further gives an overview over the amount of different settings and seed runs for each test problem and each optimizer. It does not return anything, but it prints an overview to the console.

Parameters **`results_path`** (*`str`*) – Path to the results folder.

7.2 Plot Optimizer Performances

`deepobs.analyzer.`**`plot_optimizer_performance`**(*path*, *fig=None*, *ax=None*, *mode='most'*, *metric='valid_accuracies'*, *reference_path=None*, *show=True*, *which='mean_and_std'*)

Plots the training curve of optimizers and addionally plots reference results from the `reference_path`

Parameters

- **path** (*str*) – Path to the optimizer or to a whole testproblem (in this case all optimizers in the testproblem folder are plotted).
- **fig** (*matplotlib.Figure*) – Figure to plot the training curves in.
- **ax** (*matplotlib.axes.Axes*) – The axes to plot the trainig curves for all metrices. Must have 4 subaxes (one for each metric).
- **mode** (*str*) – The mode by which to decide the best setting.
- **metric** (*str*) – The metric by which to decide the best setting.
- **reference_path** (*str*) – Path to the reference optimizer or to a whole testproblem (in this case all optimizers in the testproblem folder are taken as reference).
- **show** (*bool*) – Whether to show the plot or not.
- **which** (*str*) – ['mean_and_std', 'median_and_quartiles'] Solid plot mean or median, shaded plots standard deviation or lower/upper quartiles.

Returns The figure and axes with the plots.

Return type tuple

deepobs.analyzer.**plot_testset_performances**(*results_path*, *mode='most'*, *metric='valid_accuracies'*, *reference_path=None*, *show=True*, *which='mean_and_std'*)

Plots all optimizer performances for all testproblems.

Parameters

- **results_path** (*str*) – The path to the results folder.
- **mode** (*str*) – The mode by which to decide the best setting.
- **metric** (*str*) – The metric by which to decide the best setting.
- **reference_path** (*str*) – Path to the reference results folder. For each available reference testproblem, all optimizers are plotted as reference.
- **show** (*bool*) – Whether to show the plot or not.
- **which** (*str*) – ['mean_and_std', 'median_and_quartiles'] Solid plot mean or median, shaded plots standard deviation or lower/upper quartiles.

Returns The figure and axes.

Return type tuple

7.3 Get the Best Runs

deepobs.analyzer.**plot_results_table**(*results_path*, *mode='most'*, *metric='valid_accuracies'*, *conv_perf_file=None*)

Summarizes the performance of the optimizer and prints it to a pandas data frame.

Parameters

- **results_path** (*str*) – The path to the results directory.
- **mode** (*str*) – The mode by which to decide the best setting.
- **metric** (*str*) – The metric by which to decide the best setting.

- **conv_perf_file** (*str*) – Path to the convergence performance file. It is used to calculate the speed of the optimizer. Defaults to `None` in which case the speed measure is N.A.

Returns A data frame that summarizes the results on the test set.

Return type pandas.DataFrame

`deepobs.analyzer.`**`get_performance_dictionary`**(*optimizer_path*, *mode='most'*, *metric='valid_accuracies'*, *conv_perf_file=None*)

Summarizes the performance of the optimizer.

Parameters

- **optimizer_path** (*str*) – The path to the optimizer to analyse.
- **mode** (*str*) – The mode by which to decide the best setting.
- **metric** (*str*) – The metric by which to decide the best setting.
- **conv_perf_file** (*str*) – Path to the convergence performance file. It is used to calculate the speed of the optimizer. Defaults to `None` in which case the speed measure is N.A.

Returns A dictionary that holds the best setting and it's performance on the test set.

Return type dict

7.4 Plot Hyperparameter Sensitivity

`deepobs.analyzer.`**`plot_hyperparameter_sensitivity`**(*path*, *hyperparam*, *mode='final'*, *metric='valid_accuracies'*, *xscale='linear'*, *plot_std=True*, *reference_path=None*, *show=True*)

Plots the hyperparameter sensitivtiy of the optimizer.

Parameters

- **path** (*str*) – The path to the optimizer to analyse. Or to a whole testproblem. In that case, all optimizer sensitivities are plotted.
- **hyperparam** (*str*) – The name of the hyperparameter that should be analyzed.
- **mode** (*str*) – The mode by which to decide the best setting.
- **metric** (*str*) – The metric by which to decide the best setting.
- **xscale** (*str*) – The scale for the parameter axes. Is passed to plt.xscale().
- **plot_std** (*bool*) – Whether to plot markers for individual seed runs or not. If *False*, only the mean is plotted.
- **reference_path** (*str*) – Path to the reference optimizer or to a whole testproblem (in this case all optimizers in the testproblem folder are taken as reference).
- **show** (*bool*) – Whether to show the plot or not.

Returns The figure and axes of the plot.

Return type tuple

7.5 Estimate Runtime

deepobs.analyzer.**estimate_runtime**(*framework*, *runner_cls*, *optimizer_cls*, *optimizer_hp*, *optimizer_hyperparams*, *n_runs=5*, *sgd_lr=0.01*, *testproblem='mnist_mlp'*, *num_epochs=5*, *batch_size=128*, ***kwargs*)

Can be used to estimates the runtime overhead of a new optimizer compared to SGD. Runs the new optimizer and SGD seperately and calculates the fraction of wall clock overhead.

Parameters

- **framework** (*str*) – Framework that you use. Must be 'pytorch' or 'tensorlfow'.
- **runner_cls** – The runner class that your optimizer uses.
- **optimizer_cls** – Your optimizer class.
- **optimizer_hp** (*dict*) – Its hyperparameter specification as it is used in the runner initialization.
- **optimizer_hyperparams** (*dict*) – Optimizer hyperparameter values to run.
- **n_runs** (*int*) – The number of run calls for which the overhead is averaged over.
- **sgd_lr** (*float*) – The vanilla SGD learning rate to use.
- **testproblem** (*str*) – The deepobs testproblem to run SGD and the new optimizer on.
- **num_epochs** (*int*) – The number of epochs to run for the testproblem.
- **batch_size** (*int*) – Batch size of the testproblem.

Returns The output that is printed to the console.

Return type str

CHAPTER 8

TensorFlow

8.1 Data Sets

Currently DeepOBS includes nine different data sets. Each data set inherits from the same base class with the following signature.

class deepobs.tensorflow.datasets.dataset.**DataSet**(*batch_size*)

Base class for DeepOBS data sets.

Parameters **batch_size** (*int*) – The mini-batch size to use.

batch

A tuple of tensors, yielding batches of data from the dataset. Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op

A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op

A tensorflow operation initializing the testproblem for evaluating on training data.

valid_init_op

A tensorflow operation initializing the dataset for the validation phase.

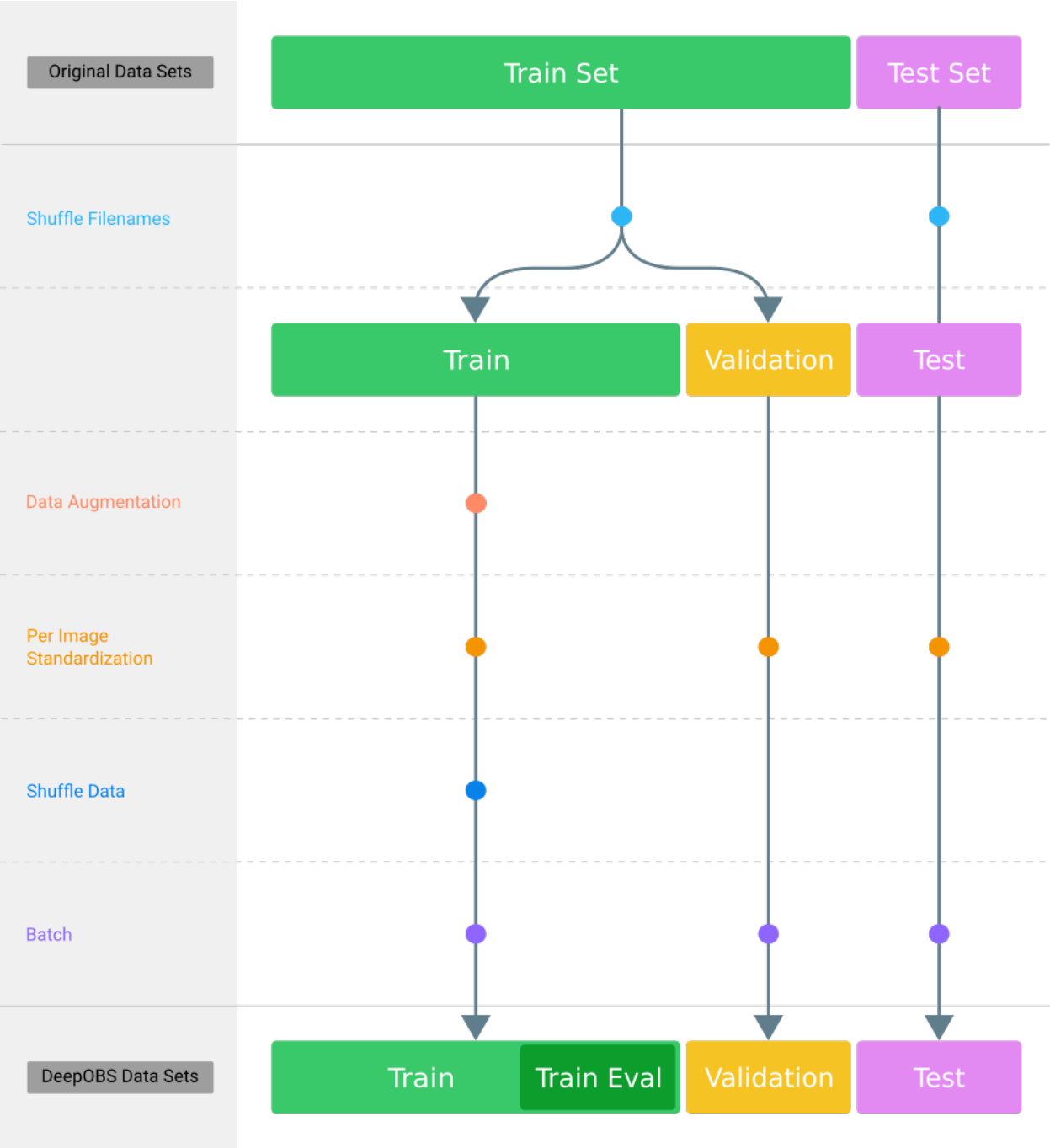
test_init_op

A tensorflow operation initializing the testproblem for evaluating on test data.

phase

A string-value tf.Variable that is set to `train`, `train_eval`, `valid`, or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

After selecting a data set (i.e. CIFAR-10, Fahion-MNIST, etc.) we define four internal TensorFlow data sets (i.e. *train*, *train_eval*, *valid* and *test*). Those are splits of the original data set that are used for training, hyperparameter tuning and performance evaluation. These internal data sets (also called DeepOBS data sets) are created as shown in the illustration below.


Original Data Sets
Train Set
Test Set
Shuffle Filenames
Train
Validation
Test
Data Augmentation
Per Image Standardization
Shuffle Data
Batch
DeepOBS Data Sets
Train
Train Eval
Validation
Test

8.1.1 2D Data Set

class deepobs.tensorflow.datasets.two_d.**two_d**(*batch_size*, *train_size=10000*, *noise_level=1.0*)

DeepOBS data set class to create two dimensional stochastic testproblems.

This toy data set consists of a fixed number (`train_size`) of iid draws from two scalar zero-mean normal distributions with standard deviation specified by the `noise_level`.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`1000` for train and test) the remainder is dropped in each epoch (after shuffling).
- **train_size** (*int*) – Size of the training data set. This will also be used as the train_eval and test set size. Defaults to `10000`.
- **noise_level** (*float*) – Standard deviation of the data points around the mean. The data points are drawn from a Gaussian distribution. Defaults to `1.0`.

batch
A tuple `(x, y)` of tensors with random x and y that can be used to create a noisy two dimensional testproblem. Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to "train", "train_eval" or "test", depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.1.2 Quadratic Data Set

class deepobs.tensorflow.datasets.quadratic.**quadratic**(*batch_size*, *dim=100*, *train_size=1000*, *noise_level=0.6*)

DeepOBS data set class to create an n dimensional stochastic quadratic testproblem.

This toy data set consists of a fixed number (`train_size`) of iid draws from a zero-mean normal distribution in `dim` dimensions with isotropic covariance specified by `noise_level`.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`1000` for train and test) the remainder is dropped in each epoch (after shuffling).
- **dim** (*int*) – Dimensionality of the quadratic. Defaults to `100`.
- **train_size** (*int*) – Size of the dataset; will be used for train, train eval and test datasets. Defaults to `1000`.

- **noise_level** (*float*) – Standard deviation of the data points around the mean. The data points are drawn from a Gaussian distribution. Defaults to `0.6`.

batch
A tensor `X` of shape `(batch_size, dim)` yielding elements from the dataset. Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to `train`, `train_eval` or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.1.3 MNIST Data Set

class deepobs.tensorflow.datasets.mnist.**mnist**(*batch_size*, *train_eval_size=10000*)
DeepOBS data set class for the MNIST data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`60 000` for train, `10 000` for test) the remainder is dropped in each epoch (after shuffling).
- **train_eval_size** (*int*) – Size of the train eval data set. Defaults to `10 000` the size of the test set.

batch
A tuple `(x, y)` of tensors, yielding batches of MNIST images (`x` with shape `(batch_size, 28, 28, 1)`) and corresponding one-hot label vectors (`y` with shape `(batch_size, 10)`). Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

valid_init_op
A tensorflow operation initializing the testproblem for evaluating on validation data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to `train`, `train_eval`, `valid`, or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.1.4 FMNIST Data Set

class deepobs.tensorflow.datasets.fmnist.**fmnist**(*batch_size*, *train_eval_size=10000*)
DeepOBS data set class for the Fashion-MNIST (FMNIST) data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`60 000` for train, `10 000` for test) the remainder is dropped in each epoch (after shuffling).
- **train_eval_size** (*int*) – Size of the train eval data set. Defaults to `10 000` the size of the test set.

batch
A tuple `(x, y)` of tensors, yielding batches of MNIST images (`x` with shape `(batch_size, 28, 28, 1)`) and corresponding one-hot label vectors (`y` with shape `(batch_size, 10)`). Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

valid_init_op
A tensorflow operation initializing the testproblem for evaluating on validation data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to `train`, `train_eval`, `valid`, or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.1.5 CIFAR-10 Data Set

class deepobs.tensorflow.datasets.cifar10.**cifar10**(*batch_size*, *data_augmentation=True*, *train_eval_size=10000*)

DeepOBS data set class for the CIFAR-10 data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`50 000` for train, `10 000` for test) the remainder is dropped in each epoch (after shuffling).
- **data_augmentation** (*bool*) – If `True` some data augmentation operations (random crop window, horizontal flipping, lighting augmentation) are applied to the training data (but not the test data).
- **train_eval_size** (*int*) – Size of the train eval data set. Defaults to `10 000` the size of the test set.

batch
A tuple `(x, y)` of tensors, yielding batches of CIFAR-10 images (`x` with shape `(batch_size, 32, 32, 3)`) and corresponding one-hot label vectors (`y` with shape `(batch_size, 10)`). Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

valid_init_op
A tensorflow operation initializing the testproblem for evaluating on validation data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to `train`, `train_eval`, `valid`, or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.1.6 CIFAR-100 Data Set

class deepobs.tensorflow.datasets.cifar100.**cifar100**(*batch_size*, *data_augmentation=True*, *train_eval_size=10000*)

DeepOBS data set class for the CIFAR-100 data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`50 000` for train, `10 000` for test) the remainder is dropped in each epoch (after shuffling).
- **data_augmentation** (*bool*) – If `True` some data augmentation operations (random crop window, horizontal flipping, lighting augmentation) are applied to the training data (but not the test data).
- **train_eval_size** (*int*) – Size of the train eval data set. Defaults to `10 000` the size of the test set.

batch
A tuple `(x, y)` of tensors, yielding batches of CIFAR-100 images (`x` with shape `(batch_size, 32, 32, 3)`) and corresponding one-hot label vectors (`y` with shape `(batch_size, 100)`). Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

valid_init_op
A tensorflow operation initializing the testproblem for evaluating on validation data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to `train`, `train_eval`, `valid`, or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.1.7 SVHN Data Set

class deepobs.tensorflow.datasets.svhn.**svhn**(*batch_size*, *data_augmentation=True*, *train_eval_size=26032*)

DeepOBS data set class for the Street View House Numbers (SVHN) data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`73 000` for train, `26 000` for test) the remainder is dropped in each epoch (after shuffling).
- **data_augmentation** (*bool*) – If `True` some data augmentation operations (random crop window, lighting augmentation) are applied to the training data (but not the test data).
- **train_eval_size** (*int*) – Size of the train eval dataset. Defaults to `26 000` the size of the test set.

batch
A tuple `(x, y)` of tensors, yielding batches of SVHN images (`x` with shape `(batch_size, 32, 32, 3)`) and corresponding one-hot label vectors (`y` with shape `(batch_size, 10)`). Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

valid_init_op
A tensorflow operation initializing the testproblem for evaluating on validation data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to `train`, `train_eval`, `valid`, or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.1.8 ImageNet Data Set

class deepobs.tensorflow.datasets.imagenet.**imagenet**(*batch_size*, *data_augmentation=True*, *train_eval_size=50000*)

DeepOBS data set class for the ImageNet data set.

Note: We use `1001` classes which includes an additional *background* class, as it is used for example by the inception net.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size the remainder is dropped in each epoch (after shuffling).
- **data_augmentation** (*bool*) – If `True` some data augmentation operations (random crop window, horizontal flipping, lighting augmentation) are applied to the training data (but not the test data).
- **train_eval_size** (*int*) – Size of the train eval dataset. Defaults to `10 000`.

batch
A tuple `(x, y)` of tensors, yielding batches of ImageNet images (`x` with shape `(batch_size, 224, 224, 3)`) and corresponding one-hot label vectors (`y` with shape `(batch_size, 1001)`). Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

valid_init_op
A tensorflow operation initializing the testproblem for evaluating on validation data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to `train`, `train_eval`, `valid`, or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.1.9 Tolstoi Data Set

class deepobs.tensorflow.datasets.tolstoi.**tolstoi**(*batch_size*, *seq_length=50*, *train_eval_size=653237*)

DeepOBS data set class for character prediction on *War and Peace* by Leo Tolstoi.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size the remainder is dropped in each epoch (after shuffling).
- **seq_length** (*int*) – Sequence length to be modeled in each step. Defaults to `50`.
- **train_eval_size** (*int*) – Size of the train eval dataset. Defaults to `653 237`, the size of the test set.

batch
A tuple `(x, y)` of tensors, yielding batches of tolstoi data (`x` with shape `(batch_size, seq_length)`) and (`y` with shape `(batch_size, seq_length)` which is `x` shifted by one). Executing these tensors raises a `tf.errors.OutOfRangeError` after one epoch.

train_init_op
A tensorflow operation initializing the dataset for the training phase.

train_eval_init_op
A tensorflow operation initializing the testproblem for evaluating on training data.

test_init_op
A tensorflow operation initializing the testproblem for evaluating on test data.

phase
A string-value tf.Variable that is set to `train`, `train_eval` or `test`, depending on the current phase. This can be used by testproblems to adapt their behavior to this phase.

8.2 Test Problems

Currently DeepOBS includes twenty-six different test problems. A test problem is given by a combination of a data set and a model and is characterized by its loss function.

Each test problem inherits from the same base class with the following signature.

class deepobs.tensorflow.testproblems.testproblem.**TestProblem**(*batch_size*, *weight_decay=None*)

Base class for DeepOBS test problems.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay (L2-regularization) factor to use. If not specified, the test problems revert to their respective defaults. Note: Some test problems do not use regularization and this value will be ignored in such a case.

dataset
The dataset used by the test problem (datasets.DataSet instance).

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term (might be a constant 0.0 for test problems that do not use regularization).

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Sets up the test problem.

This includes setting up the data loading pipeline for the data set and creating the tensorflow computation graph for this test problem (e.g. creating the neural network).

Note: Some of the test problems described here are based on more general implementations. For example the Wide ResNet 40-4 network on Cifar-100 is based on the general Wide ResNet architecture which is also implemented. Therefore, it is very easy to include new Wide ResNets if necessary.

8.2.1 2D Test Problems

Three two-dimensional test problems are included in DeepOBS. They are mainly included for illustrative purposes as these explicit loss functions can be plotted.

They are all stochastic variants of classical deterministic optimization test functions.

2D Beale

class deepobs.tensorflow.testproblems.two_d_beale.**two_d_beale**(*batch_size*, *weight_decay=None*)

DeepOBS test problem class for a stochastic version of thetwo-dimensional Beale function as the loss function.

Using the deterministic Beale function and adding stochastic noise of the form

$u \cdot x + v \cdot y$

where `x` and `y` are normally distributed with mean `0.0` and standard deviation `1.0` we get a loss function of the form

$((1.5 - u + u \cdot v)^2 + (2.25 - u + u \cdot v^2)^2 + (2.625 - \ u + u \cdot v^3)^2) + u \cdot x + v \cdot y.$

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for the two_d stochastic test problem.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

set_up()
Sets up the stochastic two-dimensional Beale test problem. Using `-4.5` and `4.5` as a starting point for the weights `u` and `v`.

2D Branin

class deepobs.tensorflow.testproblems.two_d_branin.**two_d_branin**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a stochastic version of thetwo-dimensional Branin function as the loss function.

Using the deterministic Branin function and adding stochastic noise of the form

$u \cdot x + v \cdot y$

where `x` and `y` are normally distributed with mean `0.0` and standard deviation `1.0` we get a loss function of the form

$(v - 5.1/(4 \cdot \pi^2)u^2 + 5/\pi u - 6)^2 + \ 10 \cdot (1 - 1/(8 \cdot \pi)) \cdot \cos(u) + 10 + u \cdot x + v \cdot y.$

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for the two_d stochastic test problem.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

set_up()
Sets up the stochastic two-dimensional Branin test problem. Using `2.5` and `12.5` as a starting point for the weights `u` and `v`.

2D Rosenbrock

class deepobs.tensorflow.testproblems.two_d_rosenbrock.**two_d_rosenbrock**(*batch_size*, *weight_decay=None*)

DeepOBS test problem class for a stochastic version of thetwo-dimensional Rosenbrock function as the loss function.

Using the deterministic Rosenbrock function and adding stochastic noise of the form

$u \cdot x + v \cdot y$

where `x` and `y` are normally distributed with mean `0.0` and standard deviation `1.0` we get a loss function of the form

$(1 - u)^2 + 100 \cdot (v - u^2)^2 + u \cdot x + v \cdot y$

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for the two_d stochastic test problem.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

set_up()
Sets up the stochastic two-dimensional Rosenbrock test problem. Using `-0.5` and `1.5` as a starting point for the weights `u` and `v`.

8.2.2 Quadratic Test Problems

DeepOBS includes a stochastic quadratic problem with an eigenspectrum similar to what has been reported for neural networks.

Other stochastic quadratic problems (of different dimensionality or with a different Hessian structure) can be created easily using the `quadratic_base` class.

class deepobs.tensorflow.testproblems._quadratic.**_quadratic_base**(*batch_size, weight_decay=None, hessian=array([[1., 0., 0., ..., 0., 0., 0.], [0., 1., 0., ..., 0., 0., 0.], [0., 0., 1., ..., 0., 0., 0.], ..., [0., 0., 0., ..., 1., 0., 0.], [0., 0., 0., ..., 0., 1., 0.], [0., 0., 0., ..., 0., 0., 1.]])*)

DeepOBS base class for a stochastic quadratic test problems creating lossfunctions of the form

$0.5 * (\theta - x)^T * Q * (\theta - x)$

with Hessian `Q` and "data" `x` coming from the quadratic data set, i.e., zero-mean normal.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.
- **hessian** (*np.array*) – Hessian of the quadratic problem. Defaults to the `100` dimensional identity.

dataset
The DeepOBS data set class for the quadratic test problem.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

set_up()
Sets up the stochastic quadratic test problem. The parameter `Theta` will be initialized to (a vector of) `1.0`.

Quadratic Deep

class deepobs.tensorflow.testproblems.quadratic_deep.**quadratic_deep**(*batch_size*, *weight_decay=None*)

DeepOBS test problem class for a stochastic quadratic test problem `100`dimensions. The 90 % of the eigenvalues of the Hessian are drawn from theinterval $(0.0, 1.0)$ and the other 10 % are from $(30.0, 60.0)$ simulating an eigenspectrum which has been reported for Deep Learning https://arxiv.org/abs/1611.01838.

This creatis a loss functions of the form

$0.5 * (\theta - x)^T * Q * (\theta - x)$

with Hessian `Q` and "data" `x` coming from the quadratic data set, i.e., zero-mean normal.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for the quadratic test problem.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

8.2.3 MNIST Test Problems

MNIST LogReg

class deepobs.tensorflow.testproblems.mnist_logreg.**mnist_logreg**(*batch_size*, *weight_decay=None*)

DeepOBS test problem class for multinomial logistic regression on MNIST.

No regularization is used and the weights and biases are initialized to `0.0`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for MNIST.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Sets up the logistic regression test problem on MNIST.

MNIST MLP

class deepobs.tensorflow.testproblems.mnist_mlp.**mnist_mlp**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a multi-layer perceptron neural network on MNIST.

The network is build as follows:

- Four fully-connected layers with `1000`, `500`, `100` and `10` units per layer.
- The first three layers use ReLU activation, and the last one a softmax activation.
- The biases are initialized to `0.0` and the weight matrices with truncated normal (standard deviation of `3e-2`)
- The model uses a cross entropy loss.
- No regularization is used.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for MNIST.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the multi-layer perceptron test problem instance on MNIST.

MNIST 2c2d

class deepobs.tensorflow.testproblems.mnist_2c2d.**mnist_2c2d**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a two convolutional and two dense layered neural network on MNIST.

The network has been adapted from the TensorFlow tutorial and consists of

- two conv layers with ReLUs, each followed by max-pooling
- one fully-connected layers with ReLUs
- 10-unit output layer with softmax
- cross-entropy loss
- No regularization

The weight matrices are initialized with truncated normal (standard deviation of `0.05`) and the biases are initialized to `0.05`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for MNIST.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Sets up the vanilla CNN test problem on MNIST.

MNIST VAE

class deepobs.tensorflow.testproblems.mnist_vae.**mnist_vae**(*batch_size*, *weight_decay=None*)

DeepOBS test problem class for a variational autoencoder (VAE) on MNIST.

The network has been adapted from the here and consists of an encoder:

- With three convolutional layers with each `64` filters.
- Using a leaky ReLU activation function with $\alpha = 0.3$
- Dropout layers after each convolutional layer with a rate of `0.2`.

and an decoder:

- With two dense layers with `24` and `49` units and leaky ReLU activation.
- With three deconvolutional layers with each `64` filters.
- Dropout layers after the first two deconvolutional layer with a rate of `0.2`.
- A final dense layer with `28 x 28` units and sigmoid activation.

No regularization is used.

Parameters

- **batch_size** (*type*) – Batch size to use.
- **weight_decay** (*type*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for MNIST.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

set_up()
Sets up the VAE test problem on MNIST.

8.2.4 Fashion-MNIST Test Problems

Fashion-MNIST LogReg

class deepobs.tensorflow.testproblems.fmnist_logreg.**fmnist_logreg**(*batch_size*, *weight_decay=None*)

DeepOBS test problem class for multinomial logistic regression on Fasion-MNIST.

No regularization is used and the weights and biases are initialized to `0.0`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for Fashion-MNIST.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the logistic regression test problem on Fashion-MNIST.

Fashion-MNIST MLP

class deepobs.tensorflow.testproblems.fmnist_mlp.**fmnist_mlp**(*batch_size*, *weight_decay=None*)

DeepOBS test problem class for a multi-layer perceptron neural network on Fashion-MNIST.

The network is build as follows:

- Four fully-connected layers with `1000`, `500`, `100` and `10` units per layer.
- The first three layers use ReLU activation, and the last one a softmax activation.
- The biases are initialized to `0.0` and the weight matrices with truncated normal (standard deviation of `3e-2`)
- The model uses a cross entropy loss.
- No regularization is used.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for Fashion-MNIST.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the multi-layer perceptron test problem instance on Fashion-MNIST.

Fashion-MNIST 2c2d

class deepobs.tensorflow.testproblems.fmnist_2c2d.**fmnist_2c2d**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a two convolutional and two dense layered neural network on Fashion-MNIST.

The network has been adapted from the TensorFlow tutorial and consists of

- two conv layers with ReLUs, each followed by max-pooling
- one fully-connected layers with ReLUs
- 10-unit output layer with softmax
- cross-entropy loss
- No regularization

The weight matrices are initialized with truncated normal (standard deviation of `0.05`) and the biases are initialized to `0.05`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for Fashion-MNIST.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the vanilla CNN test problem on Fashion-MNIST.

Fashion-MNIST VAE

class deepobs.tensorflow.testproblems.fmnist_vae.**fmnist_vae**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a variational autoencoder (VAE) on Fashion-MNIST.

The network has been adapted from the here and consists of an encoder:

- With three convolutional layers with each `64` filters.
- Using a leaky ReLU activation function with $\alpha = 0.3$
- Dropout layers after each convolutional layer with a rate of `0.2`.

and an decoder:

- With two dense layers with `24` and `49` units and leaky ReLU activation.
- With three deconvolutional layers with each `64` filters.
- Dropout layers after the first two deconvolutional layer with a rate of `0.2`.
- A final dense layer with `28 x 28` units and sigmoid activation.

No regularization is used.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for Fashion-MNIST.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term. Will always be `0.0` since no regularizer is used.

set_up()
Set up the VAE test problem on MNIST.

8.2.5 CIFAR-10 Test Problems

CIFAR-10 3c3d

class deepobs.tensorflow.testproblems.cifar10_3c3d.**cifar10_3c3d**(*batch_size*, *weight_decay=0.002*)

DeepOBS test problem class for a three convolutional and three dense layered neural network on Cifar-10.

The network consists of

- thre conv layers with ReLUs, each followed by max-pooling
- two fully-connected layers with `512` and `256` units and ReLU activation
- 10-unit output layer with softmax
- cross-entropy loss
- L2 regularization on the weights (but not the biases) with a default factor of 0.002

The weight matrices are initialized using Xavier initialization and the biases are initialized to `0.0`.

A working training setting is `batch size = 128`, `num_epochs = 100` and SGD with learning rate of `0.01`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `0.002`.

dataset
The DeepOBS data set class for Cifar-10.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the vanilla CNN test problem on Cifar-10.

CIFAR-10 VGG16

class deepobs.tensorflow.testproblems.cifar10_vgg16.**cifar10_vgg16**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the VGG 16 network on Cifar-10.

The CIFAR-10 images are resized to `224` by `224` to fit the input dimension of the original VGG network, which was designed for ImageNet.

Details about the architecture can be found in the original paper. VGG 16 consists of 16 weight layers, of mostly convolutions. The model uses cross-entroy loss. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for Cifar-10.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the VGG 16 test problem on Cifar-10.

CIFAR-10 VGG19

class deepobs.tensorflow.testproblems.cifar10_vgg19.**cifar10_vgg19**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the VGG 19 network on Cifar-10.

The CIFAR-10 images are resized to `224` by `224` to fit the input dimension of the original VGG network, which was designed for ImageNet.

Details about the architecture can be found in the original paper. VGG 19 consists of 19 weight layers, of mostly convolutions. The model uses cross-entroy loss. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for Cifar-10.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the VGG 19 test problem on Cifar-10.

8.2.6 CIFAR-100 Test Problems

CIFAR-100 3c3d

class deepobs.tensorflow.testproblems.cifar100_3c3d.**cifar100_3c3d**(*batch_size*, *weight_decay=0.002*)
DeepOBS test problem class for a three convolutional and three dense layered neural network on Cifar-100.

The network consists of

- thre conv layers with ReLUs, each followed by max-pooling
- two fully-connected layers with `512` and `256` units and ReLU activation
- 100-unit output layer with softmax
- cross-entropy loss
- L2 regularization on the weights (but not the biases) with a default factor of 0.002

The weight matrices are initialized using Xavier initialization and the biases are initialized to `0.0`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `0.002`.

dataset
The DeepOBS data set class for Cifar-100.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the vanilla CNN test problem on Cifar-100.

CIFAR-100 VGG16

class deepobs.tensorflow.testproblems.cifar100_vgg16.**cifar100_vgg16**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the VGG 16 network on Cifar-100.

The CIFAR-100 images are resized to `224` by `224` to fit the input dimension of the original VGG network, which was designed for ImageNet.

Details about the architecture can be found in the original paper. VGG 16 consists of 16 weight layers, of mostly convolutions. The model uses cross-entroy loss. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for Cifar-100.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the VGG 16 test problem on Cifar-100.

CIFAR-100 VGG19

class deepobs.tensorflow.testproblems.cifar100_vgg19.**cifar100_vgg19**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the VGG 19 network on Cifar-100.

The CIFAR-100 images are resized to `224` by `224` to fit the input dimension of the original VGG network, which was designed for ImageNet.

Details about the architecture can be found in the original paper. VGG 19 consists of 19 weight layers, of mostly convolutions. The model uses cross-entroy loss. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for Cifar-100.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the VGG 19 test problem on Cifar-100.

CIFAR-100 All-CNN-C

class deepobs.tensorflow.testproblems.cifar100_allcnnc.**cifar100_allcnnc**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the All Convolutional Neural Network C on Cifar-100.

Details about the architecture can be found in the original paper.

The paper does not comment on initialization; here we use Xavier for conv filters and constant 0.1 for biases.

A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

The reference training parameters from the paper are `batch size = 256`, `num_epochs = 350` using the Momentum optimizer with $\mu = 0.9$ and an initial learning rate of $\alpha = 0.05$ and decrease by a factor of `10` after `200`, `250` and `300` epochs.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for Cifar-100.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the All CNN C test problem on Cifar-100.

CIFAR-100 WideResNet 40-4

class deepobs.tensorflow.testproblems.cifar100_wrn404.**cifar100_wrn404**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the Wide Residual Network 40-4 architecture for CIFAR-100.

Details about the architecture can be found in the original paper. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Training settings recommenden in the original paper: `batch size = 128`, `num_epochs = 200` using the Momentum optimizer with $\mu = 0.9$ and an initial learning rate of `0.1` with a decrease by `0.2` after `60`, `120` and `160` epochs.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for Cifar-100.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the Wide ResNet 40-4 test problem on Cifar-100.

8.2.7 SVHN Test Problems

SVHN 3c3d

class deepobs.tensorflow.testproblems.svhn_3c3d.**svhn_3c3d**(*batch_size*, *weight_decay=0.002*)

DeepOBS test problem class for a three convolutional and three dense layered neural network on SVHN.

The network consists of

- thre conv layers with ReLUs, each followed by max-pooling
- two fully-connected layers with `512` and `256` units and ReLU activation
- 10-unit output layer with softmax
- cross-entropy loss
- L2 regularization on the weights (but not the biases) with a default factor of 0.002

The weight matrices are initialized using Xavier initialization and the biases are initialized to `0.0`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `0.002`.

dataset
The DeepOBS data set class for SVHN.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the vanilla CNN test problem on SVHN.

SVHN WideResNet 16-4

class deepobs.tensorflow.testproblems.svhn_wrn164.**svhn_wrn164**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the Wide Residual Network 16-4 architecture for SVHN.

Details about the architecture can be found in the original paper. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Training settings recommenden in the original paper: `batch size = 128`, `num_epochs = 160` using the Momentum optimizer with $\mu = 0.9$ and an initial learning rate of `0.01` with a decrease by `0.1` after `80` and `120` epochs.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for SVHN.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the Wide ResNet 16-4 test problem on SVHN.

8.2.8 ImageNet Test Problems

ImageNet VGG16

class deepobs.tensorflow.testproblems.imagenet_vgg16.**imagenet_vgg16**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the VGG 16 network on ImageNet.

Details about the architecture can be found in the original paper. VGG 16 consists of 16 weight layers, of mostly convolutions. The model uses cross-entroy loss. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for ImageNet.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the VGG 16 test problem on ImageNet.

ImageNet VGG19

class deepobs.tensorflow.testproblems.imagenet_vgg19.**imagenet_vgg19**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the VGG 19 network on ImageNet.

Details about the architecture can be found in the original paper. VGG 19 consists of 19 weight layers, of mostly convolutions. The model uses cross-entroy loss. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for ImageNet.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the VGG 19 test problem on ImageNet.

ImageNet Inception v3

class deepobs.tensorflow.testproblems.imagenet_inception_v3.**imagenet_inception_v3**(*batch_size*, *weight_deca*

DeepOBS test problem class for the Inception version 3 architecture on ImageNet.

Details about the architecture can be found in the original paper.

There are many changes from the paper to the official Tensorflow implementation as well as the model.txt that can be found in the sources of the original paper. We chose to implement the version from Tensorflow (with possibly some minor changes)

In the original paper they trained the network using:

- `100` Epochs.
- Batch size `32`.
- RMSProp with a decay of `0.9` and $\epsilon = 1.0$.
- Initial learning rate `0.045`.
- Learning rate decay every two epochs with exponential rate of `0.94`.
- Gradient clipping with threshold 2.0

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

dataset
The DeepOBS data set class for ImageNet.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the Inception v3 test problem on ImageNet.

8.2.9 Tolstoi Test Problems

Tolstoi Char RNN

class deepobs.tensorflow.testproblems.tolstoi_char_rnn.**tolstoi_char_rnn**(*batch_size*, *weight_decay=None*)

DeepOBS test problem class for a two-layer LSTM for character-level language modelling (Char RNN) on Tolstoi's War and Peace.

Some network characteristics:

- `128` hidden units per LSTM cell

- sequence length `50`
- cell state is automatically stored in variables between subsequent steps
- when the phase placeholder swithches its value from one step to the next, the cell state is set to its zero value (meaning that we set to zero state after each round of evaluation, it is therefore important to set the evaluation interval such that we evaluate after a full epoch.)

Working training parameters are:

- batch size `50`
- `200` epochs
- SGD with a learning rate of ≈ 0.1 works

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

dataset
The DeepOBS data set class for Tolstoi.

train_init_op
A tensorflow operation initializing the test problem for the training phase.

train_eval_init_op
A tensorflow operation initializing the test problem for evaluating on training data.

test_init_op
A tensorflow operation initializing the test problem for evaluating on test data.

losses
A tf.Tensor of shape (batch_size,) containing the per-example loss values.

regularizer
A scalar tf.Tensor containing a regularization term.

accuracy
A scalar tf.Tensor containing the mini-batch mean accuracy.

set_up()
Set up the Char RNN test problem instance on Tolstoi.

8.3 Runner

Runner take care of the actual training process in DeepOBS. They also log performance statistics such as the loss and accuracy on the test and training data set.

The output of those runners is saved into `JSON` files and optionally also TensorFlow output files that can be plotted in real-time using *Tensorboard*.

8.3.1 TF Runner

The base class for all TensorFlow Runner.

class deepobs.tensorflow.runners.**TFRunner**(*optimizer_class*, *hyperparameter_names*)

Bases: deepobs.abstract_runner.abstract_runner.Runner

__init__(*optimizer_class*, *hyperparameter_names*)

Creates a new Runner instance

Parameters

- **optimizer_class** – The optimizer class of the optimizer that is run on the testproblems. For PyTorch this must be a subclass of torch.optim.Optimizer. For TensorFlow a subclass of tf.train.Optimizer.
- **hyperparameter_names** – A nested dictionary that lists all hyperparameters of the optimizer, their type and their default values (if they have any).

Example

```
>>> optimizer_class = tf.train.MomentumOptimizer
>>> hyperparms = {'lr': {'type': float},
>>>    'momentum': {'type': float, 'default': 0.99},
>>>    'uses_nesterov': {'type': bool, 'default': False}}
>>> runner = StandardRunner(optimizer_class, hyperparms)
```

static create_testproblem(*testproblem*, *batch_size*, *weight_decay*, *random_seed*)

Sets up the deepobs.tensorflow.testproblems.testproblem instance.

Parameters

- **testproblem** (*str*) – The name of the testproblem.
- **batch_size** (*int*) – Batch size that is used for training
- **weight_decay** (*float*) – Regularization factor
- **random_seed** (*int*) – The random seed of the framework

Returns An instance of deepobs.pytorch.testproblems.testproblem

Return type deepobs.tensorflow.testproblems.testproblem

static evaluate(*tproblem*, *sess*, *loss*, *phase*)

Computes average loss and accuracy in the evaluation phase. :param tproblem: The testproblem instance. :type tproblem: deepobs.tensorflow.testproblems.testproblem :param sess: The current TensorFlow Session. :type sess: tensorflow.Session :param loss: The TensorFlow operation that computes the loss. :param phase: The phase of the evaluation. Muste be one of 'TRAIN', 'VALID' or 'TEST' :type phase: str

static init_summary(*loss*, *learning_rate_var*, *batch_size*, *tb_log_dir*)

Initializes the tensorboard summaries

parse_args(*testproblem*, *hyperparams*, *batch_size*, *num_epochs*, *random_seed*, *data_dir*, *output_dir*, *weight_decay*, *no_logs*, *train_log_interval*, *print_train_iter*, *tb_log*, *tb_log_dir*, *training_params*)

Constructs an argparse.ArgumentParser and parses the arguments from command line.

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.

- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **training_params** (*dict*) – Kwargs for the training method.

Returns A dicionary of all arguments.

Return type dict

run(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, *skip_if_exists=False*, ***training_params*)

Runs a testproblem with the optimizer_class. Has the following tasks:

1. setup testproblem
2. run the training (must be implemented by subclass)
3. merge and write output

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.

- **skip_if_exists** (*bool*) – Skip training if the output already exists.
- **training_params** (*dict*) – Kwargs for the training method.

Returns {<...meta data...>, 'test_losses' : test_losses, 'valid_losses': valid_losses 'train_losses': train_losses, 'test_accuracies': test_accuracies, 'valid_accuracies': valid_accuracies 'train_accuracies': train_accuracies, } where <...meta data...> stores the run args.

Return type dict

run_exists(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, ***training_params*)

Return whether output file for this run already exists.

Parameters **run method.** (*See*) –

Returns The first parameter is *True* if the *.json* output file already exists, else *False*. The list contains the paths to the files that match the run.

Return type bool, list(str)

training(*tproblem*, *hyperparams*, *num_epochs*, *print_train_iter*, *train_log_interval*, *tb_log*, *tb_log_dir*, ***training_params*)

Performs the training and stores the metrices.

Parameters

- **tproblem** (*deepobs.[tensorflow/pytorch]testproblems.testproblem*) – The testproblem instance to train on.
- **hyperparams** (*dict*) – The optimizer hyperparameters to use for the training.
- **num_epochs** (*int*) – The number of training epochs.
- **print_train_iter** (*bool*) – Whether to print the training progress at every train_log_interval
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- ****training_params** (*dict*) – Kwargs for additional training parameters that are implemented by subclass.

Returns The logged metrices. Is of the form: {'test_losses' : [...], 'valid_losses': [...], 'train_losses': [...], 'test_accuracies': [...], 'valid_accuracies': [...], 'train_accuracies': [...] } where the metrices values are lists that were filled during training.

Return type dict

static write_output(*output*, *run_folder_name*, *file_name*)

Writes the JSON output.

Parameters

- **output** (*dict*) – Output of the training loop of the runner.
- **run_folder_name** (*str*) – The name of the output folder.
- **file_name** (*str*) – The file name where the output is written to.

static write_per_epoch_summary(*sess*, *loss_*, *acc_*, *current_step*, *per_epoch_summaries*, *summary_writer*, *phase*)
Writes the tensorboard epoch summary

static write_per_iter_summary(*sess*, *per_iter_summaries*, *summary_writer*, *current_step*)
Writes the tensorboard iteration summary

8.3.2 Standard Runner

class deepobs.tensorflow.runners.**StandardRunner**(*optimizer_class*, *hyperparameter_names*)
Bases: deepobs.tensorflow.runners.runner.TFRunner

__init__(*optimizer_class*, *hyperparameter_names*)
Creates a new Runner instance

Parameters

- **optimizer_class** – The optimizer class of the optimizer that is run on the testproblems. For PyTorch this must be a subclass of torch.optim.Optimizer. For TensorFlow a subclass of tf.train.Optimizer.
- **hyperparameter_names** – A nested dictionary that lists all hyperparameters of the optimizer, their type and their default values (if they have any).

Example

```
>>> optimizer_class = tf.train.MomentumOptimizer
>>> hyperparms = {'lr': {'type': float},
>>>     'momentum': {'type': float, 'default': 0.99},
>>>     'uses_nesterov': {'type': bool, 'default': False}}
>>> runner = StandardRunner(optimizer_class, hyperparms)
```

static create_testproblem(*testproblem*, *batch_size*, *weight_decay*, *random_seed*)
Sets up the deepobs.tensorflow.testproblems.testproblem instance.

Parameters

- **testproblem** (*str*) – The name of the testproblem.
- **batch_size** (*int*) – Batch size that is used for training
- **weight_decay** (*float*) – Regularization factor
- **random_seed** (*int*) – The random seed of the framework

Returns An instance of deepobs.pytorch.testproblems.testproblem

Return type deepobs.tensorflow.testproblems.testproblem

static evaluate(*tproblem*, *sess*, *loss*, *phase*)
Computes average loss and accuracy in the evaluation phase. :param tproblem: The testproblem instance. :type tproblem: deepobs.tensorflow.testproblems.testproblem :param sess: The current TensorFlow Session. :type sess: tensorflow.Session :param loss: The TensorFlow operation that computes the loss. :param phase: The phase of the evaluation. Muste be one of 'TRAIN', 'VALID' or 'TEST' :type phase: str

static init_summary(*loss*, *learning_rate_var*, *batch_size*, *tb_log_dir*)
Initializes the tensorboard summaries

parse_args (*testproblem*, *hyperparams*, *batch_size*, *num_epochs*, *random_seed*, *data_dir*, *output_dir*, *weight_decay*, *no_logs*, *train_log_interval*, *print_train_iter*, *tb_log*, *tb_log_dir*, *training_params*)

Constructs an argparse.ArgumentParser and parses the arguments from command line.

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **training_params** (*dict*) – Kwargs for the training method.

Returns A dicionary of all arguments.

Return type dict

run (*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, *skip_if_exists=False*, ***training_params*)

Runs a testproblem with the optimizer_class. Has the following tasks:

1. setup testproblem
2. run the training (must be implemented by subclass)
3. merge and write output

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.

- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **skip_if_exists** (*bool*) – Skip training if the output already exists.
- **training_params** (*dict*) – Kwargs for the training method.

Returns {<...meta data...>, 'test_losses' : test_losses, 'valid_losses': valid_losses 'train_losses': train_losses, 'test_accuracies': test_accuracies, 'valid_accuracies': valid_accuracies 'train_accuracies': train_accuracies, } where <...meta data...> stores the run args.

Return type dict

run_exists(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, ***training_params*)

Return whether output file for this run already exists.

Parameters **run method.** (*See*) –

Returns The first parameter is *True* if the *.json* output file already exists, else *False*. The list contains the paths to the files that match the run.

Return type bool, list(str)

training(*tproblem*, *hyperparams*, *num_epochs*, *print_train_iter*, *train_log_interval*, *tb_log*, *tb_log_dir*)

Performs the training and stores the metrices.

Parameters

- **tproblem** (*deepobs.[tensorflow/pytorch]testproblems.testproblem*) – The testproblem instance to train on.
- **hyperparams** (*dict*) – The optimizer hyperparameters to use for the training.
- **num_epochs** (*int*) – The number of training epochs.
- **print_train_iter** (*bool*) – Whether to print the training progress at every train_log_interval
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- ****training_params** (*dict*) – Kwargs for additional training parameters that are implemented by subclass.

Returns The logged metrices. Is of the form: {'test_losses' : [...], 'valid_losses': [...], 'train_losses': [...], 'test_accuracies': [...], 'valid_accuracies': [...], 'train_accuracies': [...] } where the metrices values are lists that were filled during training.

Return type dict

static write_output(*output*, *run_folder_name*, *file_name*)
Writes the JSON output.

Parameters

- **output** (*dict*) – Output of the training loop of the runner.
- **run_folder_name** (*str*) – The name of the output folder.
- **file_name** (*str*) – The file name where the output is written to.

static write_per_epoch_summary(*sess*, *loss_*, *acc_*, *current_step*, *per_epoch_summaries*, *summary_writer*, *phase*)
Writes the tensorboard epoch summary

static write_per_iter_summary(*sess*, *per_iter_summaries*, *summary_writer*, *current_step*)
Writes the tensorboard iteration summary

8.3.3 Learning Rate Schedule Runner

class deepobs.tensorflow.runners.**LearningRateScheduleRunner**(*optimizer_class*, *hyperparameter_names*)

Bases: deepobs.tensorflow.runners.runner.TFRunner

__init__(*optimizer_class*, *hyperparameter_names*)
Creates a new Runner instance

Parameters

- **optimizer_class** – The optimizer class of the optimizer that is run on the testproblems. For PyTorch this must be a subclass of torch.optim.Optimizer. For TensorFlow a subclass of tf.train.Optimizer.
- **hyperparameter_names** – A nested dictionary that lists all hyperparameters of the optimizer, their type and their default values (if they have any).

Example

```
>>> optimizer_class = tf.train.MomentumOptimizer
>>> hyperparms = {'lr': {'type': float},
>>>     'momentum': {'type': float, 'default': 0.99},
>>>     'uses_nesterov': {'type': bool, 'default': False}}
>>> runner = StandardRunner(optimizer_class, hyperparms)
```

static create_testproblem(*testproblem*, *batch_size*, *weight_decay*, *random_seed*)
Sets up the deepobs.tensorflow.testproblems.testproblem instance.

Parameters

- **testproblem** (*str*) – The name of the testproblem.
- **batch_size** (*int*) – Batch size that is used for training
- **weight_decay** (*float*) – Regularization factor
- **random_seed** (*int*) – The random seed of the framework

Returns An instance of deepobs.pytorch.testproblems.testproblem

Return type deepobs.tensorflow.testproblems.testproblem

static evaluate(*tproblem*, *sess*, *loss*, *phase*)
Computes average loss and accuracy in the evaluation phase. :param tproblem: The testproblem instance. :type tproblem: deepobs.tensorflow.testproblems.testproblem :param sess: The current TensorFlow Session. :type sess: tensorflow.Session :param loss: The TensorFlow operation that computes the loss. :param phase: The phase of the evaluation. Muste be one of 'TRAIN', 'VALID' or 'TEST' :type phase: str

static init_summary(*loss*, *learning_rate_var*, *batch_size*, *tb_log_dir*)
Initializes the tensorboard summaries

parse_args(*testproblem*, *hyperparams*, *batch_size*, *num_epochs*, *random_seed*, *data_dir*, *output_dir*, *weight_decay*, *no_logs*, *train_log_interval*, *print_train_iter*, *tb_log*, *tb_log_dir*, *training_params*)
Constructs an argparse.ArgumentParser and parses the arguments from command line.

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **training_params** (*dict*) – Kwargs for the training method.

Returns A dicionary of all arguments.

Return type dict

run(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, *skip_if_exists=False*, ***training_params*)

Runs a testproblem with the optimizer_class. Has the following tasks:

1. setup testproblem
2. run the training (must be implemented by subclass)
3. merge and write output

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **skip_if_exists** (*bool*) – Skip training if the output already exists.
- **training_params** (*dict*) – Kwargs for the training method.

Returns {<...meta data...>, 'test_losses' : test_losses, 'valid_losses': valid_losses 'train_losses': train_losses, 'test_accuracies': test_accuracies, 'valid_accuracies': valid_accuracies 'train_accuracies': train_accuracies, } where <...meta data...> stores the run args.

Return type dict

run_exists(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, ***training_params*)

Return whether output file for this run already exists.

Parameters **run method.** (*See*) –

Returns The first parameter is *True* if the *.json* output file already exists, else *False*. The list contains the paths to the files that match the run.

Return type bool, list(str)

training(*tproblem*, *hyperparams*, *num_epochs*, *print_train_iter*, *train_log_interval*, *tb_log*, *tb_log_dir*, *lr_sched_epochs=None*, *lr_sched_factors=None*)

Performs the training and stores the metrices.

Parameters

- **tproblem** (*deepobs.[tensorflow/pytorch]testproblems.testproblem*) – The testproblem instance to train on.
- **hyperparams** (*dict*) – The optimizer hyperparameters to use for the training.
- **num_epochs** (*int*) – The number of training epochs.
- **print_train_iter** (*bool*) – Whether to print the training progress at every train_log_interval

- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **lr_sched_epochs** (*list*) – The epochs where to adjust the learning rate.
- **lr_sched_factors** (*list*) – The corresponding factors by which to adjust the learning rate.

Returns The logged metrices. Is of the form: {'test_losses' : [...], 'valid_losses': [...], 'train_losses': [...], 'test_accuracies': [...], 'valid_accuracies': [...], 'train_accuracies': [...] } where the metrices values are lists that were filled during training.

Return type dict

static write_output(*output*, *run_folder_name*, *file_name*)
Writes the JSON output.

Parameters

- **output** (*dict*) – Output of the training loop of the runner.
- **run_folder_name** (*str*) – The name of the output folder.
- **file_name** (*str*) – The file name where the output is written to.

static write_per_epoch_summary(*sess*, *loss_*, *acc_*, *current_step*, *per_epoch_summaries*, *summary_writer*, *phase*)
Writes the tensorboard epoch summary

static write_per_iter_summary(*sess*, *per_iter_summaries*, *summary_writer*, *current_step*)
Writes the tensorboard iteration summary

8.4 Config

The TensorFlow specific config of DeepOBS.

deepobs.tensorflow.config.**set_float_dtype**(*dtype*)

CHAPTER 9

PyTorch

9.1 Data Sets

Currently DeepOBS includes nine different data sets. Each data set inherits from the same base class with the following signature.

class deepobs.pytorch.datasets.dataset.**DataSet**(*batch_size*)

Base class for DeepOBS data sets.

Parameters **batch_size** (*int*) – The mini-batch size to use.

_make_train_and_valid_dataloader()

Creates a torch data loader for the training and validation data with batches of size batch_size.

_make_train_eval_dataloader()

Creates a torch data loader for the training evaluation data with batches of size batch_size.

_make_test_dataloader()

Creates a torch data loader for the test data with batches of size batch_size.

_pin_memory

Whether to pin memory for the dataloaders. Defaults to 'False' if 'cuda' is not the current device.

_num_workers

The number of workers used for the dataloaders. It's value is set to the global variable NUM_WORKERS.

_train_dataloader

A torch.utils.data.DataLoader instance that holds the training data.

_valid_dataloader

A torch.utils.data.DataLoader instance that holds the validation data.

_train_eval_dataloader

A torch.utils.data.DataLoader instance that holds the training evaluation data.

_test_dataloader

A torch.utils.data.DataLoader instance that holds the test data.

After selecting a data set (i.e. CIFAR-10, Fahion-MNIST, etc.) we define four internal PyTorch dataloaders (i.e. *train*, *train_eval*, *valid* and *test*). Those are splits of the original data set that are used for training, hyperparameter tuning and performance evaluation. These internal data sets (also called DeepOBS data sets) are created as shown in the illustration below.

9.1.1 Quadratic Data Set

class deepobs.pytorch.datasets.quadratic.**quadratic**(*batch_size*, *dim=100*, *train_size=1000*, *noise_level=0.6*)

DeepOBS data set class to create an n dimensional stochastic quadratic testproblem.

This toy data set consists of a fixed number (`train_size`) of iid draws from a zero-mean normal distribution in `dim` dimensions with isotropic covariance specified by `noise_level`.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`1000` for train and test) the remainder is dropped in each epoch (after shuffling).
- **dim** (*int*) – Dimensionality of the quadratic. Defaults to `100`.
- **train_size** (*int*) – Size of the dataset; will be used for train, train eval and test datasets. Defaults to `1000`.
- **noise_level** (*float*) – Standard deviation of the data points around the mean. The data points are drawn from a Gaussian distribution. Defaults to `0.6`.

9.1.2 MNIST Data Set

class deepobs.pytorch.datasets.mnist.**mnist**(*batch_size*, *train_eval_size=10000*)

DeepOBS data set class for the MNIST data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`60 000` for train, `10 000` for test) the remainder is dropped in each epoch (after shuffling).
- **train_eval_size** (*int*) – Size of the train eval data set. Defaults to `10 000` the size of the test set.

_make_dataloader()

A helper that is shared by all three data loader methods.

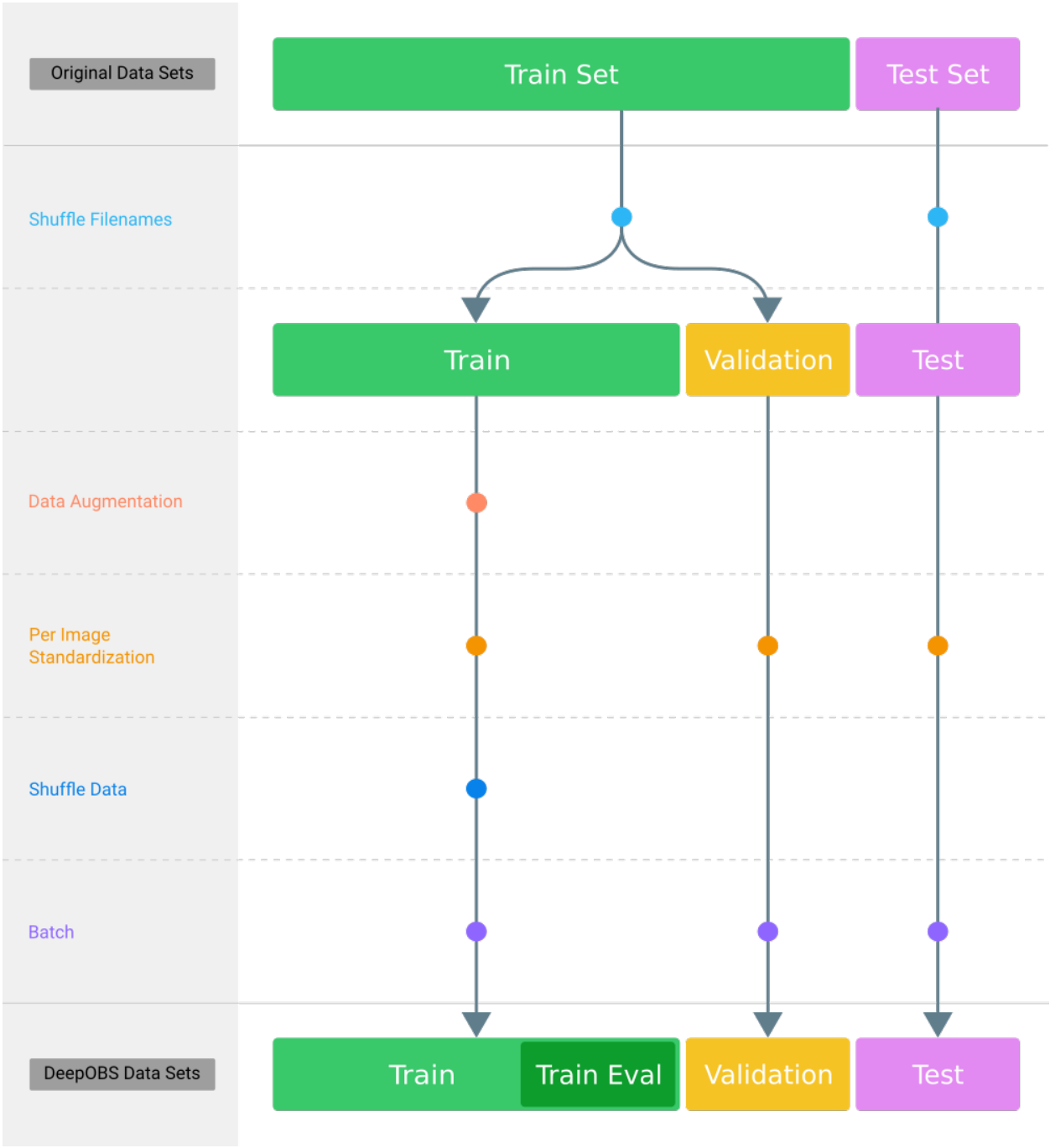
9.1.3 FMNIST Data Set

class deepobs.pytorch.datasets.fmnist.**fmnist**(*batch_size*, *train_eval_size=10000*)

DeepOBS data set class for the Fashion-MNIST (FMNIST) data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`60 000` for train, `10 000` for test) the remainder is dropped in each epoch (after shuffling).


Original Data Sets
Train Set
Test Set
Shuffle Filenames
Train
Validation
Test
Data Augmentation
Per Image Standardization
Shuffle Data
Batch
DeepOBS Data Sets
Train
Train Eval
Validation
Test

- **train_eval_size** (*int*) – Size of the train eval data set. Defaults to `10 000` the size of the test set.

9.1.4 CIFAR-10 Data Set

class deepobs.pytorch.datasets.cifar10.**cifar10**(*batch_size*, *data_augmentation=True*, *train_eval_size=10000*)

DeepOBS data set class for the CIFAR-10 data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`50 000` for train, `10 000` for test) the remainder is dropped in each epoch (after shuffling).
- **data_augmentation** (*bool*) – If `True` some data augmentation operations (random crop window, horizontal flipping, lighting augmentation) are applied to the training data (but not the test data).
- **train_eval_size** (*int*) – Size of the train eval data set. Defaults to `10 000` the size of the test set.

_make_dataloader()

A helper that is shared by all three data loader methods.

9.1.5 CIFAR-100 Data Set

class deepobs.pytorch.datasets.cifar100.**cifar100**(*batch_size*, *data_augmentation=True*, *train_eval_size=10000*)

DeepOBS data set class for the CIFAR-100 data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`50 000` for train, `10 000` for test) the remainder is dropped in each epoch (after shuffling).
- **data_augmentation** (*bool*) – If `True` some data augmentation operations (random crop window, horizontal flipping, lighting augmentation) are applied to the training data (but not the test data).
- **train_eval_size** (*int*) – Size of the train eval data set. Defaults to `10 000` the size of the test set.

_make_dataloader()

A helper that is shared by all three data loader methods.

9.1.6 SVHN Data Set

class deepobs.pytorch.datasets.svhn.**svhn**(*batch_size*, *data_augmentation=True*, *train_eval_size=26032*)

DeepOBS data set class for the Street View House Numbers (SVHN) data set.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size (`73 000` for train, `26 000` for test) the remainder is dropped in each epoch (after shuffling).
- **data_augmentation** (*bool*) – If `True` some data augmentation operations (random crop window, lighting augmentation) are applied to the training data (but not the test data).
- **train_eval_size** (*int*) – Size of the train eval dataset. Defaults to `26 000` the size of the test set.

9.1.7 Tolstoi Data Set

class deepobs.pytorch.datasets.tolstoi.**tolstoi**(*batch_size*, *seq_length=50*, *train_eval_size=653237*)

DeepOBS data set class for character prediction on *War and Peace* by Leo Tolstoi.

Parameters

- **batch_size** (*int*) – The mini-batch size to use. Note that, if `batch_size` is not a divider of the dataset size the remainder is dropped in each epoch (after shuffling).
- **seq_length** (*int*) – Sequence length to be modeled in each step. Defaults to `50`.
- **train_eval_size** (*int*) – Size of the train eval dataset. Defaults to `653 237`, the size of the test set.

9.2 Test Problems

Currently DeepOBS includes twenty-six different test problems. A test problem is given by a combination of a data set and a model and is characterized by its loss function.

Each test problem inherits from the same base class with the following signature.

class deepobs.pytorch.testproblems.**TestProblem**(*batch_size*, *weight_decay=None*)

Base class for DeepOBS test problems.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay (L2-regularization) factor to use. If not specified, the test problems revert to their respective defaults. Note: Some test problems do not use regularization and this value will be ignored in such a case.

_batch_size
Batch_size for the data of this test problem.

_weight_decay
The regularization factor for this test problem

data
The dataset used by the test problem (datasets.DataSet instance).

loss_function
The loss function for this test problem.

net
The torch module (the neural network) that is trained.

train_init_op()
Initializes the test problem for the training phase.

train_eval_init_op()
Initializes the test problem for evaluating on training data.

test_init_op()
Initializes the test problem for evaluating on test data.

_get_next_batch()
Returns the next batch of data of the current phase.

get_batch_loss_and_accuracy()
Calculates the loss and accuracy of net on the next batch of the current phase.

set_up()
Sets all public attributes.

get_batch_loss_and_accuracy(*reduction='mean'*, *add_regularization_if_available=True*)
Gets a new batch and calculates the loss and accuracy (if available) on that batch.

Parameters

- **reduction** (*str*) – The reduction that is used for returning the loss. Can be 'mean', 'sum' or 'none' in which case each indivual loss in the mini-batch is returned as a tensor.
- **add_regularization_if_available** (*bool*) – If true, regularization is added to the loss.

Returns loss and accuracy of the model on the current batch.

Return type float/torch.tensor, float

get_batch_loss_and_accuracy_func(*reduction='mean'*, *add_regularization_if_available=True*)
Get new batch and create forward function that calculates loss and accuracy (if available) on that batch. This is a default implementation for image classification. Testproblems with different calculation routines (e.g. RNNs) overwrite this method accordingly.

Parameters

- **reduction** (*str*) – The reduction that is used for returning the loss. Can be 'mean', 'sum' or 'none' in which case each indivual loss in the mini-batch is returned as a tensor.
- **add_regularization_if_available** (*bool*) – If true, regularization is added to the loss.

Returns The function that calculates the loss/accuracy on the current batch.

Return type callable

get_regularization_groups()
Creates regularization groups for the parameters.

Returns A dictionary where the key is the regularization factor and the value is a list of parameters.

Return type dict

get_regularization_loss()
Returns the current regularization loss of the network based on the parameter groups.

Returns If no regularzations is applied, it returns the integer 0. Else a torch.tensor that holds the regularization loss.

Return type int or torch.tensor

set_up()
Sets up the test problem.

test_init_op()
Initializes the testproblem instance to test mode. I.e. sets the iterator to the test set and sets the model to eval mode.

train_eval_init_op()
Initializes the testproblem instance to train eval mode. I.e. sets the iterator to the train evaluation set and sets the model to eval mode.

train_init_op()
Initializes the testproblem instance to train mode. I.e. sets the iterator to the training set and sets the model to train mode.

valid_init_op()
Initializes the testproblem instance to validation mode. I.e. sets the iterator to the validation set and sets the model to eval mode.

Note: Some of the test problems described here are based on more general implementations. For example the Wide ResNet 40-4 network on Cifar-100 is based on the general Wide ResNet architecture which is also implemented. Therefore, it is very easy to include new Wide ResNets if necessary.

9.2.1 Quadratic Test Problems

DeepOBS includes a stochastic quadratic problem with an eigenspectrum similar to what has been reported for neural networks.

Other stochastic quadratic problems (of different dimensionality or with a different Hessian structure) can be created easily using the `net_quadratic_deep` class.

class deepobs.pytorch.testproblems.testproblems_modules.**net_quadratic_deep**(*dim*, *Hessian*)
This arhcitecture creates an output which corresponds to a loss functions of the form

$0.5 * (\theta - x)^T * Q * (\theta - x)$

with Hessian `Q` and "data" `x` coming from the quadratic data set, i.e., zero-mean normal. The parameters are initialized to 1.

Quadratic Deep

class deepobs.pytorch.testproblems.quadratic_deep.**quadratic_deep**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a stochastic quadratic test problem `100`dimensions. The 90 % of the eigenvalues of the Hessian are drawn from theinterval $(0.0, 1.0)$ and the other 10 % are from $(30.0, 60.0)$ simulating an eigenspectrum which has been reported for Deep Learning https://arxiv.org/abs/1611.01838.

This creatis a loss functions of the form

$0.5 * (\theta - x)^T * Q * (\theta - x)$

with Hessian `Q` and "data" `x` coming from the quadratic data set, i.e., zero-mean normal.

Parameters

- **batch_size** (*int*) – Batch size to use.

- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

data
The DeepOBS data set class for the quadratic problem.

loss_function
None. The output of the model is the loss.

net
The DeepOBS subclass of torch.nn.Module that is trained for this tesproblem (net_quadratic_deep).

get_batch_loss_and_accuracy_func(*reduction='mean'*, *add_regularization_if_available=True*)
Get new batch and create forward function that calculates loss and accuracy (if available) on that batch.

Parameters

- **reduction** (*str*) – The reduction that is used for returning the loss. Can be 'mean', 'sum' or 'none' in which case each indivual loss in the mini-batch is returned as a tensor.
- **add_regularization_if_available** (*bool*) – If true, regularization is added to the loss.

Returns The function that calculates the loss/accuracy on the current batch.

Return type callable

set_up()
Sets up the test problem.

9.2.2 MNIST Test Problems

MNIST MLP

class deepobs.pytorch.testproblems.mnist_mlp.**mnist_mlp**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a multi-layer perceptron neural network on Fashion-MNIST.

The network is build as follows:

- Four fully-connected layers with `1000`, `500`, `100` and `10` units per layer.
- The first three layers use ReLU activation, and the last one a softmax activation.
- The biases are initialized to `0.0` and the weight matrices with truncated normal (standard deviation of `3e-2`)
- The model uses a cross entropy loss.
- No regularization is used.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) –

  No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

  data
  The DeepOBS data set class for Fashion-MNIST.

loss_function
The loss function for this testproblem is torch.nn.CrossEntropyLoss()

net
The DeepOBS subclass of torch.nn.Module that is trained for this tesproblem (net_mlp).

set_up()
Sets up the vanilla CNN test problem on MNIST.

MNIST 2c2d

class deepobs.pytorch.testproblems.mnist_2c2d.**mnist_2c2d**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a two convolutional and two dense layered neural network on MNIST.

The network has been adapted from the TensorFlow tutorial and consists of

- two conv layers with ReLUs, each followed by max-pooling
- one fully-connected layers with ReLUs
- 10-unit output layer with softmax
- cross-entropy loss
- No regularization

The weight matrices are initialized with truncated normal (standard deviation of `0.05`) and the biases are initialized to `0.05`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

set_up()
Sets up the vanilla CNN test problem on MNIST.

MNIST VAE

class deepobs.pytorch.testproblems.mnist_vae.**mnist_vae**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a variational autoencoder (VAE) on MNIST.

The network has been adapted from the here and consists of an encoder:

- With three convolutional layers with each `64` filters.
- Using a leaky ReLU activation function with $\alpha = 0.3$
- Dropout layers after each convolutional layer with a rate of `0.2`.

and an decoder:

- With two dense layers with `24` and `49` units and leaky ReLU activation.
- With three deconvolutional layers with each `64` filters.
- Dropout layers after the first two deconvolutional layer with a rate of `0.2`.
- A final dense layer with `28 x 28` units and sigmoid activation.

No regularization is used.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

data
The DeepOBS data set class for MNIST.

loss_function
The loss function for this testproblem (vae_loss_function as defined in testproblem_utils)

net
The DeepOBS subclass of torch.nn.Module that is trained for this tesproblem (net_vae).

get_batch_loss_and_accuracy_func(*reduction='mean'*, *add_regularization_if_available=True*)
Get new batch and create forward function that calculates loss and accuracy (if available) on that batch.

Parameters

- **reduction** (*str*) – The reduction that is used for returning the loss. Can be 'mean', 'sum' or 'none' in which case each indivual loss in the mini-batch is returned as a tensor.
- **add_regularization_if_available** (*bool*) – If true, regularization is added to the loss.

Returns The function that calculates the loss/accuracy on the current batch.

Return type callable

set_up()
Sets up the vanilla CNN test problem on MNIST.

9.2.3 Fashion-MNIST Test Problems

Fashion-MNIST MLP

class deepobs.pytorch.testproblems.fmnist_mlp.**fmnist_mlp**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a multi-layer perceptron neural network on Fashion-MNIST.

The network is build as follows:

- Four fully-connected layers with `1000`, `500`, `100` and `10` units per layer.
- The first three layers use ReLU activation, and the last one a softmax activation.
- The biases are initialized to `0.0` and the weight matrices with truncated normal (standard deviation of `3e-2`)
- The model uses a cross entropy loss.
- No regularization is used.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) –

No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

data
The DeepOBS data set class for Fashion-MNIST.

loss_function
The loss function for this testproblem is torch.nn.CrossEntropyLoss()

net
The DeepOBS subclass of torch.nn.Module that is trained for this tesproblem (net_mlp).

set_up()
Sets up the vanilla MLP test problem on Fashion-MNIST.

Fashion-MNIST 2c2d

class deepobs.pytorch.testproblems.fmnist_2c2d.**fmnist_2c2d**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a two convolutional and two dense layered neural network on Fashion-MNIST.

The network has been adapted from the TensorFlow tutorial and consists of

- two conv layers with ReLUs, each followed by max-pooling
- one fully-connected layers with ReLUs
- 10-unit output layer with softmax
- cross-entropy loss
- No regularization

The weight matrices are initialized with truncated normal (standard deviation of `0.05`) and the biases are initialized to `0.05`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

set_up()
Sets up the vanilla CNN test problem on Fashion-MNIST.

Fashion-MNIST VAE

class deepobs.pytorch.testproblems.fmnist_vae.**fmnist_vae**(*batch_size*, *weight_decay=None*)
DeepOBS test problem class for a variational autoencoder (VAE) on Fashion-MNIST.

The network has been adapted from the here and consists of an encoder:

- With three convolutional layers with each `64` filters.
- Using a leaky ReLU activation function with $\alpha = 0.3$
- Dropout layers after each convolutional layer with a rate of `0.2`.

and an decoder:

- With two dense layers with `24` and `49` units and leaky ReLU activation.

- With three deconvolutional layers with each `64` filters.
- Dropout layers after the first two deconvolutional layer with a rate of `0.2`.
- A final dense layer with `28 x 28` units and sigmoid activation.

No regularization is used.

Parameters

- **`batch_size`** (*`int`*) – Batch size to use.
- **`weight_decay`** (*`float`*) – No weight decay (L2-regularization) is used in this test problem. Defaults to `None` and any input here is ignored.

`data`
The DeepOBS data set class for Fashion-MNIST.

`loss_function`
The loss function for this testproblem (vae_loss_function as defined in testproblem_utils)

`net`
The DeepOBS subclass of torch.nn.Module that is trained for this tesproblem (net_vae).

`get_batch_loss_and_accuracy_func`(*reduction='mean'*, *add_regularization_if_available=True*)
Gets a new batch and calculates the loss and accuracy (if available) on that batch. This is a default implementation for image classification. Testproblems with different calculation routines (e.g. RNNs) overwrite this method accordingly.

Parameters **`return_forward_func`** (*`bool`*) – If `True`, the call also returns a function that calculates the loss on the current batch. Can be used if you need to access the forward path twice.

Returns loss and accuracy of the model on the current batch. If `return_forward_func` is `True` it also returns the function that calculates the loss on the current batch.

Return type float, float, (callable)

`set_up`()
Sets up the test problem.

9.2.4 CIFAR-10 Test Problems

CIFAR-10 3c3d

`class` `deepobs.pytorch.testproblems.cifar10_3c3d.`**`cifar10_3c3d`**(*batch_size*, *weight_decay=0.002*)
DeepOBS test problem class for a three convolutional and three dense layered neural network on Cifar-10.

The network consists of

- thre conv layers with ReLUs, each followed by max-pooling
- two fully-connected layers with `512` and `256` units and ReLU activation
- 10-unit output layer with softmax
- cross-entropy loss
- L2 regularization on the weights (but not the biases) with a default factor of 0.002

The weight matrices are initialized using Xavier initialization and the biases are initialized to `0.0`.

A working training setting is `batch size = 128`, `num_epochs = 100` and SGD with learning rate of `0.01`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `0.002`.

data
The DeepOBS data set class for Cifar-10.

loss_function
The loss function for this testproblem is torch.nn.CrossEntropyLoss()

net
The DeepOBS subclass of torch.nn.Module that is trained for this tesproblem (net_cifar10_3c3d).

get_regularization_loss()
Returns the current regularization loss of the network state.

get_regularization_groups()
Creates regularization groups for the parameters.

Returns A dictionary where the key is the regularization factor and the value is a list of parameters.

Return type dict

set_up()
Set up the vanilla CNN test problem on Cifar-10.

9.2.5 CIFAR-100 Test Problems

CIFAR-100 3c3d

class deepobs.pytorch.testproblems.cifar100_3c3d.**cifar100_3c3d**(*batch_size*, *weight_decay=0.002*)
DeepOBS test problem class for a three convolutional and three dense layered neural network on Cifar-100.

The network consists of

- thre conv layers with ReLUs, each followed by max-pooling
- two fully-connected layers with `512` and `256` units and ReLU activation
- 100-unit output layer with softmax
- cross-entropy loss
- L2 regularization on the weights (but not the biases) with a default factor of 0.002

The weight matrices are initialized using Xavier initialization and the biases are initialized to `0.0`.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `0.002`.

data
The DeepOBS data set class for Cifar-100.

loss_function
The loss function for this testproblem is torch.nn.CrossEntropyLoss().

net
The DeepOBS subclass of torch.nn.Module that is trained for this tesproblem (net_cifar10_3c3d with 100 outputs).

get_regularization_loss()
Returns the current regularization loss of the network state.

get_regularization_groups()
Creates regularization groups for the parameters.

Returns A dictionary where the key is the regularization factor and the value is a list of parameters.

Return type dict

set_up()
Set up the vanilla CNN test problem on Cifar-100.

CIFAR-100 All-CNN-C

class deepobs.pytorch.testproblems.cifar100_allcnnc.**cifar100_allcnnc**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the All Convolutional Neural Network C on Cifar-100.

Details about the architecture can be found in the original paper.

The paper does not comment on initialization; here we use Xavier for conv filters and constant 0.1 for biases.

A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

The reference training parameters from the paper are `batch size = 256`, `num_epochs = 350` using the Momentum optimizer with $\mu = 0.9$ and an initial learning rate of $\alpha = 0.05$ and decrease by a factor of `10` after `200`, `250` and `300` epochs.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

get_regularization_groups()
Creates regularization groups for the parameters.

Returns A dictionary where the key is the regularization factor and the value is a list of parameters.

Return type dict

set_up()
Set up the All CNN C test problem on Cifar-100.

9.2.6 SVHN Test Problems

SVHN Wide Resnet

class deepobs.pytorch.testproblems.svhn_wrn164.**svhn_wrn164**(*batch_size*, *weight_decay=0.0005*)

DeepOBS test problem class for the Wide Residual Network 16-4 architecture for SVHN.

Details about the architecture can be found in the original paper. A weight decay is used on the weights (but not the biases) which defaults to `5e-4`.

Training settings recommenden in the original paper: `batch size = 128`, `num_epochs = 160` using the Momentum optimizer with $\mu = 0.9$ and an initial learning rate of `0.01` with a decrease by `0.1` after `80` and `120` epochs.

Parameters

- **batch_size** (*int*) – Batch size to use.
- **weight_decay** (*float*) – Weight decay factor. Weight decay (L2-regularization) is used on the weights but not the biases. Defaults to `5e-4`.

get_regularization_groups()

Creates regularization groups for the parameters.

Returns A dictionary where the key is the regularization factor and the value is a list of parameters.

Return type dict

set_up()

Set up the Wide ResNet 16-4 test problem on SVHN.

9.3 Runner

Runner take care of the actual training process in DeepOBS. They also log performance statistics such as the loss and accuracy on the test and training data set.

The output of those runners is saved into `JSON` files and optionally also TensorFlow output files that can be plotted in real-time using *Tensorboard*.

9.3.1 PT Runner

The base class of all PyTorch Runner.

class deepobs.pytorch.runners.**PTRunner**(*optimizer_class*, *hyperparameter_names*)

Bases: deepobs.abstract_runner.abstract_runner.Runner

The abstract class for runner in the pytorch framework.

__init__(*optimizer_class*, *hyperparameter_names*)

Creates a new Runner instance

Parameters

- **optimizer_class** – The optimizer class of the optimizer that is run on the testproblems. For PyTorch this must be a subclass of torch.optim.Optimizer. For TensorFlow a subclass of tf.train.Optimizer.
- **hyperparameter_names** – A nested dictionary that lists all hyperparameters of the optimizer, their type and their default values (if they have any).

Example

```
>>> optimizer_class = tf.train.MomentumOptimizer
>>> hyperparms = {'lr': {'type': float},
>>>    'momentum': {'type': float, 'default': 0.99},
>>>    'uses_nesterov': {'type': bool, 'default': False}}
>>> runner = StandardRunner(optimizer_class, hyperparms)
```

static create_testproblem(*testproblem*, *batch_size*, *weight_decay*, *random_seed*)

Sets up the deepobs.pytorch.testproblems.testproblem instance.

Parameters

- **testproblem** (*str*) – The name of the testproblem.
- **batch_size** (*int*) – Batch size that is used for training
- **weight_decay** (*float*) – Regularization factor
- **random_seed** (*int*) – The random seed of the framework

Returns An instance of deepobs.pytorch.testproblems.testproblem

Return type deepobs.pytorch.testproblems.testproblem

static evaluate(*tproblem*, *phase*)

Evaluates the performance of the current state of the model of the testproblem instance. Has to be called in the beggining of every epoch within the training method. Returns the losses and accuracies.

Parameters

- **tproblem** (*testproblem*) – The testproblem instance to evaluate
- **phase** (*str*) – The phase of the evaluation. Must be one of 'TRAIN', 'VALID' or 'TEST'

Returns The loss of the current state. float: The accuracy of the current state.

Return type float

parse_args(*testproblem*, *hyperparams*, *batch_size*, *num_epochs*, *random_seed*, *data_dir*, *output_dir*, *weight_decay*, *no_logs*, *train_log_interval*, *print_train_iter*, *tb_log*, *tb_log_dir*, *training_params*)

Constructs an argparse.ArgumentParser and parses the arguments from command line.

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.

- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **training_params** (*dict*) – Kwargs for the training method.

Returns A dicionary of all arguments.

Return type dict

run(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, *skip_if_exists=False*, ***training_params*)

Runs a testproblem with the optimizer_class. Has the following tasks:

1. setup testproblem
2. run the training (must be implemented by subclass)
3. merge and write output

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **skip_if_exists** (*bool*) – Skip training if the output already exists.
- **training_params** (*dict*) – Kwargs for the training method.

Returns {<...meta data...>, 'test_losses' : test_losses, 'valid_losses': valid_losses 'train_losses': train_losses, 'test_accuracies': test_accuracies, 'valid_accuracies': valid_accuracies 'train_accuracies': train_accuracies, } where <...meta data...> stores the run args.

Return type dict

run_exists(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, ***training_params*)
Return whether output file for this run already exists.

Parameters **run method.** (*See*) –

Returns The first parameter is *True* if the *.json* output file already exists, else *False*. The list contains the paths to the files that match the run.

Return type bool, list(str)

training(*tproblem*, *hyperparams*, *num_epochs*, *print_train_iter*, *train_log_interval*, *tb_log*, *tb_log_dir*, ***training_params*)
Performs the training and stores the metrices.

Parameters

- **tproblem** (*deepobs.[tensorflow/pytorch]testproblems.testproblem*) – The testproblem instance to train on.
- **hyperparams** (*dict*) – The optimizer hyperparameters to use for the training.
- **num_epochs** (*int*) – The number of training epochs.
- **print_train_iter** (*bool*) – Whether to print the training progress at every train_log_interval
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- ****training_params** (*dict*) – Kwargs for additional training parameters that are implemented by subclass.

Returns The logged metrices. Is of the form: {'test_losses' : [...], 'valid_losses': [...], 'train_losses': [...], 'test_accuracies': [...], 'valid_accuracies': [...], 'train_accuracies': [...] } where the metrices values are lists that were filled during training.

Return type dict

static write_output(*output*, *run_folder_name*, *file_name*)
Writes the JSON output.

Parameters

- **output** (*dict*) – Output of the training loop of the runner.
- **run_folder_name** (*str*) – The name of the output folder.
- **file_name** (*str*) – The file name where the output is written to.

9.3.2 Standard Runner

class deepobs.pytorch.runners.**StandardRunner**(*optimizer_class*, *hyperparameter_names*)
Bases: deepobs.pytorch.runners.runner.PTRunner

A standard runner. Can run a normal training loop with fixed hyperparams. It should be used as a template to implement custom runners.

__init__(*optimizer_class*, *hyperparameter_names*)
Creates a new Runner instance

Parameters

- **optimizer_class** – The optimizer class of the optimizer that is run on the testproblems. For PyTorch this must be a subclass of torch.optim.Optimizer. For TensorFlow a subclass of tf.train.Optimizer.
- **hyperparameter_names** – A nested dictionary that lists all hyperparameters of the optimizer, their type and their default values (if they have any).

Example

```
>>> optimizer_class = tf.train.MomentumOptimizer
>>> hyperparms = {'lr': {'type': float},
>>>    'momentum': {'type': float, 'default': 0.99},
>>>    'uses_nesterov': {'type': bool, 'default': False}}
>>> runner = StandardRunner(optimizer_class, hyperparms)
```

static create_testproblem(*testproblem*, *batch_size*, *weight_decay*, *random_seed*)
Sets up the deepobs.pytorch.testproblems.testproblem instance.

Parameters

- **testproblem** (*str*) – The name of the testproblem.
- **batch_size** (*int*) – Batch size that is used for training
- **weight_decay** (*float*) – Regularization factor
- **random_seed** (*int*) – The random seed of the framework

Returns An instance of deepobs.pytorch.testproblems.testproblem

Return type deepobs.pytorch.testproblems.testproblem

static evaluate(*tproblem*, *phase*)
Evaluates the performance of the current state of the model of the testproblem instance. Has to be called in the beggining of every epoch within the training method. Returns the losses and accuracies.

Parameters

- **tproblem** (*testproblem*) – The testproblem instance to evaluate
- **phase** (*str*) – The phase of the evaluation. Must be one of 'TRAIN', 'VALID' or 'TEST'

Returns The loss of the current state. float: The accuracy of the current state.

Return type float

parse_args(*testproblem*, *hyperparams*, *batch_size*, *num_epochs*, *random_seed*, *data_dir*, *output_dir*, *weight_decay*, *no_logs*, *train_log_interval*, *print_train_iter*, *tb_log*, *tb_log_dir*, *training_params*)
Constructs an argparse.ArgumentParser and parses the arguments from command line.

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training

- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **training_params** (*dict*) – Kwargs for the training method.

Returns A dicionary of all arguments.

Return type dict

run(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, *skip_if_exists=False*, ***training_params*)

Runs a testproblem with the optimizer_class. Has the following tasks:

1. setup testproblem
2. run the training (must be implemented by subclass)
3. merge and write output

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not

- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **skip_if_exists** (*bool*) – Skip training if the output already exists.
- **training_params** (*dict*) – Kwargs for the training method.

Returns {<...meta data...>, 'test_losses' : test_losses, 'valid_losses': valid_losses 'train_losses': train_losses, 'test_accuracies': test_accuracies, 'valid_accuracies': valid_accuracies 'train_accuracies': train_accuracies, } where <...meta data...> stores the run args.

Return type dict

run_exists(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, ***training_params*)

Return whether output file for this run already exists.

Parameters **run method.** (*See*) –

Returns The first parameter is *True* if the *.json* output file already exists, else *False*. The list contains the paths to the files that match the run.

Return type bool, list(str)

training(*tproblem*, *hyperparams*, *num_epochs*, *print_train_iter*, *train_log_interval*, *tb_log*, *tb_log_dir*)

Performs the training and stores the metrices.

Parameters

- **tproblem** (*deepobs.[tensorflow/pytorch]testproblems.testproblem*) – The testproblem instance to train on.
- **hyperparams** (*dict*) – The optimizer hyperparameters to use for the training.
- **num_epochs** (*int*) – The number of training epochs.
- **print_train_iter** (*bool*) – Whether to print the training progress at every train_log_interval
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- ****training_params** (*dict*) – Kwargs for additional training parameters that are implemented by subclass.

Returns The logged metrices. Is of the form: {'test_losses' : [...], 'valid_losses': [...], 'train_losses': [...], 'test_accuracies': [...], 'valid_accuracies': [...], 'train_accuracies': [...] } where the metrices values are lists that were filled during training.

Return type dict

static write_output(*output*, *run_folder_name*, *file_name*)

Writes the JSON output.

Parameters

- **output** (*dict*) – Output of the training loop of the runner.
- **run_folder_name** (*str*) – The name of the output folder.
- **file_name** (*str*) – The file name where the output is written to.

9.3.3 Learning Rate Schedule Runner

class deepobs.pytorch.runners.**LearningRateScheduleRunner**(*optimizer_class*, *hyperparameter_names*)

Bases: deepobs.pytorch.runners.runner.PTRunner

A runner for learning rate schedules. Can run a normal training loop with fixed hyperparams or a learning rate schedule. It should be used as a template to implement custom runners.

__init__(*optimizer_class*, *hyperparameter_names*)

Creates a new Runner instance

Parameters

- **optimizer_class** – The optimizer class of the optimizer that is run on the testproblems. For PyTorch this must be a subclass of torch.optim.Optimizer. For TensorFlow a subclass of tf.train.Optimizer.
- **hyperparameter_names** – A nested dictionary that lists all hyperparameters of the optimizer, their type and their default values (if they have any).

Example

```
>>> optimizer_class = tf.train.MomentumOptimizer
>>> hyperparms = {'lr': {'type': float},
>>>    'momentum': {'type': float, 'default': 0.99},
>>>    'uses_nesterov': {'type': bool, 'default': False}}
>>> runner = StandardRunner(optimizer_class, hyperparms)
```

static create_testproblem(*testproblem*, *batch_size*, *weight_decay*, *random_seed*)

Sets up the deepobs.pytorch.testproblems.testproblem instance.

Parameters

- **testproblem** (*str*) – The name of the testproblem.
- **batch_size** (*int*) – Batch size that is used for training
- **weight_decay** (*float*) – Regularization factor
- **random_seed** (*int*) – The random seed of the framework

Returns An instance of deepobs.pytorch.testproblems.testproblem

Return type deepobs.pytorch.testproblems.testproblem

static evaluate(*tproblem*, *phase*)

Evaluates the performance of the current state of the model of the testproblem instance. Has to be called in the beggining of every epoch within the training method. Returns the losses and accuracies.

Parameters

- **tproblem** (*testproblem*) – The testproblem instance to evaluate
- **phase** (*str*) – The phase of the evaluation. Must be one of 'TRAIN', 'VALID' or 'TEST'

Returns The loss of the current state. float: The accuracy of the current state.

Return type float

parse_args (*testproblem*, *hyperparams*, *batch_size*, *num_epochs*, *random_seed*, *data_dir*, *output_dir*, *weight_decay*, *no_logs*, *train_log_interval*, *print_train_iter*, *tb_log*, *tb_log_dir*, *training_params*)

Constructs an argparse.ArgumentParser and parses the arguments from command line.

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.
- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **training_params** (*dict*) – Kwargs for the training method.

Returns A dicionary of all arguments.

Return type dict

run (*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, *skip_if_exists=False*, ***training_params*)

Runs a testproblem with the optimizer_class. Has the following tasks:

1. setup testproblem
2. run the training (must be implemented by subclass)
3. merge and write output

Parameters

- **testproblem** (*str*) – Name of the testproblem.
- **hyperparams** (*dict*) – The explizit values of the hyperparameters of the optimizer that are used for training
- **batch_size** (*int*) – Mini-batch size for the training data.
- **num_epochs** (*int*) – The number of training epochs.
- **random_seed** (*int*) – The torch random seed.
- **data_dir** (*str*) – The path where the data is stored.

- **output_dir** (*str*) – Path of the folder where the results are written to.
- **weight_decay** (*float*) – Regularization factor for the testproblem.
- **no_logs** (*bool*) – Whether to write the output or not.
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **print_train_iter** (*bool*) – Whether to print the training progress at each train_log_interval.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **skip_if_exists** (*bool*) – Skip training if the output already exists.
- **training_params** (*dict*) – Kwargs for the training method.

Returns {<...meta data...>, 'test_losses' : test_losses, 'valid_losses': valid_losses 'train_losses': train_losses, 'test_accuracies': test_accuracies, 'valid_accuracies': valid_accuracies 'train_accuracies': train_accuracies, } where <...meta data...> stores the run args.

Return type dict

run_exists(*testproblem=None*, *hyperparams=None*, *batch_size=None*, *num_epochs=None*, *random_seed=None*, *data_dir=None*, *output_dir=None*, *weight_decay=None*, *no_logs=None*, *train_log_interval=None*, *print_train_iter=None*, *tb_log=None*, *tb_log_dir=None*, ***training_params*)

Return whether output file for this run already exists.

Parameters **run method.** (*See*) –

Returns The first parameter is *True* if the *.json* output file already exists, else *False*. The list contains the paths to the files that match the run.

Return type bool, list(str)

training(*tproblem*, *hyperparams*, *num_epochs*, *print_train_iter*, *train_log_interval*, *tb_log*, *tb_log_dir*, *lr_sched_epochs=None*, *lr_sched_factors=None*)

Performs the training and stores the metrices.

Parameters

- **tproblem** (*deepobs.[tensorflow/pytorch]testproblems.testproblem*) – The testproblem instance to train on.
- **hyperparams** (*dict*) – The optimizer hyperparameters to use for the training.
- **num_epochs** (*int*) – The number of training epochs.
- **print_train_iter** (*bool*) – Whether to print the training progress at every train_log_interval
- **train_log_interval** (*int*) – Mini-batch interval for logging.
- **tb_log** (*bool*) – Whether to use tensorboard logging or not
- **tb_log_dir** (*str*) – The path where to save tensorboard events.
- **lr_sched_epochs** (*list*) – The epochs where to adjust the learning rate.
- **lr_sched_factors** (*list*) – The corresponding factors by which to adjust the learning rate.

Returns The logged metrices. Is of the form: {'test_losses' : [...], 'valid_losses': [...], 'train_losses': [...], 'test_accuracies': [...], 'valid_accuracies': [...], 'train_accuracies': [...] } where the metrices values are lists that were filled during training.

Return type dict

static write_output(*output*, *run_folder_name*, *file_name*)
Writes the JSON output.

Parameters

- **output** (*dict*) – Output of the training loop of the runner.
- **run_folder_name** (*str*) – The name of the output folder.
- **file_name** (*str*) – The file name where the output is written to.

9.4 Config

The PyTorch specific config of DeepOBS.

deepobs.pytorch.config.**set_num_workers**(*num_workers*)
Sets the number of workers that are used in the torch DataLoaders.

Parameters **num_workers** (*int*) – The number of workers that are used for data loading.

deepobs.pytorch.config.**set_is_deterministic**(*is_deterministic*)
Sets whether PyTorch should try to run deterministic.

Parameters **is_deterministic** (*bool*) – If `True`, this flag sets: `torch.backends.cudnn.deterministic = True torch.backends.cudnn.benchmark = False`. However, full determinism is not guaranteed. For more information, see: https://pytorch.org/docs/stable/notes/randomness.html

deepobs.pytorch.config.**set_default_device**(*device*)
Sets the device on which the PyTorch experiments are run.

Parameters **device** (*str*) – Device on which to run the PyTorch test problems. E.g. 'cuda' or 'cuda:0'

CHAPTER 10

Tuner

10.1 Grid Search

class deepobs.tuner.**GridSearch**(*optimizer_class*, *hyperparam_names*, *grid*, *ressources*, *runner*)

Bases: *deepobs.tuner.tuner.ParallelizedTuner*

A basic Grid Search tuner.

__init__(*optimizer_class*, *hyperparam_names*, *grid*, *ressources*, *runner*)

Parameters **grid** (*dict*) – Holds the discrete values for each hyperparameter as lists.

generate_commands_script(*testproblem*, *run_script*, *output_dir='./results'*, *random_seed=42*, *generation_dir='./command_scripts'*, ***kwargs*)

Parameters

- **testproblem** (*str*) – Testproblem for which to generate commands.
- **run_script** (*str*) – Name the run script that is used from the command line.
- **output_dir** (*str*) – The output path where the execution results are written to.
- **random_seed** (*int*) – The random seed for the tuning.
- **generation_dir** (*str*) – The path to the directory where the generated scripts are written to.

Returns The relative file path to the generated commands script.

Return type str

generate_commands_script_for_testset(*testset*, **args*, ***kwargs*)

Generates command scripts for a whole testset. :param testset: A list of the testproblem strings. :type testset: list

tune(*testproblem*, *output_dir='./results'*, *random_seed=42*, *rerun_best_setting=False*, ***kwargs*)

Tunes the optimizer on the test problem. :param testproblem: The test problem to tune the optimizer on. :type testproblem: str :param output_dir: The output directory for the results. :type output_dir: str

:param random_seed: Random seed for the whole truning process. Every individual run is seeded by it. :type random_seed: int :param rerun_best_setting: Whether to automatically rerun the best setting with 10 different seeds. :type rerun_best_setting: bool

tune_on_testset (*testset*, **args*, ***kwargs*)
Tunes the hyperparameter on a whole testset. :param testset: A list of testproblems. :type testset: list

10.2 Random Search

class deepobs.tuner.**RandomSearch** (*optimizer_class*, *hyperparam_names*, *distributions*, *ressources*, *runner*)
Bases: *deepobs.tuner.tuner.ParallelizedTuner*

A basic Random Search tuner.

__init__ (*optimizer_class*, *hyperparam_names*, *distributions*, *ressources*, *runner*)

Parameters **distributions** (*dict*) – Holds the distributions for each hyperparameter. Each distribution must implement an rvs() method to draw random variates. For instance, all scipy.stats.distribution distributions are applicable.

generate_commands_script (*testproblem*, *run_script*, *output_dir='./results'*, *random_seed=42*, *generation_dir='./command_scripts'*, ***kwargs*)

Parameters

- **testproblem** (*str*) – Testproblem for which to generate commands.
- **run_script** (*str*) – Name the run script that is used from the command line.
- **output_dir** (*str*) – The output path where the execution results are written to.
- **random_seed** (*int*) – The random seed for the tuning.
- **generation_dir** (*str*) – The path to the directory where the generated scripts are written to.

Returns The relative file path to the generated commands script.

Return type str

generate_commands_script_for_testset (*testset*, **args*, ***kwargs*)
Generates command scripts for a whole testset. :param testset: A list of the testproblem strings. :type testset: list

tune (*testproblem*, *output_dir='./results'*, *random_seed=42*, *rerun_best_setting=False*, ***kwargs*)
Tunes the optimizer on the test problem. :param testproblem: The test problem to tune the optimizer on. :type testproblem: str :param output_dir: The output directory for the results. :type output_dir: str :param random_seed: Random seed for the whole truning process. Every individual run is seeded by it. :type random_seed: int :param rerun_best_setting: Whether to automatically rerun the best setting with 10 different seeds. :type rerun_best_setting: bool

tune_on_testset (*testset*, **args*, ***kwargs*)
Tunes the hyperparameter on a whole testset. :param testset: A list of testproblems. :type testset: list

10.3 Gaussian Process

class deepobs.tuner.**GP**(*optimizer_class*, *hyperparam_names*, *bounds*, *ressources*, *runner*, *transformations=None*)

Bases: *deepobs.tuner.tuner.Tuner*

A Bayesian optimization tuner that uses a Gaussian Process surrogate.

__init__(*optimizer_class*, *hyperparam_names*, *bounds*, *ressources*, *runner*, *transformations=None*)

Parameters

- **optimizer_class** (*framework optimizer class*) – The optimizer to tune.
- **hyperparam_names** (*dict*) – Nested dictionary that holds the name, type and default values of the hyperparameters
- **bounds** (*dict*) – A dict where the key is the hyperparameter name and the value is a tuple of its bounds.
- **ressources** (*int*) – The number of total evaluations of the tuning process.
- **transformations** (*dict*) – A dict where the key is the hyperparameter name and the value is a callable that returns the transformed hyperparameter.
- **runner** – The DeepOBS runner which is used for each evaluation.

tune(*testproblem*, *output_dir='./results'*, *random_seed=42*, *n_init_samples=5*, *tuning_summary=True*, *plotting_summary=True*, *kernel=None*, *acq_type='ucb'*, *acq_kappa=2.576*, *acq_xi=0.0*, *mode='final'*, *rerun_best_setting=False*, ***kwargs*)

Tunes the optimizer hyperparameters by evaluating a Gaussian process surrogate with an acquisition function. :param testproblem: The test problem to tune the optimizer on. :type testproblem: str :param output_dir: The output directory for the results. :type output_dir: str :param random_seed: Random seed for the whole truning process. Every individual run is seeded by it. :type random_seed: int :param n_init_samples: The number of random exploration samples in the beginning of the tuning process. :type n_init_samples: int :param tuning_summary: Whether to write an additional tuning summary. Can be used to get an overview over the tuning progress :type tuning_summary: bool :param plotting_summary: Whether to store additional objects that can be used to plot the posterior. :type plotting_summary: bool :param kernel: The kernel of the GP. :type kernel: Sklearn.gaussian_process.kernels.Kernel :param acq_type: The type of acquisition function to use. Muste be one of `ucb`, `ei`, `poi`. :type acq_type: str :param acq_kappa: Scaling parameter of the acquisition function. :type acq_kappa: float :param acq_xi: Scaling parameter of the acquisition function. :type acq_xi: float :param mode: The mode that is used to evaluate the cost. Must be one of `final` or `best`. :type mode: str :param rerun_best_setting: Whether to automatically rerun the best setting with 10 different seeds. :type rerun_best_setting: bool

tune_on_testset(*testset*, **args*, ***kwargs*)

Tunes the hyperparameter on a whole testset. :param testset: A list of testproblems. :type testset: list

10.4 Tuner

The base class for all tuning methods in DeepOBS.

class deepobs.tuner.tuner.**Tuner**(*optimizer_class*, *hyperparam_names*, *ressources*, *runner*)

The base class for all tuning methods in DeepOBS.

__init__(*optimizer_class*, *hyperparam_names*, *ressources*, *runner*)

Parameters

- **optimizer_class** (*framework optimizer class*) – The optimizer class of the optimizer that is run on the testproblems. For PyTorch this must be a subclass of torch.optim.Optimizer. For TensorFlow a subclass of tf.train.Optimizer.
- **hyperparam_names** (*dict*) – A nested dictionary that lists all hyperparameters of the optimizer, their type and their default values (if they have any) in the form: {'<name>': {'type': <type>, 'default': <default value>}}, e.g. for torch.optim.SGD with momentum: {'lr': {'type': float}, 'momentum': {'type': float, 'default': 0.99}, 'uses_nesterov': {'type': bool, 'default': False}}
- **ressources** (*int*) – The number of evaluations the tuner is allowed to perform on each testproblem.
- **runner** – The DeepOBS runner that the tuner uses for evaluation.

tune(*testproblem*, **args*, *output_dir='./results'*, *random_seed=42*, *rerun_best_setting=True*, ***kwargs*)

Tunes hyperparaneter of the optimizer_class on a testproblem. :param testproblem: Testproblem for which to generate commands. :type testproblem: str :param output_dir: The output path where the execution results are written to. :type output_dir: str :param random_seed: The random seed for the tuning. :type random_seed: int :param rerun_best_setting: Whether to rerun the best setting with 10 different seeds. :type rerun_best_setting: bool

tune_on_testset(*testset*, **args*, ***kwargs*)

Tunes the hyperparameter on a whole testset. :param testset: A list of testproblems. :type testset: list

10.5 Parallelized Tuner

class deepobs.tuner.tuner.**ParallelizedTuner**(*optimizer_class*, *hyperparam_names*, *ressources*, *runner*)

Bases: *deepobs.tuner.tuner.Tuner*

The base class for all tuning methods which are uninformed and parallelizable, like Grid Search and Random Search.

generate_commands_script(*testproblem*, *run_script*, *output_dir='./results'*, *random_seed=42*, *generation_dir='./command_scripts'*, ***kwargs*)

Parameters

- **testproblem** (*str*) – Testproblem for which to generate commands.
- **run_script** (*str*) – Name the run script that is used from the command line.
- **output_dir** (*str*) – The output path where the execution results are written to.
- **random_seed** (*int*) – The random seed for the tuning.
- **generation_dir** (*str*) – The path to the directory where the generated scripts are written to.

Returns The relative file path to the generated commands script.

Return type str

generate_commands_script_for_testset(*testset*, **args*, ***kwargs*)

Generates command scripts for a whole testset. :param testset: A list of the testproblem strings. :type testset: list

tune(*testproblem*, *output_dir='./results'*, *random_seed=42*, *rerun_best_setting=False*, ***kwargs*)

Tunes the optimizer on the test problem. :param testproblem: The test problem to tune the optimizer

on. :type testproblem: str :param output_dir: The output directory for the results. :type output_dir: str :param random_seed: Random seed for the whole truning process. Every individual run is seeded by it. :type random_seed: int :param rerun_best_setting: Whether to automatically rerun the best setting with 10 different seeds. :type rerun_best_setting: bool

tune_on_testset(*testset*, **args*, ***kwargs*)
Tunes the hyperparameter on a whole testset. :param testset: A list of testproblems. :type testset: list

10.6 Tuning Utilities

10.6.1 General Utilities

deepobs.tuner.tuner_utils.**rerun_setting**(*runner*, *optimizer_class*, *hyperparam_names*, *optimizer_path*, *seeds=array([43, 44, 45, 46, 47, 48, 49, 50, 51])*, *rank=1*, *mode='final'*, *metric='valid_accuracies'*)
Reruns a hyperparameter setting with several seeds after the tuning is finished. Defaults to rerun the best setting. :param runner: The runner which was used for the tuning. :type runner: framework runner.runner :param optimizer_class: The optimizer class that was tuned. :type optimizer_class: framework optimizer class :param hyperparam_names: A nested dictionary that holds the names, the types and the default values of the hyperparams. :type hyperparam_names: dict :param optimizer_path: The path to the optimizer to analyse the best setting on. :type optimizer_path: str :param seeds: The seeds that are used to rerun the setting. :type seeds: iterable :param rank: The ranking of the setting that is to rerun. :type rank: int :param mode: The mode by which to decide the best setting. :type mode: str :param metric: The metric by which to decide the best setting. :type metric: str

deepobs.tuner.tuner_utils.**write_tuning_summary**(*optimizer_path*, *mode='final'*, *metric='valid_accuracies'*)
Writes the tuning summary to a json file in the `optimizer_path`. :param optimizer_path: Path to the optimizer folder. :type optimizer_path: str :param mode: The mode on which the performance measure for the summary is based. :type mode: str :param metric: The metric which is printed to the tuning summary as 'target' :type metric: str

deepobs.tuner.tuner_utils.**generate_tuning_summary**(*optimizer_path*, *mode='final'*, *metric='valid_accuracies'*)
Generates a list of dictionaries that holds an overview of the current tuning process. Should not be used for Bayesian tuning methods, since the order of evaluation is ignored in this summary. For Bayesian tuning methods use the tuning summary logging of the respective class.

Parameters

- **optimizer_path** (*str*) – Path to the optimizer folder.
- **mode** (*str*) – The mode on which the performance measure for the summary is based.
- **metric** (*str*) – The metric which is printed to the tuning summary as 'target'

Returns A list of dictionaries. Each dictionary corresponds to one hyperparameter evaluation of the tuning process and holds the hyperparameters and their performance. setting_analyzer_ranking (list): A ranked list of SettingAnalyzers that were used to generate the summary

Return type tuning_summary (list)

class deepobs.tuner.tuner_utils.**log_uniform**(*a*, *b*, *base=10*)
A log uniform distribution that takes an arbitrary base.

__init__(*a*, *b*, *base=10*)

Parameters

- **a** (*float*) – Lower bound.
- **b** (*float*) – Range from lower bound.
- **base** (*float*) – Base of the log.

10.6.2 Bayesian Specific Utilities

deepobs.tuner.bayesian_utils.**plot_1d_bo_posterior**(*optimizer_path*, *step*, *resolution*, *xscale='linear'*, *show=True*)

Plots the one dimensional GP posterior of the Bayesian tuning process. The tuning process must have been done for only one hyperparameter (i.e. one dimensional). :param optimizer_path: Path to the optimizer which was tuned. :type optimizer_path: str :param step: The step of the tuning process for which the posterior is plotted. :type step: int :param resolution: Resolution of the plot, i.e. number of x-values. :type resolution: int :param xscale: The scaling for the x-axis. :type xscale: str :param show: Whether to show the plot or not. :type show: bool

Returns The figure and axes of the plot.

Return type tuple

deepobs.tuner.bayesian_utils.**plot_2d_bo_posterior**(*optimizer_path*, *step*, *resolution*, *show=True*)

Plots the two dimensional GP posterior of the Bayesian tuning process. The tuning process must have been done for exactly two hyperparameters (i.e. two dimensional). :param optimizer_path: Path to the optimizer which was tuned. :type optimizer_path: str :param step: The step of the tuning process for which the posterior is plotted. :type step: int :param resolution: Resolution of the plot, i.e. number of x-values. :type resolution: int :param show: Whether to show the plot or not. :type show: bool

Returns Figure and axes of the plot.

Return type tuple

CHAPTER 11

Scripts

DeepOBS includes a few convenience scripts that can be run directly from the command line

- **Prepare Data**: Takes care of downloading and preprocessing all data sets for DeepOBS.
- **Get Baselines**: Automatically downloads the baselines of DeepOBS.
- **Plot Results**: Quickly plots the suggested outputs of a optimizer benchmark.

11.1 Prepare Data

A convenience script to download all data sets for DeepOBS and preprocess them so they are ready to be used with DeepOBS.

Note: Currently there is no data downloading and preprocessing mechanic implemented for *ImageNet*. Downloading the *ImageNet* data set requires an account and can take a lot of time to download. Additionally, it requires quite a large amount of memory. The best way currently is to download and preprocess the *ImageNet* data set separately if needed and move it into the DeepOBS data folder.

The file will create a set of folders of the following structure:

```
data_deepobs
├── cifar10
│   ├── data_batch_1.bin
│   ├── data_batch_2.bin
│   └── ...
├── cifar100
│   ├── train.bin
│   ├── test.bin
│   └── ...
```

```
├── fmnist
│   ├── t10k-images-idx3-ubyte.gz
│   ├── t10k-labels-idx1-ubyte.gz
│   └── ...
├── mnist
│   ├── t10k-images-idx3-ubyte.gz
│   ├── t10k-labels-idx1-ubyte.gz
│   └── ...
├── svhn
│   ├── data_batch_0.bin
│   ├── data_batch_1.bin
│   └── ...
├── tolstoi
│   ├── train.npy
│   ├── test.npy
│   └── ...
├── imagenet
│   ├── train-00000-of-01024
│   ├── ...
│   ├── validation-00000-of-00128
│   └── ...
```

DeepOBS expects a structure like this, so if you already have (most of the) the data sets already, you still need to bring it into this order.

Usage:

```
usage: deepobs_prepare_data.sh [--data_dir=DIR] [--skip SKIP] [--only ONLY]
```

11.1.1 Named Arguments

-d – data_dir	Path where the data sets should be saved. Defaults to the current folder.
-s – skip	Defines which data sets should be skipped. Argument needs to be one of the following `mnist`, `fmnist`, `cifar10`, `cifar100`, `svhn`, `imagenet`, `tolstoi`. You can use the `--skip` argument multiple times.
-o – only	Specify if only a single data set should be downloaded. Argument needs to be one of the following `mnist`, `fmnist`, `cifar10`, `cifar100`, `svhn`, `imagenet`, `tolstoi`. This overwrites the `--skip` argument and should can only be used once.

11.2 Download Baselines

A convenience script to download all baselines for DeepOBS.

Note: The download is currently around `470` MB large, so it might take a while, depending on your internet connec-

tion.

The baselines are currently for the three most popular deep learning optimizers, `SGD`, `Momentum` and `Adam`. The files include the `JSON` results for both the hyperparameter tuning phase (`36` runs with different learning rates) as well as the final results with the best performing setting (`10` runs with different random seeds and the same hyperparameter setting).

They can be used together with the plotting module or script to automatically compare the results of new optimizers, without having to run those baselines again.

Usage:

```
usage: deepobs_get_baselines.sh [--data_dir=DIR]
```

11.2.1 Named Arguments

-d –data_dir	Path where the baselines should be saved. Defaults to "baselines_deepobs".

11.3 Plot Results

A convenience script to extract useful information out of the results create by the runners.

This script can return one or all of the below information:

- Get best run: Returns the best hyperparameter setting for each optimizer in each test problem.
- Plot learning rate sensitivity: Creates a plot for each test problem showing the relative performance of each optimizer against the learning rate to get a sense of how difficult the tuning process was.
- Plot performance: Creates a plot for the `small` and `large` benchmark set, plotting (if available) all four performance metric (`losses` and `accuracies` for both the test and the train data set) for each optimizer.
- Plot table: Creates the overall performance table for the `small` and `large` benchmark set including metrics for the performance, speed and tuneability of each optimizer on each test problem.

If the path to the baseline folder is given, this script will also plot the performances of *SGD*, *Momentum* and *Adam*.

Usage:

Plotting tool for DeepOBS.

```
usage: deepobs_plot_results.py [-h] [--get_best_run] [--plot_lr_sensitivity]
                               [--plot_performance] [--plot_table] [--full]
                               [--baseline_path BASELINE_PATH]
                               path
```

11.3.1 Positional Arguments

path Path to the results folder

11.3.2 Named Arguments

--get_best_run Return best hyperparameter setting per optimizer and testproblem.

Default: False

--plot_lr_sensitivity Plot 'sensitivity' plot for the learning rates.

Default: False

--plot_performance Plot performance plot compared to the baselines.

Default: False

--plot_table Plot overall performance table including speed and hyperparameters.

Default: False

--full Run a full analysis and plot all figures.

Default: False

--baseline_path Path to baseline folder.

Default: "baselines_deepobs"

CHAPTER 12

Config

The global DeepOBS config.

deepobs.config.**get_testproblem_default_setting**(*testproblem*)

Returns default settings for the batch_size and the num_epochs for testproblem (if available).

Parameters **testproblem** (*str*) – Test problem for which to return the default setting.

Returns A dictionary with the default values for batch_size and num_epochs

Return type dict

deepobs.config.**set_framework**(*framework*)

Sets the current used framework. This is relevant for the higher level Tuner module of DeepOBS.

Parameters **framework** (*str*) – Can be 'pytorch' or 'tensorflow'

deepobs.config.**set_data_dir**(*data_dir*)

Sets the data directory.

Parameters **data_dir** (*str*) – Path to the data folder.

CHAPTER 13

Indices and tables

- genindex
- search

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

Q

R

S

T

V

W