Hebel Documentation

Release 0.02

Hannes Bretschneider

August 06, 2015

Contents

Contents:

CHAPTER 1

Getting Started

There are two basic methods how you can run Hebel:

1. You can write a YAML configuration file that describes your model architecture, data set, and hyperparameters and run it using the `train_model.py` script.
2. In your own Python script or program, you can create instances of models and optimizers programmatically.

The first makes estimating a model the easiest, as you don't have to write any actual code. You simply specify all your parameters and data set in an easy to read YAML configuration file and pass it to the `train_model.py` script. The script will create a directory for your results where it will save intermediary models (in pickle-format), the logs and final results.

The second method gives you more control over how exactly the model is estimated and lets you interact with Hebel from other Python programs.

1.1 Running models from YAML configuration files

If you check the example YAML files in `examples/` you will see that the configuration file defines three top-level sections:

1. `run_conf`: These options are passed to the method *`hebel.optimizers.SGD.run()`*.
2. `optimizer`: Here you instantiate a *`hebel.optimizers.SGD`* object, including the model you want to train and the data to use for training and validation.
3. `test_dataset`: This section is optional, but here you can define test data to evaluate the model on after training.

Check out `examples/mnist_neural_net_shallow.yml`, which includes everything to train a one layer neural network on the MNIST dataset:

```
run_conf:
  iterations: 50
optimizer: !obj:hebel.optimizers.SGD {
  model: !obj:hebel.models.NeuralNet {
    layers: [
      !obj:hebel.layers.HiddenLayer {
        n_in: 784,
        n_units: 2000,
        dropout: yes,
        l2_penalty_weight: .0
      }
    ],
```

```
    top_layer: !obj:hebel.layers.SoftmaxLayer {
      n_in: 2000,
      n_out: 10
    }
  },
  parameter_updater: !import hebel.parameter_updaters.MomentumUpdate,
  train_data: !obj:hebel.data_providers.MNISTDataProvider {
    batch_size: 100,
    array: train
  },
  validation_data: !obj:hebel.data_providers.MNISTDataProvider {
    array: val
  },
  learning_rate_schedule: !obj:hebel.schedulers.exponential_scheduler {
    init_value: 30., decay: .995
  },
  momentum_schedule: !obj:hebel.schedulers.linear_scheduler_up {
    init_value: .5, target_value: .9, duration: 10
  },
  progress_monitor:
    !obj:hebel.monitors.ProgressMonitor {
      experiment_name: mnist_shallow,
      save_model_path: examples/mnist,
      output_to_log: yes
    }
}
test_dataset:
  test_data: !obj:hebel.data_providers.MNISTDataProvider {
    array: test
  }
```

You can see that the only option we pass to `run_conf` is the number of iterations to train the model.

The `optimizer` section is more interesting. Hebel uses the special `!obj`, `!import`, and `!pkl` directives from PyLearn 2. The `!obj` directive is used most extensively and can be used to instantiate any Python class. First the optimizer *`hebel.optimizers.SGD`* is instantiated and in the lines below we are instantiating the model:

```
optimizer: !obj:hebel.optimizers.SGD {
  model: !obj:hebel.models.NeuralNet {
    layers: [
      !obj:hebel.layers.HiddenLayer {
        n_in: 784,
        n_units: 2000,
        dropout: yes,
        l2_penalty_weight: .0
      }
    ],
    top_layer: !obj:hebel.layers.SoftmaxLayer {
      n_in: 2000,
      n_out: 10
    }
  },
```

We are designing a model with one hidden layer that has 784 input units (the dimensionality of the MNIST data) and 2000 hidden units. We are also using dropout for regularization. The logistic output layer uses 10 classes (the number of classes in the MNIST data). You can also add different amounts of L1 or L2 penalization to each layer, which we are not doing here. Next, we define a `parameter_updater`, which is a rule that defines how the weights are updated given the gradients:

```
  parameter_updater: !import hebel.parameter_updaters.MomentumUpdate,
```

There are currently three choices:

- ***hebel.parameter_updaters.SimpleSGDUpdate*, which performs** regular gradient descent
- ***hebel.parameter_updaters.MomentumUpdate*, which performs** gradient descent with momentum, and
- ***hebel.parameter_updaters.NesterovMomentumUpdate*, which performs** gradient descent with Nesterov momentum.

The next two sections define the data for the model. All data must be given as instances of `DataProvider` objects:

```
  train_data: !obj:hebel.data_providers.MNISTDataProvider {
    batch_size: 100,
    array: train
  },
  validation_data: !obj:hebel.data_providers.MNISTDataProvider {
    array: val
  },
```

A `DataProvider` is a class that defines an iterator which returns successive minibatches of the data as well as saves some metadata, such as the number of data points. There is a special *hebel.data_providers.MNISTDataProvider* especially for the MNIST data. We use the standard splits for training and validation data here. There are several `DataProviders` defined in *hebel.data_providers*.

The next few lines define how some of the hyperparameters are changed over the course of the training:

```
  learning_rate_schedule: !obj:hebel.schedulers.exponential_scheduler {
    init_value: 30., decay: .995
  },
  momentum_schedule: !obj:hebel.schedulers.linear_scheduler_up {
    init_value: .5, target_value: .9, duration: 10
  },
```

The module `hebel.schedulers` defines several schedulers, which are basically just simple rules how certain parameters should evolve. Here, we define that the learning rate should decay exponentially with a factor of 0.995 in every epoch and the momentum should increase from 0.5 to 0.9 during the first 10 epochs and then stay at this value.

The last entry argument to *hebel.optimizers.SGD* is `progress_monitor`:

```
  progress_monitor:
    !obj:hebel.monitors.ProgressMonitor {
      experiment_name: mnist_shallow,
      save_model_path: examples/mnist,
      output_to_log: yes
    }
}
```

A progress monitor is an object that takes care of reporting periodic progress of our model, saving snapshots of the model at regular intervals, etc. When you are using the YAML configuration system, you'll probably want to use *hebel.monitors.ProgressMonitor*, which will save logs, outputs, and snapshots to disk. In contrast, *hebel.monitors.SimpleProgressMonitor* will only print progress to the terminal without saving the model itself.

Finally, you can define a test data set to be evaluated after the training completes:

```
  test_data: !obj:hebel.data_providers.MNISTDataProvider {
    array: test
  }
```

Here, we are specifying the MNIST test split.

Once you have your configuration file defined, you can run it such as in:

```
python train_model.py examples/mnist_neural_net_shallow.yml
```

The script will create the output directory you specified in `save_model_path` if it doesn't exist yet and start writing the log into a file called `output_log`. If you are interested in keeping an eye on the training process you can check on that file with:

```
tail -f output_log
```

1.2 Using Hebel in Your Own Code

If you want more control over the training procedure or integrate Hebel with your own code, then you can use Hebel programmatically.

Unlike the simpler one hidden layer model from the previous part, here we are going to build a more powerful deep neural net with multiple hidden layers.

For an example, have a look at `examples/mnist_neural_net_deep_script.py`:

```
#!/usr/bin/env python

import hebel
from hebel.models import NeuralNet
from hebel.optimizers import SGD
from hebel.parameter_updaters import MomentumUpdate
from hebel.data_providers import MNISTDataProvider
from hebel.monitors import ProgressMonitor
from hebel.schedulers import exponential_scheduler, linear_scheduler_up

hebel.init(random_seed=0)

# Initialize data providers
train_data = MNISTDataProvider('train', batch_size=100)
validation_data = MNISTDataProvider('val')
test_data = MNISTDataProvider('test')

D = train_data.D                        # Dimensionality of inputs
K = 10                                  # Number of classes

# Create model object
model = NeuralNet(n_in=train_data.D, n_out=K,
                  layers=[2000, 2000, 2000, 500],
                  activation_function='relu',
                  dropout=True, input_dropout=0.2)

# Create optimizer object
progress_monitor = ProgressMonitor(
    experiment_name='mnist',
    save_model_path='examples/mnist',
    save_interval=5,
    output_to_log=True)

optimizer = SGD(model, MomentumUpdate, train_data, validation_data, progress_monitor,
                learning_rate_schedule=exponential_scheduler(5., .995),
                momentum_schedule=linear_scheduler_up(.1, .9, 100))
```

```
# Run model
optimizer.run(50)

# Evaulate error on test set
test_error = model.test_error(test_data)
print "Error on test set: %.3f" % test_error
```

There are three basic tasks you have to do to train a model in Hebel:

1. Define the data you want to use for training, validation, or testing using `DataProvider` objects,
2. instantiate a `Model` object, and
3. instantiate an `SGD` object that will train the model using stochastic gradient descent.

1.2.1 Defining a Data Set

In this example we're using the MNIST data set again through the *hebel.data_providers.MNISTDataProvider* class:

```
from hebel.schedulers import exponential_scheduler, linear_scheduler_up

hebel.init(random_seed=0)
```

We create three data sets, corresponding to the official training, validation, and test data splits of MNIST. For the training data set, we set a batch size of 100 training examples, while the validation and test data sets are used as complete batches.

1.2.2 Instantiating a model

To train a model, you simply need to create an object representing a model that inherits from the abstract base class *hebel.models.Model*.

```
D = train_data.D                        # Dimensionality of inputs
K = 10                                  # Number of classes

# Create model object
```

Currently, Hebel implements the following models:

- *hebel.models.NeuralNet*: A neural net with any number of hidden layers for classification, using the cross-entropy loss function and softmax units in the output layer.
- *hebel.models.LogisticRegression*: Multi-class logistic regression. Like *hebel.models.NeuralNet* but does not have any hidden layers.
- *hebel.models.MultitaskNeuralNet*: A neural net trained on multiple tasks simultaneously. A multi-task neural net can have any number of hidden layers with weights that are shared between the tasks and any number of output layers with separate weights for each task.
- *hebel.models.NeuralNetRegression*: A neural net with a linear regression output layer to model continuous variables.

The *hebel.models.NeuralNet* model we are using here takes as input the dimensionality of the data, the number of classes, the sizes of the hidden layers, the activation function to use, and whether to use dropout for regularization. There are also a few more options such as for L1 or L2 weight regularization, that we don't use here.

Here, we are using the simpler form of the constructor rather than the extended form that we used in the YAML example. Also we are adding a small amount of dropout (20%) to the input layer.

1.2.3 Training the model

To train the model, you first need to create an instance of *hebel.optimizers.SGD*:

```
                  layers=[2000, 2000, 2000, 500],
                  activation_function='relu',
                  dropout=True, input_dropout=0.2)

# Create optimizer object
progress_monitor = ProgressMonitor(
    experiment_name='mnist',
    save_model_path='examples/mnist',
    save_interval=5,
    output_to_log=True)

optimizer = SGD(model, MomentumUpdate, train_data, validation_data, progress_monitor,
                learning_rate_schedule=exponential_scheduler(5., .995),
```

First we are creating a *hebel.monitors.ProgressMonitor* object, that will save regular snapshots of the model during training and save the logs and results to disk.

Next, we are creating the *hebel.optimizers.SGD* object. We instantiate the optimizer with the model, the parameter update rule, training data, validation data, and the schedulers for the learning rate and the momentum parameters.

Finally, we can start the training by invoking the *hebel.optimizers.SGD.run()* method. Here we train the model for 100 epochs. However, by default *hebel.optimizers.SGD* uses early stopping which means that it remembers the parameters that give the best result on the validation set and will reset the model parameters to them after the end of training.

1.2.4 Evaluating on test data

After training is complete we can do anything we want with the trained model, such as using it in some prediction pipeline, pickle it to disk, etc. Here we are evaluating the performance of the model on the MNIST test data split:

```
# Run model
optimizer.run(50)
```

CHAPTER 2

Initialization

Before Hebel can be used, it must be initialized using the function *hebel.init()*.

hebel.**init**(*device_id=None*, *random_seed=None*)

Initialize Hebel.

This function creates a CUDA context, CUBLAS context and initializes and seeds the pseudo-random number generator.

Parameters:

device_id [integer, optional] The ID of the GPU device to use. If this is omitted, PyCUDA's default context is used, which by default uses the fastest available device on the system. Alternatively, you can put the device id in the environment variable `CUDA_DEVICE` or into the file `.cuda-device` in the user's home directory.

random_seed [integer, optional] The seed to use for the pseudo-random number generator. If this is omitted, the seed is taken from the environment variable `RANDOM_SEED` and if that is not defined, a random integer is used as a seed.

CHAPTER 3

Data Providers

All data consumed by Hebel models must be provided in the form of `DataProvider` objects. `DataProviders` are classes that provide iterators which return batches for training. By writing custom `DataProviders`', this creates a lot of flexibility about where data can come from and enables any sort of pre-processing on the data. For example, a user could write a `DataProvider` that receives data from the internet or through a pipe from a different process. Or, when working with text data, a user may define a custom `DataProvider` to perform tokenization and stemming on the text before returning it.

A `DataProvider` is defined by subclassing the `hebel.data_provider.DataProvider` class and must implement at a minimum the special methods `__iter__` and `next`.

3.1 Abstract Base Class

class `hebel.data_providers.`**`DataProvider`**(*data*, *targets*, *batch_size*)

This is the abstract base class for `DataProvider` objects. Subclass this class to implement a custom design. At a minimum you must provide implementations of the `next` method.

3.2 Minibatch Data Provider

class `hebel.data_providers.`**`MiniBatchDataProvider`**(*data*, *targets*, *batch_size*)

This is the standard `DataProvider` for mini-batch learning with stochastic gradient descent.

Input and target data may either be provided as `numpy.array` objects, or as `pycuda.GPUArray` objects. The latter is preferred if the data can fit on GPU memory and will be much faster, as the data won't have to be transferred to the GPU for every minibatch. If the data is provided as a `numpy.array`, then every minibatch is automatically converted to to a `pycuda.GPUArray` and transferred to the GPU.

Parameters

- **`data`** – Input data.
- **`targets`** – Target data.
- **`batch_size`** – The size of mini-batches.

3.3 Multi-Task Data Provider

class hebel.data_providers.**MultiTaskDataProvider**(*data*, *targets*, *batch_size=None*)
DataProvider for multi-task learning that uses the same training data for multiple targets.

This DataProvider is similar to the hebel.data_provider.MiniBatchDataProvider, except that it has not one but multiple targets.

Parameters

- **data** – Input data.
- **targets** – Multiple targets as a list or tuple.
- **batch_size** – The size of mini-batches.

See also:

hebel.models.MultitaskNeuralNet, *hebel.layers.MultitaskTopLayer*

3.4 Batch Data Provider

class hebel.data_providers.**BatchDataProvider**(*data*, *targets*)
DataProvider for batch learning. Always returns the full data set.

Parameters

- **data** – Input data.
- **targets** – Target data.

See also:

hebel.data_providers.MiniBatchDataProvider

3.5 Dummy Data Provider

class hebel.data_providers.**DummyDataProvider**(**args*, ***kwargs*)
A dummy DataProvider that does not store any data and always returns None.

3.6 MNIST Data Provider

class hebel.data_providers.**MNISTDataProvider**(*array*, *batch_size=None*)
DataProvider that automatically provides data from the MNIST data set of hand-written digits.

Depends on the skdata package.

Parameters

- **array** – {'train', 'val', 'test'} Whether to use the official training, validation, or test data split of MNIST.
- **batch_size** – The size of mini-batches.

CHAPTER 4

Layers

4.1 Hidden Layer

class hebel.layers.**HiddenLayer**(*n_in*, *n_units*, *activation_function='sigmoid'*, *dropout=False*, *parameters=None*, *weights_scale=None*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, *lr_multiplier=None*)

A fully connected hidden layer.

The `HiddenLayer` class represents a fully connected hidden layer that can use a multitude of activation functions and supports dropout, L1, and L2 regularization.

Parameters:

n_in [integer] Number of input units.

n_out [integer] Number of hidden units.

activation_function [{`sigmoid`, `tanh`, `relu`, `linear`}, optional] Which activation function to use. Default is sigmoid.

dropout [bool] Whether the layer should use dropout (with dropout probability 0.5)

parameters [array_like of `GPUArray`] Parameters used to initialize the layer. If this is omitted, then the weights are initalized randomly using *Bengio's rule* (uniform distribution with scale $4 \cdot \sqrt{6/(\texttt{n\_in} + \texttt{n\_out})}$ if using sigmoid activations and $\sqrt{6/(\texttt{n\_in} + \texttt{n\_out})}$ if using tanh, relu, or linear activations) and the biases are initialized to zero. If `parameters` is given, then is must be in the form `[weights, biases]`, where the shape of weights is `(n_in, n_out)` and the shape of `biases` is `(n_out,)`. Both weights and biases must be `GPUArray`.

weights_scale [float, optional] If `parameters` is omitted, then this factor is used as scale for initializing the weights instead of *Bengio's rule*.

l1_penalty_weight [float, optional] Weight used for L1 regularization of the weights.

l2_penalty_weight [float, optional] Weight used for L2 regularization of the weights.

lr_multiplier [float, optional] If this parameter is omitted, then the learning rate for the layer is scaled by $2/\sqrt{\texttt{n\_in}}$. You may specify a different factor here.

Examples:

```
# Use the simple initializer and initialize with random weights
hidden_layer = HiddenLayer(500, 10000)

# Sample weights yourself, specify an L1 penalty, and don't
# use learning rate scaling
```

```
import numpy as np
from pycuda import gpuarray

n_in = 500
n_out = 1000
weights = gpuarray.to_gpu(.01 * np.random.randn(n_in, n_out))
biases = gpuarray.to_gpu(np.zeros((n_out,)))
hidden_layer = HiddenLayer(n_in, n_out,
                           parameters=(weights, biases),
                           l1_penalty_weight=.1,
                           lr_multiplier=1.)
```

architecture
Returns a dictionary describing the architecture of the layer.

backprop(*input_data*, *df_output*, *cache=None*)
Backpropagate through the hidden layer

Parameters:

input_data [GPUArray] Inpute data to compute activations for.

df_output [GPUArray] Gradients with respect to the activations of this layer (received from the layer above).

cache [list of GPUArray] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

Returns:

gradients [tuple of GPUArray] Gradients with respect to the weights and biases in the form `(df_weights, df_biases)`.

df_input [GPUArray] Gradients with respect to the input.

feed_forward(*input_data*, *prediction=False*)
Propagate forward through the layer

Parameters:

input_data [GPUArray] Inpute data to compute activations for.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the layers uses dropout.

Returns:

activations [GPUArray] The activations of the hidden units.

parameters
Return a tuple `(weights, biases)`

class hebel.layers.**InputDropout**(*n_in*, *dropout_probability=0.2*, *compute_input_gradients=False*)
This layer performs dropout on the input data.

It does not have any learnable parameters of its own. It should be used as the first layer and will perform dropout with any dropout probability on the incoming data.

Parameters:

n_in [integer] Number of input units.

dropout_probability [float in [0, 1]] Probability of dropping out each unit.

compute_input_gradients [Bool] Whether to compute the gradients with respect to the input data. This only necessary if you're training a model where the input itself is learned.

backprop(*input_data*, *df_output*, *cache=None*)
Backpropagate through the hidden layer

Parameters:

input_data [GPUArray] Inpute data to perform dropout on.

df_output [GPUArray] Gradients with respect to the output of this layer (received from the layer above).

cache [list of GPUArray] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

Returns:

gradients [empty tuple] Gradients are empty since this layer has no parameters.

df_input [GPUArray] Gradients with respect to the input.

feed_forward(*input_data*, *prediction=False*)
Propagate forward through the layer

Parameters:

input_data [GPUArray] Inpute data to perform dropout on.

prediction [bool, optional] Whether to use prediction model. If true, then the data is scaled by `1 - dropout_probability` uses dropout.

Returns:

dropout_data [GPUArray] The data after performing dropout.

class hebel.layers.**DummyLayer**(*n_in*)
This class has no hidden units and simply passes through its input

4.2 Top Layers

4.2.1 Abstract Base Class Top Layer

class hebel.layers.**TopLayer**(*n_in*, *n_units*, *activation_function='sigmoid'*, *dropout=False*, *parameters=None*, *weights_scale=None*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, *lr_multiplier=None*)
Abstract base class for a top-level layer.

4.2.2 Logistic Layer

class hebel.layers.**LogisticLayer**(*n_in*, *parameters=None*, *weights_scale=None*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, *lr_multiplier=None*, *test_error_fct='class_error'*)
A logistic classification layer for two classes, using cross-entropy loss function and sigmoid activations.

Parameters:

n_in [integer] Number of input units.

parameters [array_like of `GPUArray`] Parameters used to initialize the layer. If this is omitted, then the weights are initalized randomly using *Bengio's rule* (uniform distribution with scale $4 \cdot \sqrt{6/(\texttt{n\_in} + \texttt{n\_out})}$) and the biases are initialized to zero. If `parameters` is given, then is must be in the form `[weights, biases]`, where the shape of weights is `(n_in, n_out)` and the shape of `biases` is `(n_out,)`. Both weights and biases must be `GPUArray`.

weights_scale [float, optional] If `parameters` is omitted, then this factor is used as scale for initializing the weights instead of *Bengio's rule*.

l1_penalty_weight [float, optional] Weight used for L1 regularization of the weights.

l2_penalty_weight [float, optional] Weight used for L2 regularization of the weights.

lr_multiplier [float, optional] If this parameter is omitted, then the learning rate for the layer is scaled by $2/\sqrt{\texttt{n\_in}}$. You may specify a different factor here.

test_error_fct [{`class_error`, `kl_error`, `cross_entropy_error`}, optional] Which error function to use on the test set. Default is `class_error` for classification error. Other choices are `kl_error`, the Kullback-Leibler divergence, or `cross_entropy_error`.

See also:

hebel.layers.SoftmaxLayer, *hebel.models.NeuralNet*, *hebel.models.NeuralNetRegression*, *hebel.layers.LinearRegressionLayer*

Examples:

```
# Use the simple initializer and initialize with random weights
logistic_layer = LogisticLayer(1000)

# Sample weights yourself, specify an L1 penalty, and don't
# use learning rate scaling
import numpy as np
from pycuda import gpuarray

n_in = 1000
weights = gpuarray.to_gpu(.01 * np.random.randn(n_in, 1))
biases = gpuarray.to_gpu(np.zeros((1,)))
softmax_layer = SoftmaxLayer(n_in,
                             parameters=(weights, biases),
                             l1_penalty_weight=.1,
                             lr_multiplier=1.)
```

backprop(*input_data*, *targets*, *cache=None*)

Backpropagate through the logistic layer.

Parameters:

input_data [`GPUArray`] Inpute data to compute activations for.

targets [`GPUArray`] The target values of the units.

cache [list of `GPUArray`] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

Returns:

gradients [tuple of `GPUArray`] Gradients with respect to the weights and biases in the form `(df_weights, df_biases)`.

df_input [`GPUArray`] Gradients with respect to the input.

class_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*)

Return the classification error rate

cross_entropy_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*)
Return the cross entropy error

feed_forward(*input_data*, *prediction=False*)
Propagate forward through the layer.

Parameters:

input_data [GPUArray] Inpute data to compute activations for.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the layers uses dropout.

Returns:

activations [GPUArray] The activations of the output units.

test_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=True*)
Compute the test error function given some data and targets.

Uses the error function defined in `SoftmaxLayer.test_error_fct`, which may be different from the cross-entropy error function used for training'. Alternatively, the other test error functions may be called directly.

Parameters:

input_data [GPUArray] Inpute data to compute the test error function for.

targets [GPUArray] The target values of the units.

average [bool] Whether to divide the value of the error function by the number of data points given.

cache [list of GPUArray] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the layers uses dropout.

Returns: test_error : float

train_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*)
Return the cross entropy error

4.2.3 Softmax Layer

class hebel.layers.**SoftmaxLayer**(*n_in*, *n_out*, *parameters=None*, *weights_scale=None*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, *lr_multiplier=None*, *test_error_fct='class_error'*)
A multiclass classification layer, using cross-entropy loss function and softmax activations.

Parameters:

n_in [integer] Number of input units.

n_out [integer] Number of output units (classes).

parameters [array_like of GPUArray] Parameters used to initialize the layer. If this is omitted, then the weights are initalized randomly using *Bengio's rule* (uniform distribution with scale $4 \cdot \sqrt{6/(\texttt{n\_in} + \texttt{n\_out})}$) and the biases are initialized to zero. If `parameters` is given, then is must be in the form `[weights, biases]`, where the shape of weights is `(n_in, n_out)` and the shape of `biases` is `(n_out,)`. Both weights and biases must be GPUArray.

weights_scale [float, optional] If `parameters` is omitted, then this factor is used as scale for initializing the weights instead of *Bengio's rule*.

l1_penalty_weight [float, optional] Weight used for L1 regularization of the weights.

l2_penalty_weight [float, optional] Weight used for L2 regularization of the weights.

lr_multiplier [float, optional] If this parameter is omitted, then the learning rate for the layer is scaled by $2/\sqrt{\texttt{n\_in}}$. You may specify a different factor here.

test_error_fct [{`class_error`, `kl_error`, `cross_entropy_error`}, optional] Which error function to use on the test set. Default is `class_error` for classification error. Other choices are `kl_error`, the Kullback-Leibler divergence, or `cross_entropy_error`.

See also:

hebel.layers.LogisticLayer, *hebel.models.NeuralNet*, *hebel.models.NeuralNetRegression*, *hebel.layers.LinearRegressionLayer*

Examples:

```
# Use the simple initializer and initialize with random weights
softmax_layer = SoftmaxLayer(1000, 10)

# Sample weights yourself, specify an L1 penalty, and don't
# use learning rate scaling
import numpy as np
from pycuda import gpuarray

n_in = 1000
n_out = 10
weights = gpuarray.to_gpu(.01 * np.random.randn(n_in, n_out))
biases = gpuarray.to_gpu(np.zeros((n_out,)))
softmax_layer = SoftmaxLayer(n_in, n_out,
                             parameters=(weights, biases),
                             l1_penalty_weight=.1,
                             lr_multiplier=1.)
```

backprop(*input_data*, *targets*, *cache=None*)
Backpropagate through the logistic layer.

Parameters:

input_data [`GPUArray`] Inpute data to compute activations for.

targets [`GPUArray`] The target values of the units.

cache [list of `GPUArray`] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

Returns:

gradients [tuple of `GPUArray`] Gradients with respect to the weights and biases in the form `(df_weights, df_biases)`.

df_input [`GPUArray`] Gradients with respect to the input.

class_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*)
Return the classification error rate

cross_entropy_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*)
Return the cross entropy error

feed_forward(*input_data*, *prediction=False*)
Propagate forward through the layer.

Parameters:

input_data [GPUArray] Inpute data to compute activations for.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the layers uses dropout.

Returns:

activations [GPUArray] The activations of the output units.

kl_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=True*)
The KL divergence error

test_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=True*)
Compute the test error function given some data and targets.

Uses the error function defined in `SoftmaxLayer.test_error_fct`, which may be different from the cross-entropy error function used for training'. Alternatively, the other test error functions may be called directly.

Parameters:

input_data [GPUArray] Inpute data to compute the test error function for.

targets [GPUArray] The target values of the units.

average [bool] Whether to divide the value of the error function by the number of data points given.

cache [list of GPUArray] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the layers uses dropout.

Returns: test_error : float

train_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*)
Return the cross entropy error

4.2.4 Linear Regression Layer

class hebel.layers.**LinearRegressionLayer**(*n_in*, *n_out*, *parameters=None*, *weights_scale=None*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, *lr_multiplier=None*)
Linear regression layer with linear outputs and squared loss error function.

Parameters:

n_in [integer] Number of input units.

n_out [integer] Number of output units (classes).

parameters [array_like of GPUArray] Parameters used to initialize the layer. If this is omitted, then the weights are initalized randomly using *Bengio's rule* (uniform distribution with scale $4 \cdot \sqrt{6/(\texttt{n\_in} + \texttt{n\_out})}$) and the biases are initialized to zero. If `parameters` is given, then is must be in the form `[weights, biases]`, where the shape of weights is `(n_in, n_out)` and the shape of `biases` is `(n_out,)`. Both weights and biases must be GPUArray.

weights_scale [float, optional] If `parameters` is omitted, then this factor is used as scale for initializing the weights instead of *Bengio's rule*.

l1_penalty_weight [float, optional] Weight used for L1 regularization of the weights.

l2_penalty_weight [float, optional] Weight used for L2 regularization of the weights.

lr_multiplier [float, optional] If this parameter is omitted, then the learning rate for the layer is scaled by $2/\sqrt{\texttt{n\_in}}$. You may specify a different factor here.

test_error_fct [{`class_error`, `kl_error`, `cross_entropy_error`}, optional] Which error function to use on the test set. Default is `class_error` for classification error. Other choices are `kl_error`, the Kullback-Leibler divergence, or `cross_entropy_error`.

See also:

hebel.models.NeuralNetRegression, *hebel.models.NeuralNet*, *hebel.layers.LogisticLayer*

feed_forward(*input_data*, *prediction=False*)
Propagate forward through the layer.

Parameters:

input_data [`GPUArray`] Inpute data to compute activations for.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the layers uses dropout.

Returns:

activations [`GPUArray`] The activations of the output units.

test_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=True*)
Compute the test error function given some data and targets.

Uses the error function defined in `SoftmaxLayer.test_error_fct`, which may be different from the cross-entropy error function used for training'. Alternatively, the other test error functions may be called directly.

Parameters:

input_data [`GPUArray`] Inpute data to compute the test error function for.

targets [`GPUArray`] The target values of the units.

average [bool] Whether to divide the value of the error function by the number of data points given.

cache [list of `GPUArray`] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the layers uses dropout.

Returns: test_error : float

4.2.5 Multitask Top Layer

class hebel.layers.**MultitaskTopLayer**(*n_in=None*, *n_out=None*, *test_error_fct='class_error'*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, *tasks=None*, *task_weights=None*, *n_tasks=None*, *lr_multiplier=None*)
Top layer for performing multi-task training.

This is a top layer that enables multi-task training, which can be thought of as training multiple models on the same data and sharing weights in all but the final layer. A `MultitaskTopLayer` has multiple layers as children that are subclasses of *hebel.layers.TopLayer*. During the forward pass, the input from the previous layer is passed on to all tasks and during backpropagation, the gradients are added together from the different tasks (with different weights if necessary).

There are two ways of initializing `MultitaskTopLayer`:

1. By supplying `n_in`, `n_out`, and optionally `n_tasks`, which will initialize all tasks with *hebel.layers.LogisticLayer*. If `n_tasks` is given, `n_out` must be an integer and `n_tasks` identical tasks will be created. If `n_out` is an `array_like`, then as many tasks will be created as there are elements in `n_out` and `n_tasks` will be ignored.
2. If `tasks` is supplied, then it must be an `array_like` of objects derived from *hebel.layers.TopLayer*, one object for each class. In this case `n_in`, `n_out`, and `n_tasks` will be ignored. The user must make sure that all tasks have their `n_in` member variable set to the same value.

Parameters:

n_in [integer, optional] Number of input units. Is ignored, when `tasks` is supplied.

n_out [integer or array_like, optional] Number of output units. May be an integer (all tasks get the same number of units; `n_tasks` must be given), or `array_like` (create as many tasks as elements in `n_out` with different sizes; `n_tasks is ignored). Is always ignored when ``tasks` is supplied.

test_error_fct [string, optional] See *hebel.layers.LogisticLayer* for options. Ignored when `tasks` is supplied.

l1_penalty_weight [float or list/tuple of floats, optional] Weight(s) for L1 regularization. Ignored when `tasks` is supplied.

l2_penalty_weight [float or list/tuple of floats, optional] Weight(s)for L2 regularization. Ignored when `tasks` is supplied.

tasks [list/tuple of *hebel.layers.TopLayer* objects, optional] Tasks for multitask learning. Overrides `n_in`, `n_out`, `test_error_fct`, `l1_penalty_weight`, `l2_penalty_weight`, `n_tasks`, and `lr_multiplier`.

task_weights [list/tuple of floats, optional] Weights to use when adding the gradients from the different tasks. Default is `1./self.n_tasks`. The weights don't need to necessarily add up to one.

n_tasks [integer, optional] Number of tasks. Ignored if `n_out` is a list, or `tasks` is supplied.

lr_multiplier [float or list/tuple of floats] A task dependant multiplier for the learning rate. If this is ignored, then the tasks default is used. It is ignored when `tasks` is supplied.

See also: *hebel.layers.TopLayer*, *hebel.layers.LogisticLayer*

Examples:

```
    # Simple form of the constructor
    # Creating five tasks with same number of classes
    multitask_layer = MultitaskTopLayer(n_in=1000, n_out=10, n_tasks=5)

    # Extended form of the constructor
    # Initializing every task independently

    n_in = 1000               # n_in must be the same for all tasks
    tasks = (
        SoftmaxLayer(n_in, 10, l1_penalty_weight=.1),
        SoftmaxLayer(n_in, 15, l2_penalty_weight=.2),
        SoftmaxLayer(n_in, 10),
        SoftmaxLayer(n_in, 10),
        SoftmaxLayer(n_in, 20)
    )
    task_weights = [1./5, 1./10, 1./10, 2./5, 1./5]
    multitask_layer = MultitaskTopLayer(tasks=tasks,
                                        task_weights=task_weights)
```

architecture
Returns a dictionary describing the architecture of the layer.

backprop(*input_data*, *targets*, *cache=None*)
Compute gradients for each task and combine the results.

Parameters:

input_data [GPUArray] Inpute data to compute activations for.

targets [GPUArray] The target values of the units.

cache [list of GPUArray] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

Returns:

gradients [list] Gradients with respect to the weights and biases for each task

df_input [GPUArray] Gradients with respect to the input, obtained by adding the gradients with respect to the inputs from each task, weighted by MultitaskTopLayer.task_weights.

cross_entropy_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*, *sum_errors=True*)
Computes the cross-entropy error for all tasks.

feed_forward(*input_data*, *prediction=False*)
Call feed_forward for each task and combine the activations.

Passes input_data to all tasks and returns the activations as a list.

Parameters:

input_data [GPUArray] Inpute data to compute activations for.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the task uses dropout.

Returns:

activations [list of GPUArray] The activations of the output units, one element for each task.

l1_penalty
Compute the L1 penalty for all tasks.

l2_penalty
Compute the L2 penalty for all tasks.

parameters
Return a list where each element contains the parameters for a task.

test_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*, *sum_errors=True*)
Compute the error function on a test data set.

Parameters:

input_data [GPUArray] Inpute data to compute the test error function for.

targets [GPUArray] The target values of the units.

average [bool] Whether to divide the value of the error function by the number of data points given.

cache [list of GPUArray] Cache obtained from forward pass. If the cache is provided, then the activations are not recalculated.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved if the layers uses dropout.

sum_errors [bool, optional] Whether to add up the errors from the different tasks. If this option is chosen, the user must make sure that all tasks use the same test error function.

Returns:

test_error [float or list] Returns a float when `sum_errors == True` and a list with the individual errors otherwise.

train_error(*input_data*, *targets*, *average=True*, *cache=None*, *prediction=False*, *sum_errors=True*)
Computes the cross-entropy error for all tasks.

CHAPTER 5

Monitors

5.1 Progress Monitor

class hebel.monitors.**ProgressMonitor**(*experiment_name=None*, *save_model_path=None*, *save_interval=None*, *output_to_log=False*, *model=None*)

avg_weight()

finish_training()

makedir()

print_(*obj*)

print_error(*epoch*, *train_error*, *validation_error=None*, *new_best=None*)

report(*epoch*, *train_error*, *validation_error=None*, *new_best=None*, *epoch_t=None*)

start_training()

test_error

yaml_config

5.2 Simple Progress Monitor

class hebel.monitors.**SimpleProgressMonitor**(*model=None*)

avg_weight()

finish_training()

print_error(*epoch*, *train_error*, *validation_error=None*)

report(*epoch*, *train_error*, *validation_error=None*, *new_best=None*, *epoch_t=None*)

start_training()

CHAPTER 6

Models

6.1 Abstract Base Class Model

class hebel.models.**Model**
Abstract base-class for a Hebel model

evaluate(*input_data*, *targets*, *return_cache=False*, *prediction=True*)
Evaluate the loss function without computing gradients

feed_forward(*input_data*, *return_cache=False*, *prediction=True*)
Get predictions from the model

test_error(*input_data*, *targets*, *average=True*, *cache=None*)
Evaulate performance on a test set

training_pass(*input_data*, *targets*)
Perform a full forward and backward pass through the model

6.2 Neural Network

class hebel.models.**NeuralNet**(*layers*, *top_layer=None*, *activation_function='sigmoid'*, *dropout=False*, *input_dropout=0.0*, *n_in=None*, *n_out=None*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, ***kwargs*)
A neural network for classification using the cross-entropy loss function.

Parameters:

layers [array_like] An array of either integers or instances of `hebel.models.HiddenLayer` objects. If integers are given, they represent the number of hidden units in each layer and new `HiddenLayer` objects will be created. If `HiddenLayer` instances are given, the user must make sure that each `HiddenLayer` has `n_in` set to the preceding layer's `n_units`. If `HiddenLayer` instances are passed, then `activation_function`, `dropout`, `n_in`, `l1_penalty_weight`, and `l2_penalty_weight` are ignored.

top_layer [`hebel.models.TopLayer` instance, optional] If `top_layer` is given, then it is used for the output layer, otherwise, a `LogisticLayer` instance is created.

activation_function [{'sigmoid', 'tanh', 'relu', or 'linear'}, optional] The activation function to be used in the hidden layers.

dropout [bool, optional] Whether to use dropout regularization

input_dropout [float, in `[0, 1]`] Dropout probability for the input (default 0.0).

n_in [integer, optional] The dimensionality of the input. Must be given, if the first hidden layer is not passed as a `hebel.models.HiddenLayer` instance.

n_out [integer, optional] The number of classes to predict from. Must be given, if a `hebel.models.HiddenLayer` instance is not given in `top_layer`.

l1_penalty_weight [float, optional] Weight for L1 regularization

l2_penalty_weight [float, optional] Weight for L2 regularization

kwargs [optional] Any additional arguments are passed on to `top_layer`

See also:

hebel.models.LogisticRegression, *hebel.models.NeuralNetRegression*, *hebel.models.MultitaskNeuralNet*

Examples:

```
# Simple form
model = NeuralNet(layers=[1000, 1000],
                  activation_function='relu',
                  dropout=True,
                  n_in=784, n_out=10,
                  l1_penalty_weight=.1)

# Extended form, initializing with ``HiddenLayer`` and ``TopLayer`` objects
hidden_layers = [HiddenLayer(784, 1000, 'relu', dropout=True,
                             l1_penalty_weight=.2),
                 HiddenLayer(1000, 1000, 'relu', dropout=True,
                             l1_penalty_weight=.1)]
softmax_layer = LogisticLayer(1000, 10, l1_penalty_weight=.1)

model = NeuralNet(hidden_layers, softmax_layer)
```

TopLayerClass
alias of `SoftmaxLayer`

checksum()
Returns an MD5 digest of the model.

This can be used to easily identify whether two models have the same architecture.

evaluate(*input_data*, *targets*, *return_cache=False*, *prediction=True*)
Evaluate the loss function without computing gradients.

Parameters:

input_data [GPUArray] Data to evaluate

targets: GPUArray Targets

return_cache [bool, optional] Whether to return intermediary variables from the computation and the hidden activations.

prediction [bool, optional] Whether to use prediction model. Only relevant when using dropout. If true, then weights are halved in layers that use dropout.

Returns:

loss [float] The value of the loss function.

hidden_cache [list, only returned if `return_cache == True`] Cache as returned by *hebel.models.NeuralNet.feed_forward()*.

activations [list, only returned if `return_cache == True`] Hidden activations as returned by *hebel.models.NeuralNet.feed_forward()*.

feed_forward(*input_data*, *return_cache=False*, *prediction=True*)
Run data forward through the model.

Parameters:

input_data [GPUArray] Data to run through the model.

return_cache [bool, optional] Whether to return the intermediary results.

prediction [bool, optional] Whether to run in prediction mode. Only relevant when using dropout. If true, weights are halved. If false, then half of hidden units are randomly dropped and the dropout mask is returned in case `return_cache==True`.

Returns:

prediction [GPUArray] Predictions from the model.

cache [list of GPUArray, only returned if `return_cache == True`] Results of intermediary computations.

parameters
A property that returns all of the model's parameters.

test_error(*test_data*, *average=True*)
Evaulate performance on a test set.

Parameters:

test_data [:class:`hebel.data_provider.DataProvider`] A `DataProvider` instance to evaluate on the model.

average [bool, optional] Whether to divide the loss function by the number of examples in the test data set.

Returns:

test_error : float

training_pass(*input_data*, *targets*)
Perform a full forward and backward pass through the model.

Parameters:

input_data [GPUArray] Data to train the model with.

targets [GPUArray] Training targets.

Returns:

loss [float] Value of loss function as evaluated on the data and targets.

gradients [list of GPUArray] Gradients obtained from backpropagation in the backward pass.

6.3 Neural Network Regression

class hebel.models.**NeuralNetRegression**(*layers*, *top_layer=None*, *activation_function='sigmoid'*, *dropout=False*, *input_dropout=0.0*, *n_in=None*, *n_out=None*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, ***kwargs*)

A neural network for regression using the squared error loss function.

This class exists for convenience. The same results can be achieved by creating a *hebel.models.NeuralNet* instance and passing a *hebel.layers.LinearRegressionLayer* instance as the `top_layer` argument.

Parameters:

layers [array_like] An array of either integers or instances of `hebel.models.HiddenLayer` objects. If integers are given, they represent the number of hidden units in each layer and new `HiddenLayer` objects will be created. If `HiddenLayer` instances are given, the user must make sure that each `HiddenLayer` has `n_in` set to the preceding layer's `n_units`. If `HiddenLayer` instances are passed, then `activation_function`, `dropout`, `n_in`, `l1_penalty_weight`, and `l2_penalty_weight` are ignored.

top_layer [`hebel.models.TopLayer` instance, optional] If `top_layer` is given, then it is used for the output layer, otherwise, a `LinearRegressionLayer` instance is created.

activation_function [{'sigmoid', 'tanh', 'relu', or 'linear'}, optional] The activation function to be used in the hidden layers.

dropout [bool, optional] Whether to use dropout regularization

n_in [integer, optional] The dimensionality of the input. Must be given, if the first hidden layer is not passed as a `hebel.models.HiddenLayer` instance.

n_out [integer, optional] The number of classes to predict from. Must be given, if a `hebel.models.HiddenLayer` instance is not given in `top_layer`.

l1_penalty_weight [float, optional] Weight for L1 regularization

l2_penalty_weight [float, optional] Weight for L2 regularization

kwargs [optional] Any additional arguments are passed on to `top_layer`

See also:

hebel.models.NeuralNet, *hebel.models.MultitaskNeuralNet*, *hebel.layers.LinearRegressionLayer*

TopLayerClass
alias of `LinearRegressionLayer`

6.4 Logistic Regression

class hebel.models.**LogisticRegression**(*n_in*, *n_out*, *test_error_fct='class_error'*)

A logistic regression model

6.5 Multi-Task Neural Net

class hebel.models.**MultitaskNeuralNet**(*layers*, *top_layer=None*, *activation_function='sigmoid'*, *dropout=False*, *input_dropout=0.0*, *n_in=None*, *n_out=None*, *l1_penalty_weight=0.0*, *l2_penalty_weight=0.0*, ***kwargs*)

TopLayerClass
alias of MultitaskTopLayer

CHAPTER 7

Optimizers

7.1 Stochastic Gradient Descent

class hebel.optimizers.**SGD**(*model*, *parameter_updater*, *train_data*, *validation_data=None*, *progress_monitor=None*, *learning_rate_schedule=<generator object constant_scheduler>*, *momentum_schedule=None*, *early_stopping=True*)

best_validation_loss

norm_v_norm()

run(*iterations=200*, *validation_interval=5*, *yaml_config=None*, *task_id=None*)

CHAPTER 8

Parameter Updaters

8.1 Abstract Base Class

class hebel.parameter_updaters.**ParameterUpdater**(*model*)

post_gradient_update(*gradients*, *stream=None*)

pre_gradient_update(*stream=None*)

8.2 Simple SGD Update

class hebel.parameter_updaters.**SimpleSGDUpdate**(*model*)

post_gradient_update(*gradients*, *batch_size*, *learning_parameters*, *stream=None*)

8.3 Momentum Update

class hebel.parameter_updaters.**MomentumUpdate**(*model*)

post_gradient_update(*gradients*, *batch_size*, *learning_parameters*, *stream=None*)

8.4 Nesterov Momentum Update

class hebel.parameter_updaters.**NesterovMomentumUpdate**(*model*)

post_gradient_update(*gradients*, *batch_size*, *learning_parameters*, *stream=None*)
Second step of Nesterov momentum method: take step in direction of new gradient and update velocity

pre_gradient_update()
First step of Nesterov momentum method: take step in direction of accumulated gradient

CHAPTER 9

Schedulers

9.1 Constant Scheduler

hebel.schedulers.**constant_scheduler**(*value*)

9.2 Exponential Scheduler

hebel.schedulers.**exponential_scheduler**(*init_value*, *decay*)
Decreases exponentially

9.3 Linear Scheduler Up

hebel.schedulers.**linear_scheduler_up**(*init_value*, *target_value*, *duration*)
Increases linearly and then stays flat

9.4 Linear Scheduler Up-Down

hebel.schedulers.**linear_scheduler_up_down**(*init_value*, *target_value*, *final_value*, *duration_up*, *t_decrease*, *duration_down*)
Increases linearly to target_value, stays at target_value until t_decrease and then decreases linearly

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

h

A

B

C

D

E

F

H

I

K

L

M

N

P

R

S

T

Y