
numpydl Documentation

Release 0.4.0

NumpyDL

May 08, 2021

Contents

1	API References	3
1.1	npdl.layers	3
1.2	npdl.activations	16
1.3	npdl.initializations	27
1.4	npdl.objectives	28
1.5	npdl.optimizers	32
1.6	npdl.model	36
1.7	npdl.utils	37
2	Indices and tables	39
	Bibliography	41
	Python Module Index	43
	Index	45

NumpyDL is a simple deep learning library based on pure Python/Numpy. NumpyDL is a work in progress, input is welcome. The project is on [GitHub](#).

The main features of NumpyDL are as follows:

1. *Pure* in Numpy
2. *Native* to Python
3. *Automatic differentiations* are basically supported
4. *Commonly used models* are provided: MLP, RNNs, LSTMs and CNNs
5. *API* like Keras library
6. *Examples* for several AI tasks
7. *Application* for a toy chatbot

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

1.1 npdl.layers

1.1.1 Base Layers

class npdl.layers.Layer

The *Layer* class represents a single layer of a neural network. It should be subclassed when implementing new types of layers.

Because each layer can keep track of the layer(s) feeding into it, a network's output *Layer* instance can double as a handle to the full network.

backward (*pre_grad*, *args, **kwargs)
calculate the input gradient

connect_to (*prev_layer*)
Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input*, *args, **kwargs)
Calculate layer output for given input (forward propagation).

classmethod from_json (*config*)
From configuration

grads
Get layer parameter gradients as calculated from backward().

param_grads
Layer parameters and corresponding gradients.

params

Layer parameters.

Returns a list of `numpy.array` variables or expressions that parameterize the layer.

Returns

list of `numpy.array` variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

to_json()

To configuration

1.1.2 Core Layers

class `npdl.layers.Linear` (*n_out*, *n_in=None*, *init='glorot_uniform'*)

A fully connected layer implemented as the dot product of inputs and weights.

Parameters

n_out [(int, tuple)] Desired size or shape of layer output

n_in [(int, tuple) or None] The layer input size feeding into this layer

init [(Initializer, optional)] Initializer object to use for initializing layer weights

backward (*pre_grad*, **args*, ***kwargs*)

Apply the backward pass transformation to the input data.

Parameters

pre_grad [numpy.array] deltas back propagated from the adjacent higher layer

Returns

numpy.array deltas to propagate to the adjacent lower layer

connect_to (*prev_layer=None*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input*, **args*, ***kwargs*)

Apply the forward pass transformation to the input data.

Parameters

input [numpy.array] input data

Returns

numpy.array output data

grads

Get layer parameter gradients as calculated from `backward()`.

params

Layer parameters.

Returns a list of `numpy.array` variables or expressions that parameterize the layer.

Returns

list of `numpy.array` variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

class `npdl.layers.Dense` (*n_out*, *n_in=None*, *init='glorot_uniform'*, *activation='tanh'*)

A fully connected layer implemented as the dot product of inputs and weights. Generally used to implement nonlinearities for layer post activations.

Parameters

n_out [int] Desired size or shape of layer output

n_in [int, or None] The layer input size feeding into this layer

activation [str, or `npdl.activatns.Activation`] Defaults to `Tanh`

init [str, or `npdl.initializations.Initializer`] Initializer object to use for initializing layer weights

backward (*pre_grad*, **args*, ***kwargs*)

Apply the backward pass transformation to the input data.

Parameters

pre_grad [`numpy.array`] deltas back propagated from the adjacent higher layer

Returns

`numpy.array` deltas to propagate to the adjacent lower layer

connect_to (*prev_layer=None*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input*, **args*, ***kwargs*)

Apply the forward pass transformation to the input data.

Parameters

input [`numpy.array`] input data

Returns

`numpy.array` output data

grads

Get layer parameter gradients as calculated from `backward()`.

params

Layer parameters.

Returns a list of `numpy.array` variables or expressions that parameterize the layer.

Returns

list of `numpy.array` variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

class `numpydl.layers.Softmax` (*n_out*, *n_in=None*, *init='glorot_uniform'*)

A fully connected layer implemented as the dot product of inputs and weights.

Parameters

n_out [int] Desired size or shape of layer output

n_in [int, or None] The layer input size feeding into this layer

init [str, or `numpydl.initializations.Initializer`] Initializer object to use for initializing layer weights

class `numpydl.layers.Dropout` (*p=0.0*)

A dropout layer.

Applies an element-wise multiplication of inputs with a keep mask.

A keep mask is a tensor of ones and zeros of the same shape as the input.

Each `forward()` call generates a new keep mask stochastically where the distribution of ones in the mask is controlled by the keep param.

Parameters

p [float] fraction of the inputs that should be stochastically kept.

backward (*pre_grad*, **args*, ***kwargs*)

calculate the input gradient

connect_to (*prev_layer*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input*, *train=True*, **args*, ***kwargs*)

Apply the forward pass transformation to the input data.

Parameters

input [numpy.array] input data

train [bool] is inference only

Returns

numpy.array output data

1.1.3 Convolution Layers

class `numpydl.layers.Convolution` (*nb_filter*, *filter_size*, *input_shape=None*, *stride=1*,
init='glorot_uniform', *activation='relu'*)

Convolution operator for filtering windows of two-dimensional inputs.

When using this layer as the first layer in a model, provide the keyword argument `input_shape` (tuple of integers, does not include the sample axis), e.g. `input_shape=(3, 128, 128)` for 128x128 RGB pictures.

backward (*pre_grad*, **args*, ***kwargs*)

calculate the input gradient

connect_to (*prev_layer=None*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input, *args, **kwargs*)

Calculate layer output for given input (forward propagation).

grads

Get layer parameter gradients as calculated from backward().

params

Layer parameters.

Returns a list of numpy.array variables or expressions that parameterize the layer.

Returns

list of numpy.array variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

1.1.4 Embedding Layer

class npdl.layers.**Embedding** (*embed_words=None, static=None, input_size=None, n_out=None, nb_batch=None, nb_seq=None, init='uniform'*)

backward (*pre_grad, *args, **kwargs*)

calculate the input gradient

connect_to (*prev_layer=None*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input, *args, **kwargs*)

Calculate layer output for given input (forward propagation).

grads

Get layer parameter gradients as calculated from backward().

params

Layer parameters.

Returns a list of numpy.array variables or expressions that parameterize the layer.

Returns

list of numpy.array variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

1.1.5 Normalization Layer

class `npdl.layers.BatchNormal` (*epsilon=1e-06*, *momentum=0.9*, *axis=0*, *beta_init='zero'*,
gamma_init='one')

Batch normalization layer (Ioffe and Szegedy, 2014) [1].

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Parameters

epsilon **small float > 0** Fuzz parameter. npdl expects epsilon $\geq 1e-5$.

axis [integer] axis along which to normalize in mode 0. For instance, if your input tensor has shape (samples, channels, rows, cols), set axis to 1 to normalize per feature map (channels axis).

momentum [float] momentum in the computation of the exponential average of the mean and standard deviation of the data, for feature-wise normalization.

beta_init [npdl.initializations.Initializer] name of initialization function for shift parameter, or alternatively, npdl function to use for weights initialization.

gamma_init [npdl.initializations.Initializer] name of initialization function for scale parameter, or alternatively, npdl function to use for weights initialization.

Input shape Arbitrary. Use the keyword argument *input_shape* (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output shape Same shape as input.

References

[1]

backward (*pre_grad*, **args*, ***kwargs*)
 calculate the input gradient

connect_to (*prev_layer*)
 Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input*, **args*, ***kwargs*)
 Calculate layer output for given input (forward propagation).

params

Layer parameters.

Returns a list of numpy.array variables or expressions that parameterize the layer.

Returns

list of numpy.array variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

1.1.6 Pooling Layers

class `npdl.layers.MeanPooling` (*pool_size*)

Average pooling operation for spatial data.

Parameters

pool_size [tuple of 2 integers,] factors by which to downscale (vertical, horizontal). (2, 2) will halve the image in each dimension.

Returns

4D numpy.array with shape (*nb_samples*, *channels*, *pooled_rows*, *pooled_cols*) if `dim_ordering='th'` or 4D tensor with shape: (*samples*, *pooled_rows*, *pooled_cols*, *channels*) if `dim_ordering='tf'`.

backward (*pre_grad*, **args*, ***kwargs*)

calculate the input gradient

connect_to (*prev_layer*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input*, **args*, ***kwargs*)

Calculate layer output for given input (forward propagation).

class `npdl.layers.MaxPooling` (*pool_size*)

Max pooling operation for spatial data.

Parameters

pool_size [tuple of 2 integers,] factors by which to downscale (vertical, horizontal). (2, 2) will halve the image in each dimension.

Returns

4D numpy.array with shape (*nb_samples*, *channels*, *pooled_rows*, *pooled_cols*) if `dim_ordering='th'` or 4D tensor with shape: (*samples*, *pooled_rows*, *pooled_cols*, *channels*) if `dim_ordering='tf'`.

backward (*pre_grad*, **args*, ***kwargs*)

calculate the input gradient

connect_to (*prev_layer*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input*, **args*, ***kwargs*)

Calculate layer output for given input (forward propagation).

1.1.7 Recurrent Layers

class `npdl.layers.Recurrent` (*n_out*, *n_in=None*, *nb_batch=None*, *nb_seq=None*, *init='glorot_uniform'*, *inner_init='orthogonal'*, *activation='tanh'*, *return_sequence=False*)

A recurrent neural network (RNN) is a class of artificial neural network where connections between units

form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected handwriting recognition [Ra9eb24862aa8-1] or speech recognition. [Ra9eb24862aa8-2]

Parameters

n_out [int] hidden number

n_in [int or None] input dimension

nb_batch [int or None] batch size

nb_seq [int or None] sequent length

init [npdl.intializations.Initliazer] init function

inner_init [npdl.intializations.Initliazer] inner init function, between hidden to hidden

activation [npdl.activations.Activation] activation function

return_sequence [bool] return total sequence or not.

References

[1], [2]

connect_to (*prev_layer=None*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

class npdl.layers.SimpleRNN (***kwargs*)

Fully-connected RNN where the output is to be fed back to input.

$$o_t = \tanh(U_t x_t + W_t o_{t-1} + b_t)$$

Parameters

output_dim: dimension of the internal projections and the final output.

init: weight initialization function. Can be the name of an existing function (str), or a npdl function.

inner_init: initialization function of the inner cells.

activation: activation function. Can be the name of an existing function (str), or a npdl function.

return_sequence: if 'return_sequences', 3D 'numpy.array' with shape (*batch_size*, *timesteps*, *units*) will be returned. Else, return 2D *numpy.array* with shape (*batch_size*, *units*).

References

[1]

backward (*pre_grad*, **args*, ***kwargs*)

calculate the input gradient

connect_to (*prev_layer=None*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input, *args, **kwargs*)

Calculate layer output for given input (forward propagation).

grads

Get layer parameter gradients as calculated from backward().

params

Layer parameters.

Returns a list of numpy.array variables or expressions that parameterize the layer.

Returns

list of numpy.array variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

class npdl.layers.GRU (*gate_activation='sigmoid', need_grad=True, **kwargs*)

Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks, introduced in 2014. Their performance on polyphonic music modeling and speech signal modeling was found to be similar to that of long short-term memory.[R55b2e00ce55d-1]_ They have fewer parameters than LSTM, as they lack an output gate.[R55b2e00ce55d-2]_

$$z_t = \sigma(U_z x_t + W_z h_{t-1} + b_z)$$

$$z_t = r_t = \sigma(U_r x_t + W_r h_{t-1} + b_r)$$

$$h_t = \tanh(U_h x_t + W_h (s_{t-1} \odot r_t) + b_h)$$

$$s_t = (1 - z_t) \odot h_t + z_t \odot s_{t-1}$$

Parameters

gate_activation [npdl.activations.Activation] Gate activation.

need_grad bool If *True*, will calculate gradients.

References

[1], [2]

backward (*pre_grad, *args, **kwargs*)

calculate the input gradient

connect_to (*prev_layer=None*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input, *args, **kwargs*)

Calculate layer output for given input (forward propagation).

grads

Get layer parameter gradients as calculated from backward().

params

Layer parameters.

Returns a list of numpy.array variables or expressions that parameterize the layer.

Returns

list of numpy.array variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

class `npdl.layers.LSTM`(*gate_activation='sigmoid', need_grad=True, forget_bias_num=1, **kwargs*)

Batch LSTM, support mask, but not support training.

Long short-term memory (LSTM) is a recurrent neural network (RNN) architecture (an artificial neural network) proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber [1] and further improved in 2000 by Felix Gers et al.[Rc4406f3688f4-2] Like most RNNs, a LSTM network is universal in the sense that given enough network units it can compute anything a conventional computer can compute, provided it has the proper weight matrix, which may be viewed as its program.

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$

$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$$

$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$$

$$g_t = \tanh(U_g x_t + W_g h_{t-1} + b_g)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$

Parameters

gate_activation [`npdl.activations.Activation`] Gate activation.

need_grad **bool** If *True*, will calculate gradients.

forget_bias_num [`int`] integer.

References

[1], [2]

backward (*pre_grad, *args, **kwargs*)
calculate the input gradient

connect_to (*prev_layer=None*)
Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [`previous layer`] The previous layer to propagate through this layer.

forward (*input, mask, c0=None, h0=None*)
Calculate layer output for given input (forward propagation).

grads

Get layer parameter gradients as calculated from backward().

params

Layer parameters.

Returns a list of numpy.array variables or expressions that parameterize the layer.

Returns

list of numpy.array variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

class npdl.layers.**BatchLSTM**(*gate_activation='sigmoid', need_grad=True, forget_bias_num=1, **kwargs*)

Batch LSTM, support training, but not support mask.

Parameters

gate_activation [npdl.activations.Activation] Gate activation.

need_grad **bool** If *True*, will calculate gradients.

forget_bias_num [int] integer.

References

[1], [2]

backward (*pre_grad, dcn=None, dh=None*)

Backward propagation.

Parameters

pre_grad [numpy.array] Gradients propagated to this layer.

dcn [numpy.array] Gradients of cell state at *n* time step.

dh [numpy.array] Gradients of hidden state at *n* time step.

Returns

numpy.array The gradients propagated to previous layer.

connect_to (*prev_layer=None*)

Connection to the previous layer.

Parameters

prev_layer [npdl.layers.Layer or None] Previous layer.

AllW [numpy.array]

type	i	f	o	g
bias				
x2h				
h2h				

forward (*input*, *c0=None*, *h0=None*)

Forward propagation.

Parameters

input [numpy.array] input should be of shape (nb_batch,nb_seq,n_in)

c0 [numpy.array or None] init cell state

h0 [numpy.array or None] init hidden state

Returns

numpy.array Forward results.

grads

Get layer parameter gradients as calculated from backward().

params

Layer parameters.

Returns a list of numpy.array variables or expressions that parameterize the layer.

Returns

list of numpy.array variables or expressions A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

1.1.8 Shape Layers

class `npdl.layers.Flatten` (*outdim=2*)

backward (*pre_grad*, **args*, ***kwargs*)

calculate the input gradient

connect_to (*prev_layer*)

Propagates the given input through this layer (and only this layer).

Parameters

prev_layer [previous layer] The previous layer to propagate through this layer.

forward (*input*, **args*, ***kwargs*)

Calculate layer output for given input (forward propagation).

Base Layers

Layer

The *Layer* class represents a single layer of a neural network.

Core Layers

<i>Linear</i>	A fully connected layer implemented as the dot product of inputs and weights.
<i>Dense</i>	A fully connected layer implemented as the dot product of inputs and weights.
<i>Softmax</i>	A fully connected layer implemented as the dot product of inputs and weights.
<i>Dropout</i>	A dropout layer.

Convolution Layers

<i>Convolution</i>	Convolution operator for filtering windows of two-dimensional inputs.
--------------------	---

Embedding Layer

<i>Embedding</i>	
------------------	--

Normalization Layer

<i>BatchNormal</i>	Batch normalization layer (Ioffe and Szegedy, 2014) [1]
--------------------	---

Pooling Layers

<i>MeanPooling</i>	Average pooling operation for spatial data.
<i>MaxPooling</i>	Max pooling operation for spatial data.

Recurrent Layers

<i>Recurrent</i>	A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed cycle.
<i>SimpleRNN</i>	Fully-connected RNN where the output is to be fed back to input.
<i>GRU</i>	Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks, introduced in 2014.
<i>LSTM</i>	Batch LSTM, support mask, but not support training.
<i>BatchLSTM</i>	Batch LSTM, support training, but not support mask.

Shape Layers

<i>Flatten</i>	
----------------	--

1.2 npdl. activations

Non-linear activation functions for artificial neurons.

A function used to transform the activation level of a unit (neuron) into an output signal. Typically, activation functions have a “squashing” effect. Together with the PSP function (which is applied first) this defines the unit type. Neural Networks supports a wide range of activation functions.

1.2.1 Activations

<code>Activation()</code>	Base class for activations.
<code>Sigmoid()</code>	Sigmoid activation function.
<code>Tanh()</code>	Tanh activation function.
<code>ReLU()</code>	Rectify activation function.
<code>Linear()</code>	Linear activation function.
<code>Softmax()</code>	Softmax activation function.
<code>Elliot([steepness])</code>	A fast approximation of sigmoid.
<code>SymmetricElliot([steepness])</code>	Elliot symmetric sigmoid transfer function.
<code>SoftPlus()</code>	Softplus activation function.
<code>SoftSign()</code>	SoftSign activation function.

1.2.2 Detailed Description

class `npdl.activations.Activation`

Base class for activations.

derivative (*input=None*)

Backward step.

Parameters

input [numpy.array, optional.] If provide *input*, this function will not use *last_forward*.

forward (*input*)

Forward Step.

Parameters

input [numpy.array] the input matrix.

class `npdl.activations.Sigmoid`

Sigmoid activation function.

derivative (*input=None*)

The derivative of sigmoid is

$$\begin{aligned} \frac{dy}{dx} &= (1 - \varphi(x)) \otimes \varphi(x) \\ &= \frac{e^{-x}}{(1 + e^{-x})^2} \\ &= \frac{e^x}{(1 + e^x)^2} \end{aligned}$$

Returns

float32 The derivative of sigmoid function.

forward (*input*, **args*, ***kwargs*)

A sigmoid function is a mathematical function having a characteristic “S”-shaped curve or sigmoid curve. Often, sigmoid function refers to the special case of the logistic function and defined by the formula $\varphi(x) = \frac{1}{1+e^{-x}}$ (given the input x).

Parameters

input [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 Sigmoid(Activation):
    """Sigmoid activation function.
    """

    def __init__(self):
        super(Sigmoid, self).__init__()

    def forward(self, input, *args, **kwargs):
        """A sigmoid function is a mathematical function having a
        characteristic "S"-shaped curve or sigmoid curve. Often,
        sigmoid function refers to the special case of the logistic
        function and defined by the formula :math:`\varphi(x) = \frac{1}{1 + e^{-x}}`
        (given the input :math:`x`).

```

class npdl.activations.Tanh

Tanh activation function.

The hyperbolic tangent function is an old mathematical function. It was first used in the work by L'Abbe Sauri (1774).

derivative (*input=None*)

The derivative of `tanh()` functions is

$$\begin{aligned} \frac{d}{dx} \tanh(x) &= \frac{d}{dx} \frac{\sinh(x)}{\cosh(x)} \\ &= \frac{\cosh(x) \frac{d}{dx} \sinh(x) - \sinh(x) \frac{d}{dx} \cosh(x)}{\cosh^2(x)} \\ &= \frac{\cosh(x) \cosh(x) - \sinh(x) \sinh(x)}{\cosh^2(x)} \\ &= 1 - \tanh^2(x) \end{aligned}$$

Returns

float32 The derivative of tanh function.

forward (*input*)

This function is easily defined as the ratio between the hyperbolic sine and the cosine functions (or expanded, as the ratio of the half-difference and half-sum of two exponential functions in the points z and $-z$):

$$\begin{aligned} \tanh(z) &= \frac{\sinh(z)}{\cosh(z)} \\ &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \end{aligned}$$

Fortunately, numpy provides `tanh()` methods. So in our implementation, we directly use $\varphi(x) = \tanh(x)$.

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 Tanh(Activation):
    """Tanh activation function.

    The hyperbolic tangent function is an old mathematical function.
    It was first used in the work by L'Abbe Sauri (1774).
    """

    def __init__(self):
        super(Tanh, self).__init__()

    def forward(self, input):
        """This function is easily defined as the ratio between the hyperbolic
        sine and the cosine functions (or expanded, as the ratio of the
        half-difference and half-sum of two exponential functions in the
        points :math:`z` and :math:`-z`):"""
```

(continues on next page)

(continued from previous page)

```

.. math:: \tanh(z) &= \frac{\sinh(z)}{\cosh(z)} \\
&= \frac{e^z - e^{-z}}{e^z + e^{-z}}

Fortunately, numpy provides :meth:`tanh` methods. So in our implementation,
we directly use :math:`\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.
"""
self.last_forward = np.tanh(input)
return self.last_forward

def derivative(self, input=None):
    """The derivative of :meth:`tanh` functions is

.. math:: \frac{d}{dx} \tanh(x) &= \frac{d}{dx} \frac{\sinh(x)}{\cosh(x)} \\
&= \frac{\cosh(x) \frac{d}{dx} \sinh(x) - \sinh(x) \frac{d}{dx} \cosh(x)}{\cosh^2(x)} \\
&= \frac{\cosh(x) \cosh(x) - \sinh(x) \sinh(x)}{\cosh^2(x)} \\
&= 1 - \tanh^2(x)

Returns
-----
float32
    The derivative of tanh function.
"""
last_forward = self.forward(input) if input else self.last_forward
return 1 - np.power(last_forward, 2)

```

class npdl.activations.ReLU

Rectify activation function.

Two additional major benefits of ReLUs are sparsity and a reduced likelihood of vanishing gradient. But first recall the definition of a ReLU is $h = \max(0, a)$ where $a = Wx + b$.

One major benefit is the reduced likelihood of the gradient to vanish. This arises when $a > 0$. In this regime the gradient has a constant value. In contrast, the gradient of sigmoids becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.

The other benefit of ReLUs is sparsity. Sparsity arises when $a \leq 0$. The more such units that exist in a layer the more sparse the resulting representation. Sigmoids on the other hand are always likely to generate some non-zero value resulting in dense representations. Sparse representations seem to be more beneficial than dense representations.

derivative (*input=None*)

The point-wise derivative for ReLU is $\frac{dy}{dx} = 1$, if $x > 0$, or $\frac{dy}{dx} = 0$, if $x \leq 0$.

Returns

float32 The derivative of ReLU function.

forward (*input*)

During the forward pass, it inhibits all inhibitions below some threshold , typically 0. In other words, it computes point-wise

$$y = \max(0, x)$$

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 ReLU(Activation):
    """Rectify activation function.

    Two additional major benefits of ReLUs are sparsity and a reduced
    likelihood of vanishing gradient. But first recall the definition
    of a ReLU is :math:`h=\max(0,a)` where :math:`a=Wx+b`.

    One major benefit is the reduced likelihood of the gradient to vanish.
    This arises when :math:`a>0`. In this regime the gradient has a constant value.
    In contrast, the gradient of sigmoids becomes increasingly small as the
    absolute value of :math:`x` increases. The constant gradient of ReLUs results in
    faster learning.

    The other benefit of ReLUs is sparsity. Sparsity arises when :math:`a0`.
    The more such units that exist in a layer the more sparse the resulting
    representation. Sigmoids on the other hand are always likely to generate
    some non-zero value resulting in dense representations. Sparse representations
    seem to be more beneficial than dense representations.
    """

    def __init__(self):
        super(ReLU, self).__init__()

    def forward(self, input):
        """During the forward pass, it inhibits all inhibitions below some
        threshold :math:``, typically :math:`0`. In other words, it computes point-
        ↪wise

        .. math:: y=\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.
        """
```

(continues on next page)

(continued from previous page)

```

self.last_forward = input
return np.maximum(0.0, input)

def derivative(self, input=None):
    """The point-wise derivative for ReLU is  $\frac{dy}{dx} = 1$ , if
     $x > 0$ , or  $\frac{dy}{dx} = 0$ , if  $x \leq 0$ .

    Returns
    -----
    float32
        The derivative of ReLU function.
    """
    last_forward = input if input else self.last_forward
    res = np.zeros(last_forward.shape, dtype=get_dtype())
    res[last_forward > 0] = 1.
    return res

```

class npdl.activations.**Linear**

Linear activation function.

derivative (*input=None*)

Backward propagation. The backward also return identity matrix.

Returns

float32 The derivative of linear function.

forward (*input*)

It's also known as identity activation funtion. The forward step is $\varphi(x) = x$

Parameters

x [float32] The activation (the summed, weighted input of a neuron).

Returns

float32 The output of the identity applied to the activation.

```

class Linear(Activation):
    """Linear activation function.
    """

    def __init__(self):
        super(Linear, self).__init__()

    def forward(self, input):
        """It's also known as identity activation funtion. The
        forward step is  $\varphi(x) = x$ 

        Parameters
        -----
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns

```

(continues on next page)

(continued from previous page)

```

-----
float32
    The output of the identity applied to the activation.
    """
self.last_forward = input
return input

def derivative(self, input=None):
    """Backward propagation.
    The backward also return identity matrix.

    Returns
    -----
    float32
        The derivative of linear function.
    """
last_forward = input if input else self.last_forward
return np.ones(last_forward.shape, dtype=get_dtype())

```

class npdl.activations.**Softmax**

Softmax activation function.

derivative (*input=None*)

Backward propagation.

Returns

float32 The derivative of Softmax function.

forward (*input*)

$\varphi(\mathbf{x})_j = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}$ 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.

```

class Softmax(Activation):
    """Softmax activation function.
    """

    def __init__(self):
        super(Softmax, self).__init__()

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

```

(continues on next page)

(continued from previous page)

```

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.
"""
assert np.ndim(input) == 2
self.last_forward = input
x = input - np.max(input, axis=1, keepdims=True)
exp_x = np.exp(x)
s = exp_x / np.sum(exp_x, axis=1, keepdims=True)
return s

def derivative(self, input=None):
    """Backward propagation.

Returns
-----
float32
    The derivative of Softmax function.
"""
last_forward = input if input else self.last_forward
return np.ones(last_forward.shape, dtype=get_dtype())

```

class npdl.activations.**Elliot** (*steepness=1*)

A fast approximation of sigmoid.

The function was first introduced in 1993 by D.L. Elliot under the title A Better Activation Function for Artificial Neural Networks. The function closely approximates the Sigmoid or Hyperbolic Tangent functions for small values, however it takes longer to converge for large values (i.e. It doesn't go to 1 or 0 as fast), though this isn't particularly a problem if you're using it for classification.

derivative (*input=None*)

Backward propagation.

Returns

float32 The derivative of Elliot function.

forward (*input*)

Forward propagation.

Parameters

x [float32] The activation (the summed, weighted input of a neuron).

Returns

float32 The output of the softplus function applied to the activation.

```
class Elliot(Activation):
```

(continues on next page)

```

""" A fast approximation of sigmoid.

The function was first introduced
in 1993 by D.L. Elliot under the title A Better Activation Function for
Artificial Neural Networks. The function closely approximates the
Sigmoid or Hyperbolic Tangent functions for small values, however it
takes longer to converge for large values (i.e. It doesn't go to 1 or
0 as fast), though this isn't particularly a problem if you're using
it for classification.

"""

def __init__(self, steepness=1):
    super(Elliot, self).__init__()

    self.steepness = steepness

def forward(self, input):
    """Forward propagation.

    Parameters
    -----
    x : float32
        The activation (the summed, weighted input of a neuron).

    Returns
    -----
    float32
        The output of the softplus function applied to the activation.
    """
    self.last_forward = 1 + np.abs(input * self.steepness)
    return 0.5 * self.steepness * input / self.last_forward + 0.5

def derivative(self, input=None):
    """Backward propagation.

    Returns
    -----
    float32
        The derivative of Elliot function.
    """
    last_forward = 1 + np.abs(input * self.steepness) if input else self.last_
↪forward
    return 0.5 * self.steepness / np.power(last_forward, 2)

```

class npdl.activations.SymmetricElliot (steepness=1)

Elliot symmetric sigmoid transfer function.

derivative (input=None)

Backward propagation.

Returns

float32 The derivative of SymmetricElliot function.

forward (input)

Forward propagation.

Parameters

x [float32] The activation (the summed, weighted input of a neuron).

Returns

float32 The output of the softplus function applied to the activation.

```
class SymmetricElliot(Activation):
    """Elliot symmetric sigmoid transfer function.

    """

    def __init__(self, steepness=1):
        super(SymmetricElliot, self).__init__()
        self.steepness = steepness

    def forward(self, input):
        """Forward propagation.

        Parameters
        -----
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -----
        float32
            The output of the softplus function applied to the activation.
        """
        self.last_forward = 1 + np.abs(input * self.steepness)
        return input * self.steepness / self.last_forward

    def derivative(self, input=None):
        """Backward propagation.

        Returns
        -----
        float32
            The derivative of SymmetricElliot function.
        """
        last_forward = 1 + np.abs(input * self.steepness) if input else self.last_
        ↪forward
        return self.steepness / np.power(last_forward, 2)
```

class npdl.activations.**SoftPlus**

Softplus activation function.

derivative (*input=None*)

Backward propagation.

Returns

float32 The derivative of Softplus function.

forward (*input*)

$$\varphi(x) = \log(1 + e^x)$$

Parameters

x [float32] The activation (the summed, weighted input of a neuron).

Returns

float32 The output of the softplus function applied to the activation.

```
class SoftPlus(Activation):
    """Softplus activation function.
    """

    def __init__(self):
        super(SoftPlus, self).__init__()

    def forward(self, input):
        """math: \varphi(x) = \log(1 + e^x)

        Parameters
        -----
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -----
        float32
            The output of the softplus function applied to the activation.
        """
        self.last_forward = np.exp(input)
        return np.log(1 + self.last_forward)

    def derivative(self, input=None):
        """Backward propagation.

        Returns
        -----
        float32
            The derivative of Softplus function.
        """
        last_forward = np.exp(input) if input else self.last_forward
        return last_forward / (1 + last_forward)
```

class npdl.activations.**SoftSign**

SoftSign activation function.

derivative (*input=None*)

Backward propagation.

Returns

float32 The derivative of SoftSign function.

forward (*input*)

Forward propagation.

Parameters

x [float32] The activation (the summed, weighted input of a neuron).

Returns

float32 The output of the softplus function applied to the activation.

```
class SoftSign(Activation):
    """SoftSign activation function.

    """

    def __init__(self):
        super(SoftSign, self).__init__()

    def forward(self, input):
        """Forward propagation.

        Parameters
        -----
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -----
        float32
            The output of the softplus function applied to the activation.
        """
        self.last_forward = np.abs(input) + 1
        return input / self.last_forward

    def derivative(self, input=None):
        """Backward propagation.

        Returns
        -----
        float32
            The derivative of SoftSign function.
        """
        last_forward = np.abs(input) + 1 if input else self.last_forward
        return 1. / np.power(last_forward, 2)
```

1.3 npdl.initializations

1.3.1 Initializers

Zero

One

Uniform

Continued on next page

Table 10 – continued from previous page

Normal

Orthogonal

1.3.2 Detailed Description

1.4 npdl.objectives

Provides some minimal help with building loss expressions for training or validating a neural network.

These functions build element- or item-wise loss expressions from network predictions and targets.

1.4.1 Examples

Assuming you have a simple neural network for 3-way classification:

```
>>> import npdl
>>> model = npdl.model.Model()
>>> model.add(npdl.layers.Dense(n_out=100, n_in=50))
>>> model.add(npdl.layers.Dense(n_out=3, activation=npdl.activations.Softmax()))
>>> model.compile(loss=npdl.objectives.SCCE(), optimizer=npdl.optimizers.SGD(lr=0.
↪005))
```

1.4.2 Objectives

<i>MeanSquaredError</i>	Computes the element-wise squared difference between targets and outputs.
<i>MSE</i>	alias of <i>npdl.objectives.MeanSquaredError</i>
<i>HellingerDistance</i>	Computes the multi-class hinge loss between predictions and targets.
<i>HeD</i>	alias of <i>npdl.objectives.HellingerDistance</i>
<i>BinaryCrossEntropy</i>	Computes the binary cross-entropy between predictions and targets.
<i>BCE</i>	alias of <i>npdl.objectives.BinaryCrossEntropy</i>
<i>SoftmaxCategoricalCrossEntropy</i>	Computes the categorical cross-entropy between predictions and targets.
<i>SCCE</i>	alias of <i>npdl.objectives.SoftmaxCategoricalCrossEntropy</i>

1.4.3 Detailed Description

class `npdl.objectives.Objective`

An objective function (or loss function, or optimization score function) is one of the two parameters required to compile a model.

backward (*outputs, targets*)
Backward function.

Parameters

outputs, targets [numpy.array] The arrays to compute the derivatives of them.

Returns

numpy.array An array of derivative.

forward (*outputs, targets*)
Forward function.

class npdl.objectives.**MeanSquaredError**

Computes the element-wise squared difference between *targets* and *outputs*.

In statistics, the mean squared error (MSE) or mean squared deviation (MSD) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the squares of the errors or deviations—that is, the difference between the estimator and what is estimated. MSE is a risk function, corresponding to the expected value of the squared error loss or quadratic loss. The difference occurs because of randomness or because the estimator doesn't account for information that could produce a more accurate estimate. [1]

The MSE is a measure of the quality of an estimator—it is always non-negative, and values closer to zero are better.

The MSE is the second moment (about the origin) of the error, and thus incorporates both the variance of the estimator and its bias. For an unbiased estimator, the MSE is the variance of the estimator. Like the variance, MSE has the same units of measurement as the square of the quantity being estimated. In an analogy to standard deviation, taking the square root of MSE yields the root-mean-square error or root-mean-square deviation (RMSE or RMSD), which has the same units as the quantity being estimated; for an unbiased estimator, the RMSE is the square root of the variance, known as the standard deviation.

Notes

This is the loss function of choice for many regression problems or auto-encoders with linear output units.

References

[1]

backward (*outputs, targets*)
MeanSquaredError backward propagation.

$$dE = p - t$$

Parameters

outputs, targets [numpy.array] The arrays to compute the derivative between them.

Returns

numpy.array Derivative.

forward (*outputs, targets*)
MeanSquaredError forward propagation.

$$L = (p - t)^2$$

Parameters

outputs, targets [numpy.array] The arrays to compute the squared difference between.

Returns

numpy.array An expression for the element-wise squared difference.

`npdl.objectives.MSE`
alias of `npdl.objectives.MeanSquaredError`

class `npdl.objectives.HellingerDistance`
Computes the multi-class hinge loss between predictions and targets.

In probability and statistics, the Hellinger distance (closely related to, although different from, the Bhattacharyya distance) is used to quantify the similarity between two probability distributions. It is a type of f-divergence. The Hellinger distance is defined in terms of the Hellinger integral, which was introduced by Ernst Hellinger in 1909.[\[Re7ebbd157ef9-1\]_ \[2\]](#)

Notes

This is an alternative to the categorical cross-entropy loss for multi-class classification problems

References

[1], [2]

backward (*outputs, targets*)
HellingerDistance forward propagation.

forward (*outputs, targets*)
HellingerDistance forward propagation.

Parameters

outputs [numpy 2D array] outputs in (0, 1), such as softmax output of a neural network, with data points in rows and class probabilities in columns.

targets [numpy 2D array] Either a vector of int giving the correct class index per data point or a 2D tensor of one-hot encoding of the correct class in the same layout as predictions (non-binary targets in [0, 1] do not work!)

Returns

numpy 1D array An expression for the Hellinger Distance

`npdl.objectives.HeD`
alias of `npdl.objectives.HellingerDistance`

class `npdl.objectives.BinaryCrossEntropy` (*epsilon=1e-11*)
Computes the binary cross-entropy between predictions and targets.

Returns

numpy array An expression for the element-wise binary cross-entropy.

Notes

This is the loss function of choice for binary classification problems and sigmoid output units.

backward (*outputs, targets*)
Backward pass.

Parameters

outputs [numpy.array] Predictions in (0, 1), such as sigmoidal output of a neural network.

targets [numpy.array] Targets in [0, 1], such as ground truth labels.

forward (*outputs*, *targets*)

Forward pass.

$$L = -t \log(p) - (1 - t) \log(1 - p)$$

Parameters

outputs [numpy.array] Predictions in (0, 1), such as sigmoidal output of a neural network.

targets [numpy.array] Targets in [0, 1], such as ground truth labels.

`numpydl.objectives.BCE`

alias of `numpydl.objectives.BinaryCrossEntropy`

class `numpydl.objectives.SoftmaxCategoricalCrossEntropy` (*epsilon=1e-11*)

Computes the categorical cross-entropy between predictions and targets.

Notes

This is the loss function of choice for multi-class classification problems and softmax output units. For hard targets, i.e., targets that assign all of the probability to a single class per data point, providing a vector of int for the targets is usually slightly more efficient than providing a matrix with a single 1.0 per row.

backward (*outputs*, *targets*)

SoftmaxCategoricalCrossEntropy backward propagation.

$$dE = p - t$$

Parameters

outputs [numpy 2D array] Predictions in (0, 1), such as softmax output of a neural network, with data points in rows and class probabilities in columns.

targets [numpy 2D array] Either targets in [0, 1] matching the layout of *outputs*, or a vector of int giving the correct class index per data point.

Returns

numpy 1D array

forward (*outputs*, *targets*)

SoftmaxCategoricalCrossEntropy forward propagation.

$$L_i = - \sum_j t_{i,j} \log(p_{i,j})$$

Parameters

outputs [numpy.array] Predictions in (0, 1), such as softmax output of a neural network, with data points in rows and class probabilities in columns.

targets [numpy.array] Either targets in [0, 1] matching the layout of *outputs*, or a vector of int giving the correct class index per data point.

Returns

numpy 1D array An expression for the item-wise categorical cross-entropy.

`numpydl.objectives.SCCE`

alias of `numpydl.objectives.SoftmaxCategoricalCrossEntropy`

1.5 npdl.optimizers

Functions to generate Theano update dictionaries for training.

The update functions implement different methods to control the learning rate for use with stochastic gradient descent.

Update functions take a loss expression or a list of gradient expressions and a list of parameters as input and return an ordered dictionary of updates:

1.5.1 Examples

Using *SGD* to define an update dictionary for a toy example network:

```
>>> import npdl
>>> from npdl.activations import ReLU
>>> from npdl.activations import Softmax
>>> from npdl.objectives import SCCE
>>> model = npdl.model.Model()
>>> model.add(npdl.layers.Dense(n_out=100, n_in=50, activation=ReLU()))
>>> model.add(npdl.layers.Dense(n_out=200, activation=ReLU()))
>>> model.add(npdl.layers.Dense(n_out=100, activation=ReLU()))
>>> model.add(npdl.layers.Dense(n_out=10, activation=Softmax()))
>>> model.compile(loss=SCCE(), optimizer=npdl.optimizers.SGD(lr=0.005))
```

1.5.2 Optimizers

<i>SGD</i>	Stochastic Gradient Descent (SGD) updates
<i>Momentum</i>	Stochastic Gradient Descent (SGD) updates with momentum
<i>NesterovMomentum</i>	Stochastic Gradient Descent (SGD) updates with Nesterov momentum
<i>Adagrad</i>	Adagrad updates
<i>RMSprop</i>	RMSProp updates
<i>Adadelta</i>	Adadelta updates
<i>Adam</i>	Adam updates
<i>Adamax</i>	Adamax updates

1.5.3 Detailed Description

class npdl.optimizers.**SGD**(*args, **kwargs)
Stochastic Gradient Descent (SGD) updates

Generates update expressions of the form:

- $\text{param} := \text{param} - \text{learning_rate} * \text{gradient}$

update (params, grads)

Update parameters.

Parameters

params [list] A list of parameters in model.

grads [list] A list of gradients in model.

class `numpydl.optimizers.Momentum` (*momentum=0.9, *args, **kwargs*)
 Stochastic Gradient Descent (SGD) updates with momentum

Generates update expressions of the form:

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

Parameters

momentum [float] The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Notes

Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by $1 - \text{momentum}$.

update (*params, grads*)
 Update parameters.

Parameters

params [list] A list of parameters in model.

grads [list] A list of gradients in model.

class `numpydl.optimizers.NesterovMomentum` (*momentum=0.9, *args, **kwargs*)
 Stochastic Gradient Descent (SGD) updates with Nesterov momentum

Generates update expressions of the form:

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

Parameters

momentum [float] The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Notes

Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by $1 - \text{momentum}$.

The classic formulation of Nesterov momentum (or Nesterov accelerated gradient) requires the gradient to be evaluated at the predicted next position in parameter space. Here, we use the formulation described at <https://github.com/lisa-lab/pylearn2/pull/136#issuecomment-10381617>, which allows the gradient to be evaluated at the current parameters.

update (*params, grads*)
 Update parameters.

Parameters

params [list] A list of parameters in model.

grads [list] A list of gradients in model.

class `numpydl.optimizers.Adagrad` (*epsilon=1e-06, *args, **kwargs*)
 Adagrad updates

Scale learning rates by dividing with the square root of accumulated squared gradients. See [1] for further description.

Parameters

epsilon [float] Small value added for numerical stability.

Notes

Using step size eta Adagrad calculates the learning rate for feature i at time step t as:

$$\eta_{t,i} = \frac{\eta}{\sqrt{\sum_{t'}^t g_{t',i}^2 + \epsilon}} g_{t,i}$$

as such the learning rate is monotonically decreasing.

Epsilon is not included in the typical formula, see [2].

References

[1], [2]

update (*params, grads*)
 Update parameters.

Parameters

params [list] A list of parameters in model.

grads [list] A list of gradients in model.

class `numpydl.optimizers.RMSprop` (*rho=0.9, epsilon=1e-06, *args, **kwargs*)
 RMSProp updates

Scale learning rates by dividing with the moving average of the root mean squared (RMS) gradients. See [1] for further description.

Parameters

rho [float] Gradient moving average decay factor.

epsilon [float] Small value added for numerical stability.

Notes

rho should be between 0 and 1. A value of *rho* close to 1 will decay the moving average slowly and a value close to 0 will decay the moving average fast.

Using the step size η and a decay factor ρ the learning rate η_t is calculated as:

$$r_t = \rho r_{t-1} + (1 - \rho) * g^2$$

$$\eta_t = \frac{\eta}{\sqrt{r_t + \epsilon}}$$

References

[1]

update (*params*, *grads*)
Update parameters.

Parameters

params [list] A list of parameters in model.

grads [list] A list of gradients in model.

class `numpydl.optimizers.Adadelta` (*rho=0.9*, *epsilon=1e-06*, **args*, ***kwargs*)
Adadelta updates

Scale learning rates by the ratio of accumulated gradients to accumulated updates, see [1] and notes for further description.

Parameters

rho [float] Gradient moving average decay factor.

epsilon [float] Small value added for numerical stability.

decay [float] Decay parameter for the moving average.

Notes

rho should be between 0 and 1. A value of rho close to 1 will decay the moving average slowly and a value close to 0 will decay the moving average fast.

rho = 0.95 and epsilon=1e-6 are suggested in the paper and reported to work for multiple datasets (MNIST, speech).

In the paper, no learning rate is considered (so learning_rate=1.0). Probably best to keep it at this value. epsilon is important for the very first update (so the numerator does not become 0).

Using the step size eta and a decay factor rho the learning rate is calculated as:

$$r_t = \rho r_{t-1} + (1 - \rho) * g^2$$

$$\eta_t = \eta \frac{\sqrt{s_{t-1} + \epsilon}}{\sqrt{r_t + \epsilon}}$$

$$s_t = \rho s_{t-1} + (1 - \rho) * (\eta_t * g)^2$$

References

[1]

update (*params*, *grads*)
Update parameters.

Parameters

params [list] A list of parameters in model.

grads [list] A list of gradients in model.

class `numpydl.optimizers.Adam` (*beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, **args*, ***kwargs*)
Adam updates

Adam updates implemented as in [1].

Parameters

- beta1** [float] Exponential decay rate for the first moment estimates.
- beta2** [float] Exponential decay rate for the second moment estimates.
- epsilon** [float] Constant for numerical stability.

Notes

The paper [1] includes an additional hyperparameter lambda. This is only needed to prove convergence of the algorithm and has no practical use (personal communication with the authors), it is therefore omitted here.

References

[1]

update (*params*, *grads*)
Update parameters.

Parameters

- params** [list] A list of parameters in model.
- grads** [list] A list of gradients in model.

class `numpydl.optimizers.Adamax` (*beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, **args*, ***kwargs*)
Adamax updates

Adamax updates implemented as in [1]. This is a variant of the Adam algorithm based on the infinity norm.

Parameters

- beta1** [float] Exponential decay rate for the first moment estimates.
- beta2** [float] Exponential decay rate for the second moment estimates.
- epsilon** [float] Constant for numerical stability.

References

[1]

update (*params*, *grads*)
Update parameters.

Parameters

- params** [list] A list of parameters in model.
- grads** [list] A list of gradients in model.

1.6 `numpydl.model`

Linear stack of layers.

1.6.1 Detailed Description

`class npdl.model.Model (layers=None)`

`predict (X)`

Calculate an output Y for the given input X.

1.7 npdl.utils

1.7.1 Data Utils

`npdl.utils.one_hot (labels, nb_classes=None)`

One-hot encoding is often used for indicating the state of a state machine. When using binary or Gray code, a decoder is needed to determine the state. A one-hot state machine, however, does not need a decoder as the state machine is in the `n`th state if and only if the `n`th bit is high.

A ring counter with 15 sequentially-ordered states is an example of a state machine. A `one-hot` implementation would have 15 flip flops chained in series with the Q output of each flip flop connected to the D input of the next and the D input of the first flip flop connected to the Q output of the 15th flip flop. The first flip flop in the chain represents the first state, the second represents the second state, and so on to the 15th flip flop which represents the last state. Upon reset of the state machine all of the flip flops are reset to 0 except the first in the chain which is set to 1. The next clock edge arriving at the flip flops advances the `one_hot` bit to the second flip flop. The `hot` bit advances in this way until the 15th state, after which the state machine returns to the first state.

An address decoder converts from binary or gray code to one-hot representation. A priority encoder converts from one-hot representation to binary or gray code.

In natural language processing, a one-hot vector is a $1N$ matrix (vector) used to distinguish each word in a vocabulary from every other word in the vocabulary. The vector consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the word.

Parameters

`labels` iterable

`nb_classes` [(iterable, optional)]

Returns

`numpy.array` Returns a one-hot numpy array.

`npdl.utils.unhot (one_hot_labels)`

Get argmax indexes.

Parameters

`one_hot_labels` [numpy.array]

Returns

`numpy.array` Returns a unhot numpy array.

1.7.2 Random Utils

`npdl.utils.get_rng ()`

Get the package-level random number generator.

Returns

numpy.random.RandomState instance The `numpy.random.RandomState` instance passed to the most recent call of `set_rng()`, or `numpy.random` if `set_rng()` has never been called.

`npdl.utils.set_rng(rng)`

Set the package-level random number generator.

Parameters

new_rng [`numpy.random` or a `numpy.random.RandomState` instance] The random number generator to use.

`npdl.utils.set_seed(seed)`

Set numpy seed.

Parameters

seed [int]

`npdl.utils.get_dtype()`

Get data dtype `numpy.dtype`.

Returns

str or `numpy.dtype`

`npdl.utils.set_dtype(dtype)`

Set numpy dtype.

Parameters

dtype [str or `numpy.dtype`]

Data Utils

<code>one_hot</code>	One-hot encoding is often used for indicating the state of a state machine.
<code>unhot</code>	Get argmax indexes.

Random Utils

<code>get_rng</code>	Get the package-level random number generator.
<code>set_rng</code>	Set the package-level random number generator.
<code>set_seed</code>	Set numpy seed.
<code>get_dtype</code>	Get data dtype <code>numpy.dtype</code> .
<code>set_dtype</code>	Set numpy dtype.

CHAPTER 2

Indices and tables

- `genindex`
- `modindex`
- `search`

Bibliography

- [1] [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](<https://arxiv.org/abs/1502.03167>)
- [1] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, J. Schmidhuber. A Novel Connectionist System for Improved Unconstrained Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, 2009.
- [2] H. Sak and A. W. Senior and F. Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. *Proc. Interspeech*, pp338-342, Singapore, Sept. 2010
- [1] A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. <http://arxiv.org/abs/1512.05287>
- [1] Chung, Junyoung; Gulcehre, Caglar; Cho, KyungHyun; Bengio, Yoshua (2014). "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". arXiv:1412.3555Freely accessible [cs.NE].
- [2] "Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano – WildML". Wildml.com. Retrieved May 18, 2016.
- [1] Sepp Hochreiter; Jürgen Schmidhuber (1997). "Long short-term memory". *Neural Computation*. 9 (8): 1735–1780. doi:10.1162/ne co.1997.9.8.1735. PMID 9377276.
- [2] Felix A. Gers; Jürgen Schmidhuber; Fred Cummins (2000). "Learning to Forget: Continual Prediction with LSTM". *Neural Computation*. 12 (10): 2451–2471. doi:10.1162/089976600300015015.
- [1] Sepp Hochreiter; Jürgen Schmidhuber (1997). "Long short-term memory". *Neural Computation*. 9 (8): 1735–1780. doi:10.1162/ne co.1997.9.8.1735. PMID 9377276.
- [2] Felix A. Gers; Jürgen Schmidhuber; Fred Cummins (2000). "Learning to Forget: Continual Prediction with LSTM". *Neural Computation*. 12 (10): 2451–2471. doi:10.1162/089976600300015015.
- [1] Lehmann, E. L.; Casella, George (1998). *Theory of Point Estimation* (2nd ed.). New York: Springer. ISBN 0-387-98502-6. MR 1639875.
- [1] Nikulin, M.S. (2001), "Hellinger distance", in Hazewinkel, Michiel, *Encyclopedia of Mathematics*, Springer, ISBN 978-1-55608-010-4
- [2] Jump up ^ Hellinger, Ernst (1909), "Neue Begründung der Theorie quadratischer Formen von unendlichvielen Veränderlichen", *Journal für die reine und angewandte Mathematik* (in German), 136: 210–271, doi:10.1515/crll.1909.136.210, JFM 40.0393.01
- [1] Duchi, J., Hazan, E., & Singer, Y. (2011): Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121-2159.

- [2] Chris Dyer: Notes on AdaGrad. <http://www.ark.cs.cmu.edu/cdyer/adagrad.pdf>
- [1] Tieleman, T. and Hinton, G. (2012): Neural Networks for Machine Learning, Lecture 6.5 - rmsprop. Coursera. <http://www.youtube.com/watch?v=O3sxAc4hxZU> (formula @5:20)
- [1] Zeiler, M. D. (2012): ADADELTA: An Adaptive Learning Rate Method. arXiv Preprint arXiv:1212.5701.
- [1] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.
- [1] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

n

`npdl.activations`, 16
`npdl.layers`, 3
`npdl.layers.base`, 3
`npdl.layers.convolution`, 6
`npdl.layers.core`, 4
`npdl.layers.embedding`, 7
`npdl.layers.normalization`, 8
`npdl.layers.pooling`, 9
`npdl.layers.recurrent`, 9
`npdl.layers.shape`, 14
`npdl.model`, 36
`npdl.objectives`, 28
`npdl.optimizers`, 32
`npdl.utils`, 37
`npdl.utils.data`, 37
`npdl.utils.random`, 37

A

Activation (class in *npdl.activations*), 16
 Adadelta (class in *npdl.optimizers*), 35
 Adagrad (class in *npdl.optimizers*), 33
 Adam (class in *npdl.optimizers*), 35
 Adamax (class in *npdl.optimizers*), 36

B

backward() (*npdl.layers.BatchLSTM* method), 13
 backward() (*npdl.layers.BatchNormal* method), 8
 backward() (*npdl.layers.Convolution* method), 6
 backward() (*npdl.layers.Dense* method), 5
 backward() (*npdl.layers.Dropout* method), 6
 backward() (*npdl.layers.Embedding* method), 7
 backward() (*npdl.layers.Flatten* method), 14
 backward() (*npdl.layers.GRU* method), 11
 backward() (*npdl.layers.Layer* method), 3
 backward() (*npdl.layers.Linear* method), 4
 backward() (*npdl.layers.LSTM* method), 12
 backward() (*npdl.layers.MaxPooling* method), 9
 backward() (*npdl.layers.MeanPooling* method), 9
 backward() (*npdl.layers.SimpleRNN* method), 10
 backward() (*npdl.objectives.BinaryCrossEntropy* method), 30
 backward() (*npdl.objectives.HellingerDistance* method), 30
 backward() (*npdl.objectives.MeanSquaredError* method), 29
 backward() (*npdl.objectives.Objective* method), 28
 backward() (*npdl.objectives.SoftmaxCategoricalCrossEntropy* method), 31
 BatchLSTM (class in *npdl.layers*), 13
 BatchNormal (class in *npdl.layers*), 8
 BCE (in module *npdl.objectives*), 31
 BinaryCrossEntropy (class in *npdl.objectives*), 30

C

connect_to() (*npdl.layers.BatchLSTM* method), 13
 connect_to() (*npdl.layers.BatchNormal* method), 8

connect_to() (*npdl.layers.Convolution* method), 6
 connect_to() (*npdl.layers.Dense* method), 5
 connect_to() (*npdl.layers.Dropout* method), 6
 connect_to() (*npdl.layers.Embedding* method), 7
 connect_to() (*npdl.layers.Flatten* method), 14
 connect_to() (*npdl.layers.GRU* method), 11
 connect_to() (*npdl.layers.Layer* method), 3
 connect_to() (*npdl.layers.Linear* method), 4
 connect_to() (*npdl.layers.LSTM* method), 12
 connect_to() (*npdl.layers.MaxPooling* method), 9
 connect_to() (*npdl.layers.MeanPooling* method), 9
 connect_to() (*npdl.layers.Recurrent* method), 10
 connect_to() (*npdl.layers.SimpleRNN* method), 10
 Convolution (class in *npdl.layers*), 6

D

Dense (class in *npdl.layers*), 5
 derivative() (*npdl.activations.Activation* method), 16
 derivative() (*npdl.activations.Elliot* method), 23
 derivative() (*npdl.activations.Linear* method), 21
 derivative() (*npdl.activations.ReLU* method), 19
 derivative() (*npdl.activations.Sigmoid* method), 16
 derivative() (*npdl.activations.Softmax* method), 22
 derivative() (*npdl.activations.SoftPlus* method), 25
 derivative() (*npdl.activations.SoftSign* method), 26
 derivative() (*npdl.activations.SymmetricElliot* method), 24
 derivative() (*npdl.activations.Tanh* method), 18
 Dropout (class in *npdl.layers*), 6

E

Elliot (class in *npdl.activations*), 23
 Embedding (class in *npdl.layers*), 7

F

Flatten (class in *npdl.layers*), 14
 forward() (*npdl.activations.Activation* method), 16
 forward() (*npdl.activations.Elliot* method), 23

forward() (*npdl.activations.Linear method*), 21
 forward() (*npdl.activations.ReLU method*), 20
 forward() (*npdl.activations.Sigmoid method*), 16
 forward() (*npdl.activations.Softmax method*), 22
 forward() (*npdl.activations.SoftPlus method*), 25
 forward() (*npdl.activations.SoftSign method*), 26
 forward() (*npdl.activations.SymmetricElliot method*),
 24
 forward() (*npdl.activations.Tanh method*), 18
 forward() (*npdl.layers.BatchLSTM method*), 13
 forward() (*npdl.layers.BatchNormal method*), 8
 forward() (*npdl.layers.Convolution method*), 7
 forward() (*npdl.layers.Dense method*), 5
 forward() (*npdl.layers.Dropout method*), 6
 forward() (*npdl.layers.Embedding method*), 7
 forward() (*npdl.layers.Flatten method*), 14
 forward() (*npdl.layers.GRU method*), 11
 forward() (*npdl.layers.Layer method*), 3
 forward() (*npdl.layers.Linear method*), 4
 forward() (*npdl.layers.LSTM method*), 12
 forward() (*npdl.layers.MaxPooling method*), 9
 forward() (*npdl.layers.MeanPooling method*), 9
 forward() (*npdl.layers.SimpleRNN method*), 11
 forward() (*npdl.objectives.BinaryCrossEntropy
 method*), 31
 forward() (*npdl.objectives.HellingerDistance
 method*), 30
 forward() (*npdl.objectives.MeanSquaredError
 method*), 29
 forward() (*npdl.objectives.Objective method*), 29
 forward() (*npdl.objectives.SoftmaxCategoricalCrossEntropy
 method*), 31
 from_json() (*npdl.layers.Layer class method*), 3

G

get_dtype() (*in module npdl.utils*), 38
 get_rng() (*in module npdl.utils*), 37
 grads (*npdl.layers.BatchLSTM attribute*), 14
 grads (*npdl.layers.Convolution attribute*), 7
 grads (*npdl.layers.Dense attribute*), 5
 grads (*npdl.layers.Embedding attribute*), 7
 grads (*npdl.layers.GRU attribute*), 12
 grads (*npdl.layers.Layer attribute*), 3
 grads (*npdl.layers.Linear attribute*), 4
 grads (*npdl.layers.LSTM attribute*), 13
 grads (*npdl.layers.SimpleRNN attribute*), 11
 GRU (*class in npdl.layers*), 11

H

HeD (*in module npdl.objectives*), 30
 HellingerDistance (*class in npdl.objectives*), 30

L

Layer (*class in npdl.layers*), 3

Linear (*class in npdl.activations*), 21
 Linear (*class in npdl.layers*), 4
 LSTM (*class in npdl.layers*), 12

M

MaxPooling (*class in npdl.layers*), 9
 MeanPooling (*class in npdl.layers*), 9
 MeanSquaredError (*class in npdl.objectives*), 29
 Model (*class in npdl.model*), 37
 Momentum (*class in npdl.optimizers*), 32
 MSE (*in module npdl.objectives*), 30

N

NesterovMomentum (*class in npdl.optimizers*), 33
 npdl.activations (*module*), 16
 npdl.layers (*module*), 3
 npdl.layers.base (*module*), 3
 npdl.layers.convolution (*module*), 6
 npdl.layers.core (*module*), 4
 npdl.layers.embedding (*module*), 7
 npdl.layers.normalization (*module*), 8
 npdl.layers.pooling (*module*), 9
 npdl.layers.recurrent (*module*), 9
 npdl.layers.shape (*module*), 14
 npdl.model (*module*), 36
 npdl.objectives (*module*), 28
 npdl.optimizers (*module*), 32
 npdl.utils (*module*), 37
 npdl.utils.data (*module*), 37
 npdl.utils.random (*module*), 37

O

Objective (*class in npdl.objectives*), 28
 one_hot() (*in module npdl.utils*), 37

P

param_grads (*npdl.layers.Layer attribute*), 3
 params (*npdl.layers.BatchLSTM attribute*), 14
 params (*npdl.layers.BatchNormal attribute*), 8
 params (*npdl.layers.Convolution attribute*), 7
 params (*npdl.layers.Dense attribute*), 5
 params (*npdl.layers.Embedding attribute*), 7
 params (*npdl.layers.GRU attribute*), 12
 params (*npdl.layers.Layer attribute*), 3
 params (*npdl.layers.Linear attribute*), 4
 params (*npdl.layers.LSTM attribute*), 13
 params (*npdl.layers.SimpleRNN attribute*), 11
 predict() (*npdl.model.Model method*), 37

R

Recurrent (*class in npdl.layers*), 9
 ReLU (*class in npdl.activations*), 19
 RMSprop (*class in npdl.optimizers*), 34

S

SCCE (*in module npdl.objectives*), 31
set_dtype () (*in module npdl.utils*), 38
set_rng () (*in module npdl.utils*), 38
set_seed () (*in module npdl.utils*), 38
SGD (*class in npdl.optimizers*), 32
Sigmoid (*class in npdl.activations*), 16
SimpleRNN (*class in npdl.layers*), 10
Softmax (*class in npdl.activations*), 22
Softmax (*class in npdl.layers*), 6
SoftmaxCategoricalCrossEntropy (*class in npdl.objectives*), 31
SoftPlus (*class in npdl.activations*), 25
SoftSign (*class in npdl.activations*), 26
SymmetricElliot (*class in npdl.activations*), 24

T

Tanh (*class in npdl.activations*), 17
to_json () (*npdl.layers.Layer method*), 4

U

unhot () (*in module npdl.utils*), 37
update () (*npdl.optimizers.Adadelta method*), 35
update () (*npdl.optimizers.Adagrad method*), 34
update () (*npdl.optimizers.Adam method*), 36
update () (*npdl.optimizers.Adamax method*), 36
update () (*npdl.optimizers.Momentum method*), 33
update () (*npdl.optimizers.NesterovMomentum method*), 33
update () (*npdl.optimizers.RMSprop method*), 35
update () (*npdl.optimizers.SGD method*), 32