Hebel Documentation

Release 0.001

Hannes Bretschneider

January 01, 2014

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.LogisticLayer {
      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,
      save_interval: 10,
      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.LogisticLayer {
      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,
    save_interval: 10,
    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_dataset:
  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`:

```
import pycuda.autoinit
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

# 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,
                learning_rate_schedule=exponential_scheduler(5., .995),
                momentum_schedule=linear_scheduler_up(.1, .9, 100))

# Run model
optimizer.run(500)

# 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:

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

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`.

```
# 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)
```

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`:

```
# 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,
                learning_rate_schedule=exponential_scheduler(5., .995),
                momentum_schedule=linear_scheduler_up(.1, .9, 100))

# Run model
optimizer.run(500)
```

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:

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

CHAPTER 2

Data Providers

2.1 Abstract Base Class

2.2 Minibatch Data Provider

2.3 Multi-Task Data Provider

2.4 Batch Data Provider

2.5 Dummy Data Provider

2.6 MNIST Data Provider

CHAPTER 3

Layers

3.1 Hidden Layer

3.2 Top Layers

3.2.1 Abstract Base Class Top Layer

3.2.2 Logistic Layer

3.2.3 Linear Regression Layer

3.2.4 Multitask Top Layer

CHAPTER 4

Monitors

4.1 Progress Monitor

4.2 Simple Progress Monitor

CHAPTER 5

Models

5.1 Abstract Base Class Model

5.2 Neural Network

5.3 Neural Network Regression

5.4 Logistic Regression

5.5 Multi-Task Neural Net

CHAPTER 6

Optimizers

6.1 Stochastic Gradient Descent

CHAPTER 7

Parameter Updaters

7.1 Abstract Base Class

7.2 Simple SGD Update

7.3 Momentum Update

7.4 Nesterov Momentum Update

CHAPTER 8

Schedulers

8.1 Constant Scheduler

8.2 Exponential Scheduler

8.3 Linear Scheduler Up

8.4 Linear Scheduler Up-Down

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*