numpydl Documentation

Release 0.1.0

NumpyDL

May 11, 2017

Contents:

NumpyDL is a simple deep learning library based on pure Python/Numpy.

NumpyDL is a work in progress, input is welcome. The available documentation is limited for now. The project is on GitHub.

CHAPTER 1

User Guide

The NumpyDL user guide explains how to install NumpyDL, how to build and train neural networks using NumpyDL, and how to contribute to the library as a developer.

Installation

NumpyDL has a couple of prerequisites that need to be installed first, but it is not very picky about versions. The most important package is Numpy. At the same time, you should install some other useful packages, such as scipy and scikit-learn. Most importantly, these packages are not required to install the specific version to fit the version of NumpyDL you choose to install.

We strongly recommend you to install the Miniconda or a bigger installer Anaconda which is a leading open data science platform powered by Python and well integrated the efficient scientific computing platform MKL.

Prerequisites

Python + pip

NumpyDL currently requires Python 3.3 or higher to run. Please install Python via the package manager of your operating system if it is not included already.

Python includes `pip` for installing additional modules that are not shipped with your operating system, or shipped in an old version, and we will make use of it below. We recommend installing these modules into your home directory via `--user`, or into a virtual environment via `virtualenv`.

C compiler

Numpy/scipy require a C compiler if you install them via `pip`. On Linux, the default compiler is usually‘‘gcc‘‘, and on Mac OS, it’s `clang`. On Windows, we recommend you to install the Miniconda or Anaconda. Again, please install them via the package manager of your operating system.

numpy/scipy + BLAS

NumpyDL requires numpy of version 1.6.2 or above, and sometimes also requires scipy 0.11 or above. Numpy/scipy rely on a BLAS library to provide fast linear algebra routines. They will work fine without one, but a lot slower, so it is worth getting this right (but this is less important if you plan to use a GPU).

If you install numpy and scipy via your operating system's package manager, they should link to the BLAS library installed in your system. If you install numpy and scipy via `pip install numpy` and `pip install scipy`, make sure to have development headers for your BLAS library installed (e.g., the `libopenblas-dev` package on Debian/Ubuntu) while running the installation command. Please refer to the numpy/scipy build instructions if in doubt.

Stable NumpyDL release

NumpyDL 0.1 requires a more recent version of Theano than the one available on PyPI. To install a version that is known to work, run the following command:

```
pip install -r https://github.com/oujago/NumpyDL/blob/master/requirements.txt
```

```
pip install npdl
```

If you do not use `virtualenv`, add `--user` to both commands to install into your home directory instead. To upgrade from an earlier installation, add `--upgrade`.

Development installation

Alternatively, you can install NumpyDL from source, in a way that any changes to your local copy of the source tree take effect without requiring a reinstall. This is often referred to as *editable* or *development* mode. Firstly, you will need to obtain a copy of the source tree:

```
git clone https://github.com/oujago/NumpyDL.git
```

It will be cloned to a subdirectory called `NumpyDL`. Make sure to place it in some permanent location, as for an *editable* installation, Python will import the module directly from this directory and not copy over the files. Enter the directory and install the known good version of Theano:

```
cd NumpyDL
pip install -r requirements.txt
```

To install the NumpyDL package itself, in editable mode, run:

```
pip install --editable .
```

As always, add `--user` to install it to your home directory instead.

Optional: If you plan to contribute to NumpyDL, you will need to fork the NumpyDL repository on GitHub. This will create a repository under your user account. Update your local clone to refer to the official repository as `upstream`, and your personal fork as `origin`:

```
git remote rename origin upstream
git remote add origin https://github.com/<your-github-name>/NumpyDL.git
```

If you set up an SSH key, use the SSH clone URL instead: `git@github.com:<your-github-name>/NumpyDL.git`.

You can now use this installation to develop features and send us pull requests on GitHub, see development!

CHAPTER 2

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

npdl.activation

Non-linear activation functions for artificial neurons.

Activations

Sigmoid()	Sigmoid activation function $\varphi(x) = \frac{1}{1+e^{-x}}$
Tanh()	Tanh activation function $\varphi(x) = \tanh(x)$
ReLU()	Rectify activation function $\varphi(x) = \max(0, x)$
Linear()	Linear activation function $\varphi(x) = x$
Softmax()	Softmax activation function $\varphi(\mathbf{x})_j = \frac{e^{\mathbf{x}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}_k}}$ where K is the total number of neurons in the layer.

Detailed description

class npdl.activation.**Sigmoid**

Sigmoid activation function $\varphi(x) = \frac{1}{1+e^{-x}}$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 in [0, 1]

The output of the sigmoid function applied to the activation.

class npdl.activation.**Tanh**

Tanh activation function $\varphi(x) = \tanh(x)$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 in [-1, 1]

The output of the tanh function applied to the activation.

class npdl.activation.**ReLU**

Rectify activation function $\varphi(x) = \max(0, x)$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32

The output of the rectify function applied to the activation.

class npdl.activation.**Linear**

Linear activation function $\varphi(x) = x$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32

The output of the identity applied to the activation.

class npdl.activation.**Softmax**

Softmax activation function $\varphi(\mathbf{x})_j = \frac{e^{\mathbf{x}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}_k}}$ where K is the total number of neurons in the layer. This activation function gets applied row-wise.

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 where the sum of the row is 1 and each single value is in [0, 1]

The output of the softmax function applied to the activation.

npdl.initialization

Functions to create initializers for parameter variables.

Examples

```
>>> from npdl.layers import Dense
>>> from npdl.initialization import GlorotUniform
>>> l1 = Dense(n_out=300, n_in=100, init=GlorotUniform())
```

Initializers

Zero	Initialize weights with zero value.
One	Initialize weights with one value.
Uniform([scale])	Sample initial weights from the uniform distribution.

Continued on next page

Table 2.2 – continued from previous page

Normal([std, mean])	Sample initial weights from the Gaussian distribution.
Orthogonal([gain])	Intialize weights as Orthogonal matrix.

Detailed description

class npdl.initialization.**Initializer**
Base class for parameter tensor initializers.

The *Initializer* class represents a weight initializer used to initialize weight parameters in a neural network layer. It should be subclassed when implementing new types of weight initializers.

call(*size*)
Sample should return a theano.tensor of size shape and data type theano.config.floatX.

Parameters **size** : tuple or int

Integer or tuple specifying the size of the returned matrix.

returns : theano.tensor

Matrix of size shape and dtype theano.config.floatX.

class npdl.initialization.**Zero**
Initialize weights with zero value.

class npdl.initialization.**One**
Initialize weights with one value.

class npdl.initialization.**Normal**(*std=0.01*, *mean=0.0*)
Sample initial weights from the Gaussian distribution.

Initial weight parameters are sampled from N(mean, std).

Parameters **std** : float

Std of initial parameters.

mean : float

Mean of initial parameters.

class npdl.initialization.**Uniform**(*scale=0.05*)
Sample initial weights from the uniform distribution.

Parameters are sampled from U(a, b).

Parameters **scale** : float or tuple

When std is None then range determines a, b. If range is a float the weights are sampled from U(-range, range). If range is a tuple the weights are sampled from U(range[0], range[1]).

class npdl.initialization.**Orthogonal**(*gain=1.0*)
Intialize weights as Orthogonal matrix.

Orthogonal matrix initialization *[R2]*. For n-dimensional shapes where n > 2, the n-1 trailing axes are flattened. For convolutional layers, this corresponds to the fan-in, so this makes the initialization usable for both dense and convolutional layers.

Parameters **gain** : float or 'relu'

Scaling factor for the weights. Set this to `1.0` for linear and sigmoid units, to 'relu' or `sqrt(2)` for rectified linear units, and to `sqrt(2/(1+alpha**2))` for leaky rectified linear units with leakiness `alpha`. Other transfer functions may need different factors.

References

[R2]

CHAPTER 3

Examples

This part provides examples for building deep neural networks.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[R2] Saxe, Andrew M., James L. McClelland, and Surya Ganguli. “Exact solutions to the nonlinear dynamics of learning in deep linear neural networks.” arXiv preprint arXiv:1312.6120 (2013).

Python Module Index

n

Index