yadll Documentation

Release 0.0.1

Philippe Chavanne

Sep 29, 2017

Contents

Yet **a**nother **d**eep **l**earning **l**ab.

This is an ultra light deep learning framework written in Python and based on Theano. It allows you to very quickly start building Deep Learning models and play with toy examples.

If you are looking for mature deep learning APIs I would recommend Lasagne, keras or blocks instead of yadll, they are well documented and contributed projects.

Read the documentation at Read the doc

CHAPTER 1

User Guide

The Yadll user guide explains how to install Yadll, how to build and train neural networks using many different models on mnist.

Installation

Prerequisites

We assume your are running a linux system. We assume you have Python >=2.7 with numpy and pandas. You can install it from Anaconda from Continuum Analytics

We assume you have pip:

```
sudo apt-get install pip
```

We assume you have Installed Theano.

Installation

The easiest way to install yadll is with the Python package manager `pip`:

```
git clone git@github.com:pchavanne/yadll.git
cd yadll
pip install -e .
```

GPU Support

If you have a NVIDA card you can set up CUDA and have Theano to use your GPU. See the 'Using the GPU' in the installing Theano instruction.

Tutorial

Building and training your first network

Let's build our first MLP with dropout on the MNIST example. to run this example, just do:

```
cd /yadll/examples
python model_template.py
```

We will first import yadll and configure a basic logger.

```
import os

import yadll

import logging

logging.basicConfig(level=logging.DEBUG, format='%(message)s')
```

Then we load the MNIST dataset (or download it) and create a *yadll.data.Data* instance that will hold the data. We call a loader function to retrieve the data and fill the container.

```
# load the data
data = yadll.data.Data(yadll.data.mnist_loader())
```

We now create a *yadll.model.Model*, that is the class that contain the data, the network, the hyperparameters and the updates function. As a file name is provided, the model will be saved (see *Saving/loading models*).

```
# create the model
model = yadll.model.Model(name='mlp with dropout', data=data, file='best_model.ym')
```

We define the hyperparameters(see *Hyperparameters and Grid search*) of the model and add it to our model object.

```
# Hyperparameters
hp = yadll.hyperparameters.Hyperparameters()
hp('batch_size', 500)
hp('n_epochs', 1000)
hp('learning_rate', 0.1)
hp('momentum', 0.5)
hp('l1_reg', 0.00)
hp('l2_reg', 0.0000)
hp('patience', 10000)

# add the hyperparameters to the model
model.hp = hp
```

We now create each layers of the network by implementing *yadll.layers* classes. The first layers must be a `yadll.layers.Input` that give the shape of the input data. This network will be a mlp with two dense layer with rectified linear unit activation and dropout. Each layer receive as *incoming* the previous layer. Each layer has a name. You can provide it or it will be, by default, the name of the layer class, space, the number of the instantiation. The last layer is a *yadll.layers.LogisticRegression* which is a dense layer with softmax activation. Layers names are optional.

```
# Create connected layers
# Input layer
l_in = yadll.layers.InputLayer(shape=(hp.batch_size, 28 * 28), name='Input')
```

```
# Dropout Layer 1
l_dro1 = yadll.layers.Dropout(incoming=l_in, corruption_level=0.4, name='Dropout 1')
# Dense Layer 1
l_hid1 = yadll.layers.DenseLayer(incoming=l_dro1, n_units=500, W=yadll.init.glorot_
↪uniform,
                                 l1=hp.l1_reg, l2=hp.l2_reg, activation=yadll.
↪activations.relu,
                                 name='Hidden layer 1')
# Dropout Layer 2
l_dro2 = yadll.layers.Dropout(incoming=l_hid1, corruption_level=0.2, name='Dropout 2')
# Dense Layer 2
l_hid2 = yadll.layers.DenseLayer(incoming=l_dro2, n_units=500, W=yadll.init.glorot_
↪uniform,
                                 l1=hp.l1_reg, l2=hp.l2_reg, activation=yadll.
↪activations.relu,
                                 name='Hidden layer 2')
# Logistic regression Layer
l_out = yadll.layers.LogisticRegression(incoming=l_hid2, n_class=10, l1=hp.l1_reg,
                                        l2=hp.l2_reg, name='Logistic regression')
```

We create a *yadll.network.Network* object and add all the layers sequentially. Order matters!!!

```
# Create network and add layers
net = yadll.network.Network('2 layers mlp with dropout')
net.add(l_in)
net.add(l_dro1)
net.add(l_hid1)
net.add(l_dro2)
net.add(l_hid2)
net.add(l_out)
```

We add the network and the updates function to the model and train the model. Here we update with the stochastic gradient descent with Nesterov momentum.

```
# add the network to the model
model.network = net

# updates method
model.updates = yadll.updates.nesterov_momentum

# train the model and save it to file at each best
model.train(save_mode='each')
```

Here is the output when trained on a NVIDIA Geforce Titan X card:

```
epoch 463, minibatch 100/100, validation error 1.360 %
epoch 464, minibatch 100/100, validation error 1.410 %
epoch 465, minibatch 100/100, validation error 1.400 %

 Optimization completed. Early stopped at epoch: 466
 Validation score of 1.260 % obtained at iteration 23300, with test performance 1.320
↪%
 Training mlp with dropout took 02 m 29 s
```

Making Prediction

Once the model is trained let's use it to make prediction:

```
# make prediction
# We can test it on some examples from test
test_set_x = data.test_set_x.get_value()
test_set_y = data.test_set_y.eval()

predicted_values = model.predict(test_set_x[:30])

print ("Predicted values for the first 30 examples in test set:")
print predicted_values
print test_set_y[:30]
```

This should give you

```
Predicted values for the first 30 examples in test set:
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1]
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1]
```

Saving/loading models

Yadll provides two ways to save and load models.

Save the model

This first method for saving your model is to pickle the whole model. It is not recommended for long term storage but is very convenient to handle models. All you have to do is provide you model constructor with a file name. The model will be saved after training.

```
model = yadll.model.Model(name='mlp with dropout', data=data, file='best_model.ym')
```

You can also save your model by setting the *save_mode* argument of the train function. If you didn't give a file name to the constructor it will create one (model.name + '_YmdHMS.ym'). You can set it to 'end' (save at the end of the training) or 'each' (save after each best model).

```
model.train(save_mode='each')
```

If you used 'each' and if your system crash you will be able to restart the training from the last best model.

To load the model just do

```
# load the saved model
model2 = yadll.model.load_model('best_model.ym')
```

Warning:

- Do not use this method for long term storage or production environment.
- Model trained on GPU will not be usable on CPU.

Save the network parameters

This second method is more robust and can be used for long term storage. It consists in saving the parameters (pickling) of the network.

Once the model has trained the network you can save its parameters

```
# saving network parameters
net.save_params('net_params.yp')
```

Now you can retrieve the model with those parameters, but first you have to recreate the model. When loading the parameters, the network name must match the saved parameters network name.

```
# load network parameters
# first we recreate the network
# create the model
model3 = yadll.model.Model(name='mlp with dropout', data=data,)

# Hyperparameters
hp = yadll.hyperparameters.Hyperparameters()
hp('batch_size', 500)
hp('n_epochs', 1000)
hp('learning_rate', 0.1)
hp('momentum', 0.5)
hp('l1_reg', 0.00)
hp('l2_reg', 0.0000)
hp('patience', 10000)

# add the hyperparameters to the model
model3.hp = hp

# Create connected layers
# Input layer
l_in = yadll.layers.InputLayer(shape=(hp.batch_size, 28 * 28), name='Input')
# Dropout Layer 1
l_dro1 = yadll.layers.Dropout(incoming=l_in, corruption_level=0.4, name='Dropout 1')
# Dense Layer 1
l_hid1 = yadll.layers.DenseLayer(incoming=l_dro1, n_units=500, W=yadll.init.glorot_
↪uniform,
                                 l1=hp.l1_reg, l2=hp.l2_reg, activation=yadll.
↪activations.relu,
                                 name='Hidden layer 1')
# Dropout Layer 2
l_dro2 = yadll.layers.Dropout(incoming=l_hid1, corruption_level=0.2, name='Dropout 2')
# Dense Layer 2
l_hid2 = yadll.layers.DenseLayer(incoming=l_dro2, n_units=500, W=yadll.init.glorot_
↪uniform,
                                 l1=hp.l1_reg, l2=hp.l2_reg, activation=yadll.
↪activations.relu,
                                 name='Hidden layer 2')
# Logistic regression Layer
l_out = yadll.layers.LogisticRegression(incoming=l_hid2, n_class=10, l1=hp.l1_reg,
                                        l2=hp.l2_reg, name='Logistic regression')

# Create network and add layers
net2 = yadll.network.Network('2 layers mlp with dropout')
net2.add(l_in)
net2.add(l_dro1)
net2.add(l_hid1)
```

```
net2.add(l_dro2)
net2.add(l_hid2)
net2.add(l_out)

# load params
net2.load_params('net_params.yp')    # Here we don't train the model but reload saved
↪parameters

# add the network to the model
model3.network = net2
```

Save the configuration

Models can be saved as configuration objects or files.

```
# Saving configuration of the model. Model doesn't have to be trained
conf = model.to_conf()     # get the configuration
model.to_conf('conf.yc')   # or save it to file .yc by convention
```

and reloaded:

```
# Reconstruction the model from configuration and load paramters
model4 = yadll.model.Model()
model4.from_conf(conf)          # load from conf obj
model5 = yadll.model.Model()
model5.from_conf(file='conf.yc')     # load from conf file
```

You can now reload parameters or train the network.

Networks can be modified directly from the conf object.

Note: By convention we use the .ym extension for Yadll Model file, .yp for Yadll Parameters file and .yc for configuration but it is not mandatory.

Run the examples

Yadll provide a rather exhaustive list of conventional network implementation. You will find them in the `/yadll/examples/networks.py` file.

Let's try those network on the MNIST dataset. in the `/yadll/examples/mnist_examples.py` file.

- Logisitic Regression
- Multi Layer Perceptron
- MLP with dropout
- MLP with dropconnect
- Conv Pool
- LeNet-5
- Autoencoder
- Denoising Autoencoder

- Gaussian Denoising Autoencoder
- Contractive Denoising Autoencoder
- Stacked Denoising Autoencoder
- Restricted Boltzmann Machine
- Deep Belief Network
- Recurrent Neural Networks
- Long Short-Term Memory

You can get the list of all available networks:

```
python mnist_examples.py --network_list
```

Training a model for example lenet5:

```
python mnist_examples.py lenet5
```

Hyperparameters and Grid search

Yadll provide the *yadll.hyperparameters.Hyperparameters* to hold the hyperparameters of the model. It also allows to perform a grid search optimisation as the class is iterable over all hyperparameters combinations.

Let's first define our hyperparameters and their search space

```
# Hyperparameters
hps = Hyperparameters()
hps('batch_size', 500, [50, 100, 500, 1000])
hps('n_epochs', 1000)
hps('learning_rate', 0.1, [0.001, 0.01, 0.1, 1])
hps('l1_reg', 0.00, [0, 0.0001, 0.001, 0.01])
hps('l2_reg', 0.0001, [0, 0.0001, 0.001, 0.01])
hps('activation', tanh, [tanh, sigmoid, relu])
hps('initialisation', glorot_uniform, [glorot_uniform, glorot_normal])
hps('patience', 10000)
```

Now we will loop over each possible combination

```
reports = []
for hp in hps:
    # create the model
    model = Model(name='mlp grid search', data=data)
    # add the hyperparameters to the model
    model.hp = hp
    # Create connected layers
    # Input layer
    l_in = InputLayer(shape=(None, 28 * 28), name='Input')
    # Dense Layer 1
    l_hid1 = DenseLayer(incoming=l_in, n_units=5, W=hp.initialisation, l1=hp.l1_reg,
                        l2=hp.l2_reg, activation=hp.activation, name='Hidden layer 1')
    # Dense Layer 2
    l_hid2 = DenseLayer(incoming=l_hid1, n_units=5, W=hp.initialisation, l1=hp.l1_reg,
                        l2=hp.l2_reg, activation=hp.activation, name='Hidden layer 2')
    # Logistic regression Layer
    l_out = LogisticRegression(incoming=l_hid2, n_class=10, l1=hp.l1_reg,
                               l2=hp.l2_reg, name='Logistic regression')
```

```
    # Create network and add layers
    net = Network('mlp')
    net.add(l_in)
    net.add(l_hid1)
    net.add(l_hid2)
    net.add(l_out)
    # add the network to the model
    model.network = net

    # updates method
    model.updates = yadll.updates.sgd
    reports.append((hp, model.train()))
```

Warning: These hyperparameters would generate 4*4*4*4*3*2=1536 different combinations. Each of these combinations would have a different training time but if it takes 10 minutes on average, the whole optimisation would last more the 10 days!!!

to run this example, just do:

```
python hp_grid_search.py
```

CHAPTER 2

API Reference

References on functions, classes or methods, with notes and references.

yadll.model

Model

yadll.model.**save_model**(*model*, *file=None*)

Save the model to file with cPickle This function is used by the training function to save the model. Parameters ———- model : *yadll.model.Model*

model to be saved in file

file [*string*] file name

yadll.model.**load_model**(*file*)

load (unpickle) a saved model

Parameters **file** : 'string'

file name

Returns a *yadll.model.Model*

Examples

```
>>> my_model = load_model('my_best_model.ym')
```

class yadll.model.**Model**(*network=None*, *data=None*, *hyperparameters=None*, *name='model'*, *updates=<function sgd>*, *objective=<function categorical_crossentropy_error>*, *evaluation_metric=<function categorical_accuracy>*, *file=None*)

The *yadll.model.Model* contains the data, the network, the hyperparameters, and the report. It pre-trains

unsupervised layers, trains the network and save it to file.

Parameters **network** : *yadll.network.Network*

the network to be trained

data : *yadll.data.Data*

the training, validating and testing set

name : *string*

the name of the model

updates : *yadll.updates()*

an update function

file : *string*

name of the file to save the model. If omitted a name is generated with the model name + date + time of training

compile(**args*, ***kwargs*)

Compile theano functions of the model

Parameters **compile_arg: 'string' or 'List' of 'string'**

value can be 'train', 'validate', 'test', 'predict' and 'all'

from_conf(*conf*)

build model from conf object or conf file

pretrain(**args*, ***kwargs*)

Pre-training of the unsupervised layers sequentially

Returns update unsupervised layers weights

to_conf(*file=None*)

Save model as a conf object or conf file

train(**args*, ***kwargs*)

Training the network

Parameters **unsupervised_training: 'bool', (default is True)**

pre-training of the unsupervised layers if any

save_mode : {None, 'end', 'each'}

None (default), model will not be saved unless name specified in the model definition. 'end', model will only be saved at the end of the training 'each', model will be saved each time the model is improved

early_stop : *bool*, (default is True)

early stopping when validation score is not improving

shuffle : *bool*, (default is True)

reshuffle the training set at each epoch. Batches will then be different from one epoch to another

Returns report

yadll.network

Network

class yadll.network.**Network**(*name=None*, *layers=None*)

The *Network* class is the container of all the layers of the network.

Parameters **name** : *string*

The name of the network

layers : list of :class: *Layer*, optional

create a network from another network.layers

Attributes

layers	(list of :class: *Layer*) the list of layers in the network
params	(list of *theano shared variables*) the list of all the parameters of the network
reguls	(symbolic expression) regularization cost for the network
has_unsupervised_layer	(*bool*) True if one of the layer is a subclass of :class: *UnsupervisedLayer*

add(*layer*)

add a layer to the Network

Parameters **layer** : :class: *Layer*

get_layer(*layer_name*)

Get a layer of the network from its name

Parameters **layer_name** : *string*

name of the layer requested

Returns yaddl.layers object

a yadll layer object in the

get_output(***kwargs*)

Returns the output of the network

Returns symbolic expresssion

output of the network

load_params(*file*)

load (unpickle) saved parameters of a network.

Parameters **file** : 'string'

name of the file containing the saved parameters

Examples

```
>>> my_network.load_params('my_network_params.yp')
```

save_params(*file*)

Save the parameters of the network to file with cPickle

Parameters **file** : *string*

file name

Examples

```
>>> my_network.save_params('my_network_params.yp')
```

yadll.data

Data

yadll.data.**normalize**(*x*)
Normalization: Scale data to [0, 1]

$$z = (x - min(x))/(max(x) - min(x))$$

Parameters **x: numpy array**

Returns z, min, max

yadll.data.**apply_normalize**(*x*, *x_min*, *x_max*)
Apply normalization to data given min and max

yadll.data.**revert_normalize**(*z*, *x_min*, *x_max*)
Return x given z, min and max

yadll.data.**standardize**(*x*, *epsilon=1e-06*)
Standardization: Scale to mean=0 and std=1

$$z = (x - mean(x))/std(x)$$

Parameters **x: numpy array**

Returns z, mean, std

yadll.data.**apply_standardize**(*x*, *x_mean*, *x_std*)
Apply standardization to data given mean and std

yadll.data.**revert_standardize**(*z*, *x_mean*, *x_std*)
Return x given z, mean and std

yadll.data.**one_hot_encoding**(*arr*, *N=None*)
One hot encoding of a vector of integer categorical variables in a range [0..N].

You can provide the higher category N or max(arr) will be used.

Parameters **arr** : numpy array

array of integer in a range [0, N]

N : *int*, optional

Higher category

Returns one hot encoding [0, 1, 0, 0]

Examples

```
>>> a = np.asarray([1, 0, 3])
>>> one_hot_encoding(a)
array([[ 0.,  1.,  0.,  0.],
       [ 1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> one_hot_encoding(a, 5)
array([[ 0.,  1.,  0.,  0.,  0.,  0.],
   [ 1.,  0.,  0.,  0.,  0.,  0.],
   [ 0.,  0.,  0.,  1.,  0.,  0.]])
```

yadll.data.**one_hot_decoding**(*mat*)

decoding of a one hot matrix

Parameters **mat** : numpy matrix

one hot matrix

Returns vector of decoded value

Examples

```
>>> a = np.asarray([[0, 1, 0, 0], [1, 0, 0, 0], [0, 0, 0, 1]])
>>> one_hot_decoding(a)
array([1, 0, 3])
```

class yadll.data.**Data**(*data*, *preprocessing=None*, *shared=True*, *borrow=True*, *cast_y=False*)

Data container.

data is made of train_set, valid_set, test_set and set_x, set_y = set

Parameters **data** : *string*

data file name (with path)

shared : *bool*

theano shared variable

borrow : *bool*

theano borrowable variable

cast_y : *bool*

cast y to *intX*

Examples

Load data

```
>>> yadll.data.Data('data/mnist/mnist.pkl.gz')
```

Methods

dataset :	return the dataset as Theano shared variables [(train_set_x, train_set_y), (valid_set_x, valid_set_y), (test_set_x, test_set_y)]

yadll.hyperparameters

Hyperparameters

class yadll.hyperparameters.**Hyperparameters**

Container class for the hyperparameters. Define each parameters with a name and a default value and optionally a list of values that will be iterated over during a grid search.

It create an iterable of all the different parameters values combination.

Parameters **name** : *string*, {'batch_size', 'n_epochs', 'learning_rate', 'l1_reg', 'l2_reg', 'patience'}

The name of the hyperparameter.

value : *float*

The default value of the hyperparameter.

range : *list* of *float*

A list of values iterated over during the gris search

Examples

Define the hyperparameters of the model:

```
>>> hp = Hyperparameters()    # Create an Hyperparameters instance
>>> hp('batch_size', 500)     # Define an hyperparameter with its default value
>>> hp('n_epochs', 1000, [10, 100, 1000, 1000])    # and range for the grid search
```

Grid search on the hyperparameters space:

```
>>> for param in hp:
>>>     # Do something with this set of hyperparameters
```

Methods

reset	reset all hyperparameters to default values.

yadll.layers

Layers

The Layers classes implement one layer of neural network of different types. the ::class:Layer is the mother class of all the layers and has to be inherited by any new layer. All the neural network layers currently supported by yaddll.

Layer(incoming[, name])	Layer is the base class of any neural network layer.
InputLayer(input_shape[, input])	Input layer of the data, it has no parameters, it just shapes the data as the input for any network.
ReshapeLayer(incoming[, output_shape])	Reshape the incoming layer to the output_shape.
FlattenLayer(incoming[, n_dim])	Reshape layers back to flat
Activation(incoming[, activation])	Apply activation function to previous layer
DenseLayer(incoming, n_units[, W, b, ...])	Fully connected neural network layer

Continued on next page

Table 2.1 – continued from previous page

UnsupervisedLayer(incoming, n_units, ...)	Base class for all unsupervised layers.
LogisticRegression(incoming, n_class[, W, ...])	Dense layer with softmax activation
Dropout(incoming[, corruption_level])	Dropout layer
Dropconnect(incoming, n_units[, ...])	DropConnect layer
PoolLayer(incoming, pool_size[, stride, ...])	Pooling layer, default is maxpooling
ConvLayer(incoming[, image_shape, ...])	Convolutional layer
ConvPoolLayer(incoming, pool_size[, ...])	Convolutional and pooling layer
AutoEncoder(incoming, n_units, hyperparameters)	Autoencoder
RBM(incoming, n_units, hyperparameters[, W, ...])	Restricted Boltzmann Machines
BatchNormalization(incoming[, axis, alpha, ...])	Normalize the input layer over each mini-batch according to [R3333]:
RNN(incoming, n_units[, n_out, activation, ...])	Recurrent Neural Network
LSTM(incoming, n_units[, peepholes, ...])	Long Short Term Memory
GRU(incoming, n_units[, activation, ...])	Gated Recurrent unit

Detailed description

class yadll.layers.**Layer**(*incoming*, *name=None*, ***kwargs*)

Layer is the base class of any neural network layer. It has to be subclassed by any kind of layer.

Parameters **incoming** : a *Layer* , a *List* of *Layers* or a *tuple* of *int*

The incoming layer, a list of incoming layers or the shape of the input layer

name : *string*, optional

The layer name. default name is the class name plus instantiation number i.e: 'Dense-Layer 3'

get_output(***kwargs*)

Return the output of this layer

Raises **NotImplementedError**

This method has to be overriden by new layer implementation.

get_params()

Theano shared variables representing the parameters of this layer.

Returns list of Theano shared variables that parametrize the layer

get_reguls()

Theano expression representing the sum of the regulators of this layer.

Returns Theano expression representing the sum of the regulators

of this layer

output_shape

Compute the output shape of this layer given the input shape.

Returns a tuple representing the shape of the output of this layer.

Notes

This method has to be overriden by new layer implementation or will return the input shape.

class yadll.layers.**InputLayer**(*input_shape*, *input=None*, ***kwargs*)
Input layer of the data, it has no parameters, it just shapes the data as the input for any network. A :*InputLayer* is always the first layer of any network.

class yadll.layers.**ReshapeLayer**(*incoming*, *output_shape=None*, ***kwargs*)
Reshape the incoming layer to the output_shape.

class yadll.layers.**FlattenLayer**(*incoming*, *n_dim=2*, ***kwargs*)
Reshape layers back to flat

class yadll.layers.**DenseLayer**(*incoming*, *n_units*, *W=<function glorot_uniform>*, *b=<function constant>*, *activation=<function tanh>*, *l1=None*, *l2=None*, ***kwargs*)
Fully connected neural network layer

class yadll.layers.**Activation**(*incoming*, *activation=<function linear>*, ***kwargs*)
Apply activation function to previous layer

class yadll.layers.**UnsupervisedLayer**(*incoming*, *n_units*, *hyperparameters*, ***kwargs*)
Base class for all unsupervised layers. Unsupervised layers are pre-trained against its own input.

class yadll.layers.**LogisticRegression**(*incoming*, *n_class*, *W=<function constant>*, *activation=<function softmax>*, ***kwargs*)
Dense layer with softmax activation

References

[R5757]

class yadll.layers.**Dropout**(*incoming*, *corruption_level=0.5*, ***kwargs*)
Dropout layer

class yadll.layers.**Dropconnect**(*incoming*, *n_units*, *corruption_level=0.5*, ***kwargs*)
DropConnect layer

class yadll.layers.**PoolLayer**(*incoming*, *pool_size*, *stride=None*, *ignore_border=True*, *pad=(0, 0)*, *mode='max'*, ***kwargs*)
Pooling layer, default is maxpooling

class yadll.layers.**ConvLayer**(*incoming*, *image_shape=None*, *filter_shape=None*, *W=<function glorot_uniform>*, *border_mode='valid'*, *subsample=(1, 1)*, *l1=None*, *l2=None*, *pool_scale=None*, ***kwargs*)
Convolutional layer

class yadll.layers.**ConvPoolLayer**(*incoming*, *pool_size*, *image_shape=None*, *filter_shape=None*, *b=<function constant>*, *activation=<function tanh>*, ***kwargs*)
Convolutional and pooling layer

References

[R5959]

class yadll.layers.**AutoEncoder**(*incoming*, *n_units*, *hyperparameters*, *corruption_level=0.0*, *W=(<function glorot_uniform>, {'gain': <function sigmoid>})*, *b_prime=<function constant>*, *sigma=None*, *contraction_level=None*, ***kwargs*)
Autoencoder

References

[R6161]

class yadll.layers.**RBM**(*incoming*, *n_units*, *hyperparameters*, *W=<function glorot_uniform>*, *b_hidden=<function constant>*, *activation=<function sigmoid>*, ***kwargs*)
Restricted Boltzmann Machines

References

[R6363]

class yadll.layers.**BatchNormalization**(*incoming*, *axis=-2*, *alpha=0.1*, *epsilon=1e-05*, *has_beta=True*, ***kwargs*)
Normalize the input layer over each mini-batch according to *[R6565]*:

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}}$$
$$y = \gamma * \hat{x} + \beta$$

Warning: When a BatchNormalization layer is used the batch size has to be given at compile time. You can not use None as the first dimension anymore. Prediction has to be made on the same batch size.

References

[R6565]

class yadll.layers.**RNN**(*incoming*, *n_units*, *n_out=None*, *activation=<function sigmoid>*, *last_only=True*, *grad_clipping=0*, *go_backwards=False*, *allow_gc=False*, ***kwargs*)
Recurrent Neural Network

$$h_t = \sigma(x_t.W + h_{t-1}.U + b)$$

References

[R6769], *[R6869]*, *[R6969]*

class yadll.layers.**LSTM**(*incoming*, *n_units*, *peepholes=False*, *tied_i_f=False*, *activation=<function tanh>*, *last_only=True*, *grad_clipping=0*, *go_backwards=False*, *allow_gc=False*, ***kwargs*)

Long Short Term Memory

$$
\begin{aligned}
i_t &= \sigma(x_t.W_i + h_{t-1}.U_i + b_i) \\
f_t &= \sigma(x_t.W_f + h_{t-1}.U_f + b_f) \\
\tilde{C}_t &= \tanh(x_t.W_c + h_{t-1}.U_c + b_c) \\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
o_t &= \sigma(x_t.W_o + h_{t-1}.U_o + b_o) \\
h_t &= o_t * \tanh(C_t)
\end{aligned}
$$

with Peephole connections:

$$
\begin{aligned}
i_t &= \sigma(x_t.W_i + h_{t-1}.U_i + C_{t-1}.P_i + b_i) \\
f_t &= \sigma(x_t.W_f + h_{t-1}.U_f + C_{t-1}.P_f + b_f) \\
\tilde{C}_t &= \tanh(x_t.W_c + h_{t-1}.U_c + b_c) \\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
o_t &= \sigma(x_t.W_o + h_{t-1}.U_o + C_t.P_o + b_o) \\
h_t &= o_t * \tanh(C_t)
\end{aligned}
$$

with tied forget and input gates:

$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$

Parameters **incoming** : a *Layer*

The incoming layer with an output_shape = (n_batches, n_time_steps, n_dim)

n_units : int

n_hidden = n_input_gate = n_forget_gate = n_cell_gate = n_output_gate = n_units All gates have the same number of units

n_out : int

number of output units

peephole : boolean default is False

use peephole connections.

tied_i : boolean default is false

tie input and forget gate

activation : *yadll.activations* function default is *yadll.activations.tanh*

activation function

last_only : boolean default is True

set to true if you only need the last element of the output sequence. Theano will optimize graph.

References

[R7377], *[R7477]*, *[R7577]*, *[R7677]*, *[R7777]*

class `yadll.layers.`**`GRU`**(*incoming*, *n_units*, *activation=<function tanh>*, *last_only=True*, *grad_clipping=0*, *go_backwards=False*, *allow_gc=False*, ***kwargs*)

Gated Recurrent unit

$$z_t = \sigma(x_t.W_z + h_{t-1}.U_z + b_z)$$
$$r_t = \sigma(x_t.W_r + h_{t-1}.U_r + b_r)$$
$$\tilde{h_t} = \tanh(x_t.W_h + (r_t * h_{t-1}).U_h + b_h)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h_t}$$

References

[R8385], *[R8485]*, *[R8585]*

yadll.updates

Updates

Updating functions that are passed to the network for optimization. Updates functions

Arguments

cost [cost function] The cost function that will be minimised during training

params [list of parameters] The list of all the weights of the network that will be modified

sgd(cost, params[, learning_rate])	Stochastic Gradient Descent (SGD) updates
momentum(cost, params[, learning_rate, momentum])	Stochastic Gradient Descent (SGD) updates with momentum
nesterov_momentum(cost, params[, ...])	Stochastic Gradient Descent (SGD) updates with Nesterov momentum
adagrad(cost, params[, learning_rate, epsilon])	Adaptive Gradient Descent
rmsprop(cost, params[, learning_rate, rho, ...])	RMSProp updates
adadelta(cost, params[, learning_rate, rho, ...])	Adadelta Gradient Descent
adam(cost, params[, learning_rate, beta1, ...])	Adam Gradient Descent
adamax(cost, params[, learning_rate, beta1, ...])	Adam Gradient Descent
nadam(cost, params[, learning_rate, rho, ...])	Adam Gradient Descent with nesterov momentum

Detailed description

yadll.updates.**sgd**(*cost*, *params*, *learning_rate=0.1*, ***kwargs*)
Stochastic Gradient Descent (SGD) updates

*param := param - learning_rate * gradient*

yadll.updates.**momentum**(*cost*, *params*, *learning_rate=0.1*, *momentum=0.9*, ***kwargs*)
Stochastic Gradient Descent (SGD) updates with momentum

*velocity := momentum * velocity - learning_rate * gradient*

param := param + velocity

yadll.updates.**nesterov_momentum**(*cost*, *params*, *learning_rate=0.1*, *momentum=0.9*, ***kwargs*)
Stochastic Gradient Descent (SGD) updates with Nesterov momentum

*velocity := momentum * velocity - learning_rate * gradient*

*param := param + momentum * velocity - learning_rate * gradient*

References

[R101101]

yadll.updates.**adagrad**(*cost*, *params*, *learning_rate=0.1*, *epsilon=1e-06*, ***kwargs*)
Adaptive Gradient Descent Scale learning rates by dividing with the square root of accumulated squared gradients

References

[R103103]

yadll.updates.**rmsprop**(*cost*, *params*, *learning_rate=0.01*, *rho=0.9*, *epsilon=1e-06*, ***kwargs*)
RMSProp updates Scale learning rates by dividing with the moving average of the root mean squared (RMS) gradients

yadll.updates.**adadelta**(*cost*, *params*, *learning_rate=0.1*, *rho=0.95*, *epsilon=1e-06*, ***kwargs*)
Adadelta Gradient Descent Scale learning rates by a the ratio of accumulated gradients to accumulated step sizes

References

[R105105]

yadll.updates.**adam**(*cost*, *params*, *learning_rate=0.001*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-06*, ***kwargs*)
Adam Gradient Descent Scale learning rates by Adaptive moment estimation

References

[R107107]

yadll.updates.**adamax**(*cost*, *params*, *learning_rate=0.001*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-06*, ***kwargs*)
Adam Gradient Descent Scale learning rates by adaptive moment estimation

References

[R109109]

yadll.updates.**nadam**(*cost*, *params*, *learning_rate=1.0*, *rho=0.95*, *epsilon=1e-06*, ***kwargs*)
Adam Gradient Descent with nesterov momentum

References

[R111111]

yadll.init

Initializers

constant(shape[, value, name, borrow])	Initialize all the weights to a constant value
uniform(shape[, scale, name, borrow])	Initialize all the weights from the uniform distribution
normal(shape[, scale, name, borrow])	Initialize all the weights from the normal distribution
glorot_uniform(shape[, gain, name, fan, borrow])	Initialize all the weights from the uniform distribution with glorot scaling
glorot_normal(shape[, gain, name, fan, borrow])	Initialize all the weights from the normal distribution with glorot scaling
He_uniform(shape[, name, borrow])	
He_normal(shape[, name, borrow])	
selu_normal(shape[, name, borrow])	
orthogonal(shape[, gain, name, borrow])	Orthogonal initialization for Recurrent Networks

Detailed description

yadll.init.**initializer**(*init_obj*, *shape*, *name*, ***kwargs*)

Call an Initializer from an init_obj

Parameters **init_obj** : *init_obj*

an init_obj is an initializer function or the tuple of (initializer function, dict of args) example : init_obj = glorot_uniform or init_obj = (glorot_uniform, {'gain':tanh, 'borrow':False})

shape : *tuple* or int

shape of the return shared variables

Returns

——-

Initialized shared variables

yadll.init.**constant**(*shape*, *value=0.0*, *name=None*, *borrow=True*, ***kwargs*)

Initialize all the weights to a constant value

Parameters **shape**

scale

yadll.init.**uniform**(*shape*, *scale=0.5*, *name=None*, *borrow=True*, ***kwargs*)

Initialize all the weights from the uniform distribution

Parameters **shape**

scale

name

borrow

kwargs

yadll.init.**normal**(*shape*, *scale=0.5*, *name=None*, *borrow=True*, ***kwargs*)

Initialize all the weights from the normal distribution

Parameters **shape**

scale

name

borrow

kwargs

yadll.init.**glorot_uniform**(*shape*, *gain=1.0*, *name=None*, *fan=None*, *borrow=True*, ***kwargs*)
Initialize all the weights from the uniform distribution with glorot scaling

Parameters **shape**

gain

name

fan

borrow

kwargs

References

[R1717]

yadll.init.**glorot_normal**(*shape*, *gain=1*, *name=None*, *fan=None*, *borrow=True*, ***kwargs*)
Initialize all the weights from the normal distribution with glorot scaling

Parameters **shape**

gain

name

fan

borrow

kwargs

References

[R1919]

yadll.init.**He_uniform**(*shape*, *name=None*, *borrow=True*, ***kwargs*)

yadll.init.**He_normal**(*shape*, *name=None*, *borrow=True*, ***kwargs*)

yadll.init.**selu_normal**(*shape*, *name=None*, *borrow=True*, ***kwargs*)

yadll.init.**orthogonal**(*shape*, *gain=1*, *name=None*, *borrow=True*, ***kwargs*)
Orthogonal initialization for Recurrent Networks

Orthogonal initialization solve the vanishing/exploding gradient for recurrent network.

Parameters **shape**

gain

name

borrow

kwargs

References

[R2122], [R2222]

yadll.activations

Activation

Activation functions

get_activation(activator)	Call an activation function from an activator object
linear(x)	Linear activation function
sigmoid(x)	Sigmoid function
ultra_fast_sigmoid(x)	Ultra fast Sigmoid function return an approximated standard sigmoid
tanh(x)	Tanh activation function
softmax(x)	Softmax activation function
softplus(x)	Softplus activation function $\varphi(x) = \log 1 + \exp x$
relu(x[, alpha])	Rectified linear unit activation function
elu(x[, alpha])	Compute the element-wise exponential linear activation function.

Detailed description

yadll.activations.**get_activation**(*activator*)
Call an activation function from an activator object

Parameters **activator** : *activator*

an activator is an activation function, a tuple of (activation function, dict of args), the name of the activation function as a str or a tuple (name of function, dict of args) example : activator = tanh or activator = (elu, {'alpha':0.5})

or activator = 'tanh' or activator = ('elu', {'alpha':0.5})

Returns an activation function

yadll.activations.**linear**(*x*)
Linear activation function :math: *varphi(x)* = *x*

Parameters **x** : symbolic tensor

Tensor to compute the activation function for.

Returns symbolic tensor

The output of the identity applied to the activation *x*.

yadll.activations.**sigmoid**(*x*)
Sigmoid function $\varphi(x) = \frac{1}{1+\exp -x}$

Parameters **x** : symbolic tensor

Tensor to compute the activation function for.

Returns symbolic tensor of value in [0, 1]

The output of the sigmoid function applied to the activation *x*.

yadll.activations.**ultra_fast_sigmoid**(*x*)

Ultra fast Sigmoid function return an approximated standard sigmoid $\varphi(x) = \frac{1}{1+\exp -x}$

Parameters **x** : symbolic tensor

Tensor to compute the activation function for.

Returns symbolic tensor of value in [0, 1]

The output of the sigmoid function applied to the activation *x*.

Notes

Use the Theano flag optimizer_including=local_ultra_fast_sigmoid to use

ultra_fast_sigmoid systematically instead of sigmoid.

yadll.activations.**tanh**(*x*)

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

Parameters **x** : symbolic tensor

Tensor to compute the activation function for.

Returns symbolic tensor of value in [-1, 1]

The output of the tanh function applied to the activation *x*.

yadll.activations.**softmax**(*x*)

Softmax activation function $\varphi(x)_j = \frac{\exp x_j}{\sum_{k=1}^{K} \exp x_k}$

where K is the total number of neurons in the layer. This activation function gets applied row-wise.

Parameters **x** : symbolic tensor

Tensor to compute the activation function for.

Returns symbolic tensor 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 *x*.

yadll.activations.**softplus**(*x*)

Softplus activation function $\varphi(x) = \log 1 + \exp x$

Parameters **x** : symbolic tensor

Tensor to compute the activation function for.

Returns symbolic tensor

The output of the softplus function applied to the activation *x*.

yadll.activations.**relu**(*x*, *alpha=0*)

Rectified linear unit activation function $\varphi(x) = \max x, \alpha * x$

Parameters **x** : symbolic tensor

Tensor to compute the activation function for.

alpha : scalar or tensor, optional

Slope for negative input, usually between 0 and 1. The default value of 0 will lead to the standard rectifier, 1 will lead to a linear activation function, and any value in between will give a leaky rectifier. A shared variable (broadcastable against *x*) will result in a parameterized rectifier with learnable slope(s).

Returns symbolic tensor

Element-wise rectifier applied to the activation *x*.

Notes

This is numerically equivalent to `T.switch(x > 0, x, alpha * x)` (or `T.maximum(x, alpha * x)` for `alpha < 1`), but uses a faster formulation or an optimized Op, so we encourage to use this function.

References

[R55]

`yadll.activations.`**`elu`**(*x*, *alpha=1*)

Compute the element-wise exponential linear activation function.

Parameters **x** : symbolic tensor

Tensor to compute the activation function for.

alpha : scalar

Returns symbolic tensor

Element-wise exponential linear activation function applied to *x*.

References

yadll.objectives

Objectives

mean_squared_error(prediction, target)	Mean Squared Error:
root_mean_squared_error(prediction, target)	Root Mean Squared Error:
mean_absolute_error(prediction, target)	Mean Absolute Error:
binary_hinge_error(prediction, target)	Binary Hinge Error:
categorical_hinge_error(prediction, target)	Categorical Hinge Error:
binary_crossentropy_error(prediction, target)	Binary Cross-entropy Error:
categorical_crossentropy_error(prediction, ...)	Categorical Cross-entropy Error:
kullback_leibler_divergence(prediction, target)	Kullback Leibler Divergence:

Detailed description

`yadll.objectives.`**`mean_squared_error`**(*prediction*, *target*)

Mean Squared Error:

$$MSE_i = \frac{1}{n}\sum_j (prediction_{i,j} - target_{i,j})^2$$

`yadll.objectives.`**`root_mean_squared_error`**(*prediction*, *target*)

Root Mean Squared Error:

$$RMSE_i = \sqrt{\frac{1}{n}\sum_j (target_{i,j} - prediction_{i,j})^2}$$

yadll.objectives.**mean_absolute_error**(*prediction*, *target*)
Mean Absolute Error:

$$MAE_i = \frac{1}{n}\sum_j |target_{i,j} - prediction_{i,j}|$$

yadll.objectives.**binary_hinge_error**(*prediction*, *target*)
Binary Hinge Error:

$$BHE_i = \frac{1}{n}\sum_j \max(0, 1 - target_{i,j} * prediction_{i,j})$$

yadll.objectives.**categorical_hinge_error**(*prediction*, *target*)
Categorical Hinge Error:

$$CHE_i = \frac{1}{n}\sum_j \max(1 - target_{i,j} * prediction_{i,j}, 0)$$

yadll.objectives.**binary_crossentropy_error**(*prediction*, *target*)
Binary Cross-entropy Error:

$$BCE_i = -\frac{1}{n}\sum_j (target_{i,j} * \log(prediction_{i,j}) - (1 - target_{i,j}) * \log(1 - prediction_{i,j}))$$

yadll.objectives.**categorical_crossentropy_error**(*prediction*, *target*)
Categorical Cross-entropy Error:

$$CCE_i = -\frac{1}{n}\sum_j target_{i,j} * \log(prediction_{i,j})$$

yadll.objectives.**kullback_leibler_divergence**(*prediction*, *target*)
Kullback Leibler Divergence:

$$KLD_i = \sum_j target_{i,j} * \log(fractarget_{i,j}prediction_{i,j})$$

yadll.utils

Utils

yadll.utils.**to_float_X**(*arr*)
Cast to floatX numpy array

Parameters **arr: list or numpy array**

Returns numpy array of floatX

yadll.utils.**shared_variable**(*value*, *dtype='float64'*, *name=None*, *borrow=True*, ***kwargs*)
Create a Theano Shared Variable

Parameters **value:**

value of the shared variable

dtype : default floatX

type of the shared variable

name : string, optional

shared variable name

borrow : bool, default is True

if True shared variable we construct does not get a [deep] copy of value. So changes we subsequently make to value will also change our shared variable.

Returns Theano Shared Variable

yadll.utils.**format_sec**(*sec*)
format a time

Parameters **sec** : float

time in seconds

Returns string :

formatted time in days, hours, minutes and seconds

yadll.utils.**timer**(*what_to_show='Function execution'*)
decorator that send the execution time of the argument function to the logger

Parameters **what_to_show** : *string*, optional

message displayed after execution

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[R5757] http://deeplearning.net/tutorial/logreg.html

[R5959] http://deeplearning.net/tutorial/lenet.html

[R6161] http://deeplearning.net/tutorial/dA.html

[R6363] http://deeplearning.net/tutorial/rbm.html

[R6565] http://jmlr.org/proceedings/papers/v37/ioffe15.pdf

[R6769] http://deeplearning.net/tutorial/rnnslu.html

[R6869] https://arxiv.org/pdf/1602.06662.pdf

[R6969] https://arxiv.org/pdf/1511.06464.pdf

[R7377] http://deeplearning.net/tutorial/lstm.html

[R7477] http://christianherta.de/lehre/dataScience/machineLearning/neuralNetworks/LSTM.php

[R7577] http://people.idsia.ch/~juergen/lstm/

[R7677] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[R7777] https://arxiv.org/pdf/1308.0850v5.pdf

[R8385] http://deeplearning.net/tutorial/lstm.html

[R8485] https://arxiv.org/pdf/1412.3555.pdf

[R8585] http://jmlr.org/proceedings/papers/v37/jozefowicz15.pdf

[R101101] https://github.com/lisa-lab/pylearn2/pull/136#issuecomment-10381617

[R103103] http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

[R105105] https://arxiv.org/pdf/1212.5701v1.pdf

[R107107] https://arxiv.org/pdf/1412.6980v8.pdf

[R109109] https://arxiv.org/pdf/1412.6980v8.pdf

[R111111] http://cs229.stanford.edu/proj2015/054_report.pdf

[R1717] http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

[R1919] http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

[R2122] http://smerity.com/articles/2016/orthogonal_init.html

[R55] Xavier Glorot, Antoine Bordes and Yoshua Bengio (2011): Deep sparse rectifier neural networks. AISTATS. http://jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

[R77] Djork-Arne Clevert, Thomas Unterthiner, Sepp Hochreiter

"Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)" <http://arxiv.org/abs/1511.07289>'.

Python Module Index

y

Index

N

O

P

R

S

T

U

Y