NumPyNet - Neural Networks Library in pure Numpy

Release 1.0.0

Nico Curti, Mattia Ceccarelli

Mar 28, 2022

CONTENTS:

Implementation in **pure** *Numpy* of neural networks models. *NumPyNet* supports a syntax very close to the *Keras* one but it is written using **only** *Numpy* functions: in this way it is very light and fast to install and use/modify.

CHAPTER

ONE

OVERVIEW

NumPyNet is born as educational framework for the study of Neural Network models. It is written trying to balance code readability and computational performances and it is enriched with a large documentation to better understand the functionality of each script. The library is written in pure *Python* and the only external library used is Numpy (a base package for the scientific research).

Despite all common libraries are correlated by a wide documentation is often difficult for novel users to move around the many hyper-links and papers cited in them. *NumPyNet* tries to overcome this problem with a minimal mathematical documentation associated to each script and a wide range of comments inside the code.

An other "problem" to take in count is related to performances. Libraries like Tensorflow are certainly efficient from a computational point-of-view and the numerous wrappers (like *Keras* library) guarantee an extremely simple user interface. On the other hand, the deeper functionalities of the code and the implementation strategies used are unavoidably hidden behind tons of code lines. In this way the user can perform complex computational tasks using the library as black-box package. *NumPyNet* wants to overcome this problem using simple *Python* codes, with extremely readability also for novel users, to better understand the symmetry between mathematical formulas and code.

CHAPTER

TWO

USAGE EXAMPLE

First of all we have to import the main modules of the *NumPyNet* package as

```
from NumPyNet.network import Network
from NumPyNet.layers.connected_layer import Connected_layer
from NumPyNet.layers.convolutional_layer import Convolutional_layer
from NumPyNet.layers.maxpool_layer import Maxpool_layer
from NumPyNet.layers.softmax_layer import Softmax_layer
from NumPyNet.layers.batchnorm_layer import BatchNorm_layer
from NumPyNet.optimizer import Adam
```

Now we can try to create a very simple model able to classify the well known MNIST-digit dataset. The MNIST dataset can be extracted from the *sklearn* library as

```
from sklearn import datasets
from sklearn.model_selection import train_test_split

digits = datasets.load_digits()
X, y = digits.images, digits.target

X = np.asarray([np.dstack((x, x, x)) for x in X])
X = X.transpose(0, 2, 3, 1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=42)
```

Now we have to create our model. We can use a syntax very close to the *Keras* one and simply define a model object adding a series of layers

```
model = Network(batch=batch, input_shape=X_train.shape[1:])

model.add(Convolutional_layer(size=3, filters=32, stride=1, pad=True, activation='Relu'))
model.add(BatchNorm_layer())
model.add(Maxpool_layer(size=2, stride=1, padding=True))
model.add(Connected_layer(outputs=100, activation='Relu'))
model.add(BatchNorm_layer())
model.add(Connected_layer(outputs=num_classes, activation='Linear'))
model.add(Softmax_layer(spatial=True, groups=1, temperature=1.))

model.compile(optimizer=Adam(), metrics=[accuracy])

model.summary()
```

The model automatically creates an *InputLayer* if it is not explicitly provided, but pay attention to the right *input_shape* values!

Before feeding our model we have to convert the image dataset into categorical variables. To this purpose we can use the simple *utilities* of the *NumPyNet* package.

```
from NumPyNet.utils import to_categorical
from NumPyNet.utils import from_categorical
from NumPyNet.metrics import mean_accuracy_score

# normalization to [0, 1]
X_train *= 1. / 255.
X_test  *= 1. / 255.

n_train = X_train.shape[0]
n_test  = X_test.shape[0]

# transform y to array of dimension 10 and in 4 dimension
y_train = to_categorical(y_train).reshape(n_train, 1, 1, -1)
y_test  = to_categorical(y_test).reshape(n_test, 1, 1, -1)
```

Now you can run your *fit* function to train the model as

```
model.fit(X=X_train, y=y_train, max_iter=100)
```

and evaluate the results on the testing set

```
loss, out = model.evaluate(X=X_test, truth=y_test, verbose=True)

truth = from_categorical(y_test)
predicted = from_categorical(out)
accuracy  = mean_accuracy_score(truth, predicted)

print('\nLoss Score: {:.3f}'.format(loss))
print('Accuracy Score: {:.3f}'.format(accuracy))
```

You should see something like this

```
layer       filters  size              input                output
   0 input                     128 x   8 x   3 x   8   ->   128 x   8 x   3 x   8
   1 conv     32 3 x 3 / 1   128 x   8 x   3 x   8   ->   128 x   8 x   3 x  32  0.000␣
↪BFLOPs
   2 batchnorm                         8 x   3 x  32 image
   3 max         2 x 2 / 1   128 x   8 x   3 x  32   ->   128 x   7 x   2 x  32
   4 connected               128 x   7 x   2 x  32   ->   128 x 100
   5 batchnorm                         1 x   1 x 100 image
   6 connected               128 x   1 x   1 x 100   ->   128 x  10
   7 softmax x entropy                                    128 x   1 x   1 x  10

Epoch 1/10
512/512 || (0.7 sec/iter) loss: 26.676 accuracy: 0.826

Epoch 2/10
512/512 || (0.6 sec/iter) loss: 22.547 accuracy: 0.914
```

(continues on next page)

(continued from previous page)

```
Epoch 3/10
512/512 || (0.7 sec/iter) loss: 21.333 accuracy: 0.943

Epoch 4/10
512/512 || (0.6 sec/iter) loss: 20.832 accuracy: 0.963

Epoch 5/10
512/512 || (0.5 sec/iter) loss: 20.529 accuracy: 0.975

Epoch 6/10
512/512 || (0.3 sec/iter) loss: 20.322 accuracy: 0.977

Epoch 7/10
512/512 || (0.3 sec/iter) loss: 20.164 accuracy: 0.986

Epoch 8/10
512/512 || (0.3 sec/iter) loss: 20.050 accuracy: 0.992

Epoch 9/10
512/512 || (0.3 sec/iter) loss: 19.955 accuracy: 0.994

Epoch 10/10
512/512 || (0.3 sec/iter) loss: 19.875 accuracy: 0.996

Training on 10 epochs took 21.6 sec

300/300 || (0.0 sec/iter) loss: 10.472
Prediction on 300 samples took 0.1 sec

Loss Score: 2.610
Accuracy Score: 0.937
```

Obviously the execution time can vary according to your available resources! You can find a full list of example scripts here

2.1 NumPyNet API

2.1.1 NumPyNet layers

Activation layer

class layers.activation_layer.**Activation_layer**(*input_shape=None*, *activation=<class 'NumPyNet.activations.Activations'>*, ***kwargs*)

Bases: `NumPyNet.layers.base.BaseLayer`

Activation layer

Parameters

- **input_shape** (*tuple (default=None)*) – Input dimensions as tuple of 4 integers
- **activation** (*str or Activation object*) – Activation function to apply into the layer.

Example

```
>>> import os
>>> import pylab as plt
>>> from PIL import Image
>>> from NumPyNet import activations
>>>
>>> activation_func = activations.Relu()
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 255.
↪)).astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data', 'dog.jpg
↪')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>> # Relu activation constrain
>>> inpt = inpt * 2 - 1
>>>
>>> # add batch = 1
>>> inpt = np.expand_dims(inpt, axis=0)
>>>
>>> layer = Activation_layer(input_shape=inpt.shape, activation=activation_func)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt)
>>> forward_out = layer.output
>>> print(layer)
>>>
>>> # BACKWARD
>>>
>>> layer.delta = np.ones(shape=inpt.shape, dtype=float)
>>> delta = np.zeros(shape=inpt.shape, dtype=float)
>>> layer.backward(delta, copy=True)
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>>
>>> fig.suptitle('Activation Layer : {}'.format(activation_func.name))
>>>
>>> ax1.imshow(float_2_img(inpt[0]))
>>> ax1.set_title('Original image')
>>> ax1.axis('off')
>>>
>>> ax2.imshow(float_2_img(forward_out[0]))
>>> ax2.set_title("Forward")
>>> ax2.axis("off")
```

(continues on next page)

(continued from previous page)

```
>>>
>>>  ax3.imshow(float_2_img(delta[0]))
>>>  ax3.set_title('Backward')
>>>  ax3.axis('off')
>>>
>>>  fig.tight_layout()
>>>  plt.show()
```

References

- TODO

backward(*delta*, *copy=False*)

Compute the backward of the activation layer

Parameters **delta** (*array-like*) – Global error to be backpropagated.

Return type self

forward(*inpt*, *copy=True*)

Forward of the activation layer, apply the selected activation function to the input.

Parameters

- **inpt** (*array-like*) – Input array to activate.
- **copy** (*bool (default=True)*) – If True make a copy of the input before applying the activation.

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

Batchnorm layer

class layers.batchnorm_layer.**BatchNorm_layer**(*scales=None*, *bias=None*, *input_shape=None*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

BatchNormalization Layer

It performs a Normalization over the Batch axis of the Input. Both scales and bias are trainable weights.

Equation:

output = scales * input_normalized + bias

Parameters

- **scales** (*array-like (default=None)*) – Starting scale to be multiplied to the normalized input, array-like of shape (w, h, c). If None, the array will be initialized with ones.
- **bias** (*array-like (default=None)*) – Bias to be added to the multiplication of scale and normalized input of shape (w, h, c). If None, the array will be initialized with zeros.
- **input_shape** (*tuple (default=None)*) – Shape of the input in the format (batch, w, h, c), None is used when the layer is part of a Network model.

Example

```
>>> import os
>>>
>>> import pylab as plt
>>> from PIL import Image
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 255.)).
↪astype(np.uint8)
>>>
>>> # I need to load at least to images, or made a copy of it
>>> filename = os.path.join(os.path.dirname('__file__'), '..', '..', 'data', 'dog.
↪jpg')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> w, h, c = inpt.shape
>>>
>>> batch_size = 5
>>>
>>> np.random.seed(123) # set seed to have fixed bias and scales
>>>
>>> # create a pseudo-input with batch_size images with a random offset from the␣
↪original image
>>> rng  = np.random.uniform(low=0., high=100., size=(batch_size, w, h, c))
>>> inpt = np.concatenate([np.expand_dims(inpt, axis=0) + r for r in rng], axis=0)
↪# create a set of image
>>>
>>> # img_to_float of input, to work with numbers btween 0. and 1.
>>> inpt = np.asarray([img_2_float(x) for x in inpt ])
```

(continues on next page)

(continued from previous page)

```
>>>
>>> b, w, h, c = inpt.shape # needed for initializations of bias and scales
>>>
>>> bias   = np.random.uniform(0., 1., size=(w, h, c)) # random biases
>>> scales = np.random.uniform(0., 1., size=(w, h, c)) # random scales
>>>
>>> bias = np.zeros(shape=(w, h, c), dtype=float)
>>> scales = np.ones(shape=(w, h, c), dtype=float)
>>>
>>> # Model Initialization
>>> layer = BatchNorm_layer(input_shape=inpt.shape, scales=scales, bias=bias)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt)
>>> forward_out = layer.output
>>> print(layer)
>>>
>>> # BACKWARD
>>>
>>> layer.delta = np.random.uniform(low=0., high=100., size=layer.out_shape)
>>> delta = np.ones(shape=inpt.shape, dtype=float) # delta same shape as the Input
>>> layer.backward(delta)
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=2, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>>
>>> fig.suptitle('BatchNormalization Layer')
>>>
>>> ax1[0].imshow(float_2_img(inpt[0]))
>>> ax1[0].set_title('Original image')
>>> ax1[0].axis('off')
>>>
>>> ax1[1].imshow(float_2_img(layer.mean))
>>> ax1[1].set_title("Mean Image")
>>> ax1[1].axis("off")
>>>
>>> ax2[0].imshow(float_2_img(forward_out[0]))
>>> ax2[0].set_title('Forward')
>>> ax2[0].axis('off')
>>>
>>> ax2[1].imshow(float_2_img(delta[0]))
>>> ax2[1].set_title('Backward')
>>> ax2[1].axis('off')
>>>
>>> fig.tight_layout()
>>> plt.show()
```

References

- https://arxiv.org/abs/1502.03167

backward(*delta=None*)

BackPropagation function of the BatchNormalization layer. Every formula is a derivative computed by chain rules: dbeta = derivative of output w.r.t. bias, dgamma = derivative of output w.r.t. scales etc...

Parameters **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.

Return type self

epsil = 1e-08

forward(*inpt*)

Forward function of the BatchNormalization layer. It computes the output of the layer, the formula is:

output = scale * input_norm + bias

Where input_norm is:

input_norm = (input - mean) / sqrt(var + epsil)

where mean and var are the mean and the variance of the input batch of images computed over the first axis (batch)

Parameters **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c)

Return type self

load_weights(*chunck_weights*, *pos=0*)

Load weights from full array of model weights

Parameters

- **chunck_weights** (*array-like*) – Model weights and bias
- **pos** (*int (default=0)*) – Current position of the array

Returns **pos** – Updated stream position.

Return type int

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

save_weights()

Return the biases and weights in a single ravel fmt to save in binary file

update()

Update function for the batch-normalization layer. Optimizer must be assigned externally as an optimizer object.

Return type self

Connected layer

class layers.connected_layer.**Connected_layer**(*outputs*, *activation=<class 'NumPyNet.activations.Activations'>*, *input_shape=None*, *weights=None*, *bias=None*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

Connected layer

It's the equivalent of a Dense layer in keras, or a single layer of an MLP in scikit-learn

Parameters

- **outputs** (*int*) – Number of outputs of the layers. It's also the number of Neurons of the layer.
- **activation** (*str or Activation object*) – Activation function of the layer.
- **input_shape** (*tuple (default=None)*) – Shape of the input in the format (batch, w, h, c), None is used when the layer is part of a Network model.
- **weights** (*array-like (default=None)*) – Array of shape (w * h * c, outputs), default is None. Weights of the dense layer. If None, weights initialization is random and follows a uniform distribution in the range [-scale, scale] where:

  scale = sqrt(2 / (w * h* c))

- **bias** (*array-like (default=None)*) – Array of shape (outputs,). Bias of the fully-connected layer. If None, bias inititialization is zeros

Example

```
>>> import os
>>>
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from PIL import Image
>>>
>>> from NumPyNet import activations
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 255.)).
↪astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data', 'dog.jpg
↪')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>>
>>> # from (w, h, c) to shape (1, w, h, c)
>>> inpt = np.expand_dims(inpt, axis=0)  # just to add the 'batch' dimension
>>>
>>> # Number of outputs
>>> outputs = 10
```

(continues on next page)

(continued from previous page)

```
>>> layer_activation = activations.Relu()
>>> batch, w, h, c = inpt.shape
>>>
>>> # Random initialization of weights with shape (w * h * c) and bias with shape␣
↪(outputs,)
>>> np.random.seed(123)  # only if one want always the same set of weights
>>> weights = np.random.uniform(low=-1., high=1., size=(np.prod(inpt.shape[1:]),␣
↪outputs))
>>> bias = np.random.uniform(low=-1., high=1., size=(outputs,))
>>>
>>> # Model initialization
>>> layer = Connected_layer(outputs, input_shape=inpt.shape,
>>>                         activation=layer_activation, weights=weights, bias=bias)
>>> print(layer)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt)
>>> forward_out = layer.output.copy()
>>>
>>> # BACKWARD
>>>
>>> layer.delta = np.ones(shape=(layer.out_shape), dtype=float)
>>> delta = np.zeros(shape=(batch, w, h, c), dtype=float)
>>> layer.backward(inpt, delta=delta, copy=True)
>>>
>>> # print('Output: {}'.format(', '.join( ['{:.3f}'.format(x) for x in forward_out[0]]␣
↪) ) )
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>> fig.suptitle('Connected Layer activation : {}'.format(layer_activation.name))
>>>
>>> ax1.imshow(float_2_img(inpt[0]))
>>> ax1.set_title('Original Image')
>>> ax1.axis('off')
>>>
>>> ax2.matshow(forward_out[:, 0, 0, :], cmap='bwr')
>>> ax2.set_title('Forward', y=4)
>>> ax2.axes.get_yaxis().set_visible(False)         # no y axis tick
>>> ax2.axes.get_xaxis().set_ticks(range(outputs))  # set x axis tick for every␣
↪output
>>>
>>> ax3.imshow(float_2_img(delta[0]))
>>> ax3.set_title('Backward')
>>> ax3.axis('off')
>>>
>>> fig.tight_layout()
>>> plt.show()
```

References

- TODO

backward(*inpt*, *delta=None*, *copy=False*)

Backward function of the connected layer, updates the global delta of the network to be Backpropagated, he weights upadtes and the biases updates

Parameters

- **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.
- **copy** (*bool (default=False)*) – States if the activation function have to return a copy of the input or not.

Return type self

forward(*inpt*, *copy=False*)

Forward function of the connected layer. It computes the matrix product between inpt and weights, add bias and activate the result with the chosen activation function.

Parameters

- **inpt** (*array-like*) – Input batch in format (batch, in_w, in_h, in _c)
- **copy** (*bool (default=False)*) – If False the activation function modifies its input, if True make a copy instead

Return type self

property inputs

Number of inputs of the layer, not considering the batch size

load_weights(*chunck_weights*, *pos=0*)

Load weights from full array of model weights

Parameters

- **chunck_weights** (*array-like*) – model weights and bias
- **pos** (*int (default=0)*) – Current position of the array

Returns **pos** – Updated stream position.

Return type int

property out_shape

Returns the output shape in the format (batch, 1, 1, outputs)

save_weights()

Return the biases and weights in a single ravel fmt to save in binary file

update()

Update function for the convolution layer. Optimizer must be assigned externally as an optimizer object.

Return type self

Convolutional layer

class layers.convolutional_layer.**Convolutional_layer**(*filters*, *size*, *stride=None*, *input_shape=None*, *weights=None*, *bias=None*, *pad=False*, *activation=<class 'NumPyNet.activations.Activations'>*, ***kwargs*)

Bases: `NumPyNet.layers.base.BaseLayer`

Convolutional Layer

Parameters

- **filters** (*int*) – Number of filters to be slided over the input, and also the number of channels of the output (channels_out)
- **size** (*tuple*) – Size of the kernel of shape (kx, ky).
- **stride** (*tuple (default=None)*) – Step of the kernel, with shape (st1, st2). If None, stride is assigned size values.
- **input_shape** (*tuple (default=None)*) – Shape of the input in the format (batch, w, h, c), None is used when the layer is part of a Network model.
- **weights** (*array-like (default=None)*) – Filters of the convolutionanl layer, with shape (kx, ky, channels_in, filters). If None, random weights are initialized
- **bias** (*array-like (default=None)*) – Bias of the convolutional layer. If None, bias init is random with shape (filters,)
- **pad** (*bool (default=False)*) – If False the image is cutted along the last raws and columns, if True the input is padded following keras SAME padding
- **activation** (*str or Activation object*) – Activation function of the layer.

Example

```
>>> import os
>>> from PIL import Image
>>> import pylab as plt
>>> from NumPyNet import activations
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 255.
↪)).astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data', 'dog.jpg
↪')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>> # Relu activation constrain
>>> inpt = inpt * 2 - 1
>>>
>>> inpt = np.expand_dims(inpt, axis=0) # shape from (w, h, c) to (1, w, h, c)
>>>
```

(continues on next page)

(continued from previous page)

```
>>>  channels_out = 10
>>>  size         = (3, 3)
>>>  stride       = (1, 1)
>>>  pad          = False
>>>
>>>  layer_activation = activations.Relu()
>>>
>>>  np.random.seed(123)
>>>
>>>  b, w, h, c = inpt.shape
>>>  filters    = np.random.uniform(-1., 1., size = (size[0], size[1], c, channels_
↪out))
>>>  # bias       = np.random.uniform(-1., 1., size = (channels_out,))
>>>  bias = np.zeros(shape=(channels_out,))
>>>
>>>  layer = Convolutional_layer(input_shape=inpt.shape,
>>>                              filters=channels_out,
>>>                              weights=filters,
>>>                              bias=bias,
>>>                              activation=layer_activation,
>>>                              size=size,
>>>                              stride=stride,
>>>                              pad=pad)
>>>
>>>  # FORWARD
>>>
>>>  layer.forward(inpt)
>>>  forward_out = layer.output.copy()
>>>
>>>  # after the forward to load all the attribute
>>>  print(layer)
>>>
>>>  # BACKWARD
>>>
>>>  layer.delta = np.ones(layer.out_shape, dtype=float)
>>>  delta = np.zeros(shape=inpt.shape, dtype=float)
>>>  layer.backward(delta)
>>>
>>>  # layer.update()
>>>
>>>  # Visualization
>>>
>>>  fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>>  fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>>
>>>  fig.suptitle('Convolutional Layer')
>>>
>>>  ax1.imshow(float_2_img(inpt[0]))
>>>  ax1.set_title('Original image')
>>>  ax1.axis('off')
>>>  # here every filter effect on the image can be shown
>>>  ax2.imshow(float_2_img(forward_out[0, :, :, 1]))
```

(continues on next page)

(continued from previous page)

```
>>>  ax2.set_title('Forward')
>>>  ax2.axis('off')
>>>
>>>  ax3.imshow(float_2_img(delta[0]))
>>>  ax3.set_title('Backward')
>>>  ax3.axis('off')
>>>
>>>  fig.tight_layout()
>>>  plt.show()
```

References

- https://arxiv.org/abs/1603.07285
- https://cs231n.github.io/convolutional-networks/
- https://stackoverflow.com/questions/42463172/how-to-perform-max-mean-pooling-on-a-2d-array-using-numpy
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.lib.stride_tricks.as_strided.html
- https://stackoverflow.com/questions/53819528/how-does-tf-keras-layers-conv2d-with-padding-same-and-strides-1-behave

backward(*delta*, *copy=False*)

Backward function of the Convolutional layer. Source: https://arxiv.org/abs/1603.07285

Parameters

- **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.
- **copy** (*bool (default=False)*) – States if the activation function have to return a copy of the input or not.

Return type self

forward(*inpt*, *copy=False*)

Forward function of the Convolutional Layer: it convolves an image with 'channels_out' filters with dimension (kx, ky, channels_in). In doing so, it creates a view of the image with shape (batch, out_w, out_h, in_c, kx, ky) in order to perform a single matrix multiplication with the reshaped filters array, which shape is (in_c * kx * ky, out_c).

Parameters

- **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c)
- **copy** (*bool (default=False)*) – If False the activation function modifies its input, if True make a copy instead

Return type self

load_weights(*chunck_weights*, *pos=0*)

Load weights from full array of model weights

Parameters

- **chunck_weights** (*array-like*) – model weights and bias
- **pos** (*int (default=0)*) – Current position of the array

Returns **pos** – Updated stream position.

Return type int

property out_shape

Get the output shape as (batch, out_w, out_h, out_channels)

save_weights()

Return the biases and weights in a single ravel fmt to save in binary file

update()

Update function for the convolution layer. Optimizer must be assigned externally as an optimizer object.

Return type self

Cost layer

class layers.cost_layer.**Cost_layer**(*cost_type*, *input_shape=None*, *scale=1.0*, *ratio=0.0*, *noobject_scale=1.0*, *threshold=0.0*, *smoothing=0.0*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

Cost layer

Compute the cost of the output based on the selected cost function.

Parameters

- **input_shape** – tuple (default=None) Shape of the input in the format (batch, w, h, c), None is used when the layer is part of a Network model.
- **cost_type** – cost_type or str Cost function to be applied to the layer, from the enum cost_type.
- **scale** – float (default=1.)
- **ratio** – float (default=0.)
- **noobject_scale** – float (default=1)
- **threshold** – float (default=0.)
- **smooothing** – float (default=0.)

Example

```
>>> import os
>>>
>>> import numpy as np
>>> import pylab as plt
>>> from PIL import Image
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1. / (im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1. / (im.max() - im.min()) * 255.
↪)).astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data', 'dog.jpg
↪')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
```

(continues on next page)

(continued from previous page)

```
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>>
>>> # batch == 1
>>> inpt = np.expand_dims(inpt, axis=0)
>>>
>>> cost_type = cost_type.mse
>>> scale = 1.
>>> ratio = 0.
>>> noobject_scale = 1.
>>> threshold = 0.
>>> smoothing = 0.
>>>
>>> truth = np.random.uniform(low=0., high=1., size=inpt.shape)
>>>
>>> layer = Cost_layer(input_shape=inpt.shape,
>>>                    cost_type=cost_type, scale=scale,
>>>                    ratio=ratio,
>>>                    noobject_scale=noobject_scale,
>>>                    threshold=threshold,
>>>                    smoothing=smoothing,
>>>                    trainable=True)
>>> print(layer)
>>>
>>> layer.forward(inpt, truth)
>>> forward_out = layer.output
>>>
>>> print('Cost: {:.3f}'.format(layer.cost))
```

References

- TODO

SECRET_NUM = 12345

backward(*delta*)

Backward function of the cost_layer, it updates the delta variable to be backpropagated. *self.delta* is updated inside the cost function.

Parameters **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.

Return type self

forward(*inpt*, *truth=None*)

Forward function for the cost layer. Using the chosen cost function, computes output, delta and cost.

Parameters

- **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c).
- **truth** (*array-like*) – truth values, it must have the same dimension as inpt.

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

Dropout layer

class layers.dropout_layer.**Dropout_layer**(*prob*, *input_shape=None*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

Dropout Layer

Drop a random selection of input pixels. This helps avoid overfitting.

Parameters

- **prob** (*float,*) – probability for each entry to be set to zero. It Ranges 0. to 1.
- **input_shape** (*tuple (default=None)*) – Shape of the input in the format (batch, w, h, c), None is used when the layer is part of a Network model.

Example

```
>>> import os
>>>
>>> import pylab as plt
>>> from PIL import Image
>>>
>>> np.random.seed(123)
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 255.)).
↪astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data', 'dog.jpg
↪')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>>
>>> inpt = np.expand_dims(inpt, axis=0)
>>>
>>> prob = 0.1
>>>
>>> layer = Dropout_layer(input_shape=inpt.shape, prob=prob)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt)
>>> forward_out = layer.output
>>>
```

(continues on next page)

(continued from previous page)

```
>>> print(layer)
>>>
>>> # BACKWARD
>>>
>>> delta = np.ones(shape=inpt.shape, dtype=float)
>>> layer.delta = np.ones(shape=layer.out_shape, dtype=float)
>>> layer.backward(delta)
>>>
>>> # Visualitations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>>
>>> fig.suptitle('Dropout Layer Drop Probability : {}'.format(prob))
>>> # Shown first image of the batch
>>> ax1.imshow(float_2_img(inpt[0]))
>>> ax1.set_title('Original image')
>>> ax1.axis('off')
>>>
>>> ax2.imshow(float_2_img(layer.output[0]))
>>> ax2.set_title('Forward')
>>> ax2.axis('off')
>>>
>>> ax3.imshow(float_2_img(delta[0]))
>>> ax3.set_title('Backward')
>>> ax3.axis('off')
>>>
>>> fig.tight_layout()
>>> plt.show()
```

References

- TODO

backward(*delta=None*)

Backward function of the Dropout layer Given the same mask as the layer it backprogates delta only to those pixel which values has not been set to zero in the forward function

Parameters **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.

Return type self

forward(*inpt*)

Forward function of the Dropout layer It create a random mask for every input in the batch and set to zero the chosen values. Other pixels are scaled with the inverse of (1 - prob)

Parameters **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c)

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

Input layer

class layers.input_layer.**Input_layer**(*input_shape*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

Input layer, this layer can be used at the beginning of a Network to define all the model's input-output dimensions

Parameters **input_shape** (*tuple*) – Shape of the input in the format (batch, w, h, c).

Example

```
>>> import os
>>>
>>> import pylab as plt
>>> from PIL import Image
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 255.)).
↪astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data', 'dog.jpg
↪')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>> inpt = np.expand_dims(inpt, axis=0)
>>>
>>> layer = Input_layer(input_shape=inpt.shape)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt)
>>> forward_out_byron = layer.output
>>>
>>> # BACKWARD
>>>
>>> delta = np.zeros(shape=inpt.shape, dtype=float)
>>> layer.backward(delta)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>>
>>> fig.suptitle('Input Layer')
>>>
>>> ax1.imshow(float_2_img(inpt[0]))
>>> ax1.set_title('Original image')
>>> ax1.axis('off')
>>>
>>> ax2.imshow(float_2_img(layer.output[0]))
>>> ax2.set_title("Forward")
>>> ax2.axis("off")
>>>
>>> ax3.imshow(float_2_img(delta[0]))
>>> ax3.set_title('Backward')
>>> ax3.axis('off')
>>>
>>> fig.tight_layout()
>>> plt.show()
```

References

TODO

backward(*delta*)

Simply pass the gradient.

Parameters **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.

Return type self

forward(*inpt*)

Forward function of the Input Layer: simply store the input array.

Parameters **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c)

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

L1Norm layer

class layers.l1norm_layer.**L1Norm_layer**(*input_shape=None*, *axis=None*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

L1Norm layer

Parameters

- **input_shape** (*tuple (default=None)*) – Shape of the input in the format (batch, w, h, c), None is used when the layer is part of a Network model.
- **axis** (*integer, default None.*) – Axis along which the L1Normalization is performed. If None, normalize the entire array.

Example

```
>>> import os
>>>
>>> import pylab as plt
>>> from PIL import Image
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 255.)).
↪astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data', 'dog.jpg
↪')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>>
>>> # add batch = 1
>>> inpt = np.expand_dims(inpt, axis=0)
>>>
>>> layer = L1Norm_layer(input_shape=inpt.shape)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt)
>>> forward_out = layer.output
>>> print(layer)
>>>
>>> # BACKWARD
>>>
>>> delta = np.zeros(shape=inpt.shape, dtype=float)
>>> layer.backward(delta, copy=True)
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> fig.suptitle('L1Normalization Layer')
>>>
>>> ax1.imshow(float_2_img(inpt[0]))
>>> ax1.set_title('Original image')
>>> ax1.axis('off')
>>>
>>> ax2.imshow(float_2_img(forward_out[0]))
>>> ax2.set_title("Forward")
>>> ax2.axis("off")
>>>
>>> ax3.imshow(float_2_img(delta[0]))
>>> ax3.set_title('Backward')
>>> ax3.axis('off')
>>>
>>> fig.tight_layout()
>>> plt.show()
```

TODO

backward(*delta*)

Backward function of the l1norm_layer

Parameters **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.

Return type self

forward(*inpt*)

Forward of the l1norm layer, apply the l1 normalization over the input along the given axis

Parameters **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c)

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

L2Norm layer

class layers.l2norm_layer.**L2Norm_layer**(*input_shape=None*, *axis=None*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

L2Norm layer

Parameters

- **input_shape** (*tuple (default=None)*) – Shape of the input in the format (batch, w, h, c), None is used when the layer is part of a Network model.
- **axis** (*integer, default None.*) – Axis along which the L1Normalization is performed. If None, normalize the entire array.

Example

```
>>> import os
>>>
>>> import pylab as plt
>>> from PIL import Image
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 1.)).
↪astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min()) * 255.)).
↪astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data', 'dog.jpg
↪')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>>
>>> # add batch = 1
>>> inpt = np.expand_dims(inpt, axis=0)
>>>
>>> layer = L2Norm_layer(input_shape=inpt.shape)
>>>
>>> # FORWARD
```

```
>>> layer.forward(inpt)
>>> forward_out = layer.output
>>> print(layer)
>>>
>>> # BACKWARD
>>>
>>> delta = np.zeros(shape=inpt.shape, dtype=float)
>>> layer.backward(delta)
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>>
>>> fig.suptitle('L2Normalization Layer')
>>>
>>> ax1.imshow(float_2_img(inpt[0]))
>>> ax1.set_title('Original image')
>>> ax1.axis('off')
>>>
>>> ax2.imshow(float_2_img(forward_out[0]))
>>> ax2.set_title("Forward")
>>> ax2.axis("off")
>>>
>>> ax3.imshow(float_2_img(delta[0]))
>>> ax3.set_title('Backward')
>>> ax3.axis('off')
>>>
```

(continues on next page)

(continued from previous page)

```
>>> fig.tight_layout()
>>> plt.show()
```

TODO

backward(*delta*)

Backward function of the l2norm layer

Parameters **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.

Return type self

forward(*inpt*)

Forward of the l2norm layer, apply the l2 normalization over the input along the given axis

Parameters **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c)

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

Logistic layer

class layers.logistic_layer.**Logistic_layer**(*input_shape=None*)

Bases: NumPyNet.layers.base.BaseLayer

Logistic Layer: performs a logistic transformation of the input and computes the binary cross entropy cost.

input_shape [tuple (default=None)] Shape of the input in the format (batch, w, h, c), None is used when the layer is part of a Network model.

```
>>> import os
>>>
>>> import pylab as plt
>>> from PIL import Image
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min())␣
↪* 1.)).astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min())␣
↪* 255.)).astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname(__file__), '..', '..', 'data',
↪'dog.jpg')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
>>> inpt = img_2_float(inpt)
>>> inpt = inpt * 2. - 1.
>>>
>>> inpt = np.expand_dims(inpt, axis=0)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> np.random.seed(123)
>>> batch, w, h, c = inpt.shape
>>>
>>> # truth definition, it's random so don't expect much
>>> truth = np.random.choice([0., 1.], p=[.5, .5], size=(batch, w, h, c))
>>>
>>> # Model Initialization
>>> layer = Logistic_layer(input_shape=inpt.shape)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt, truth)
>>> forward_out = layer.output
>>> layer_loss = layer.cost
>>>
>>> print(layer)
>>> print('Loss: {:.3f}'.format(layer_loss))
>>>
>>> # BACKWARD
>>>
>>> delta = np.zeros(shape=inpt.shape, dtype=float)
>>> layer.backward(delta)
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>>
>>> fig.suptitle('Logistic Layer:
```

loss({0:.3f})'.format(layer_loss))

```
>>>
>>> ax1.imshow(float_2_img(inpt[0]))
>>> ax1.axis('off')
>>> ax1.set_title('Original Image')
>>>
>>> ax2.imshow(float_2_img(forward_out[0]))
>>> ax2.axis('off')
>>> ax2.set_title('Forward Image')
>>>
>>> ax3.imshow(float_2_img(delta[0]))
>>> ax3.axis('off')
>>> ax3.set_title('Delta Image')
>>>
>>> fig.tight_layout()
>>> plt.show()
```

TODO

backward(*delta=None*)

Backward function of the Logistic Layer

Parameters **delta** (*array-like (default = None)*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.

Return type self

forward(*inpt*, *truth=None*)

Forward function of the logistic layer

Parameters

- **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c)
- **truth** (*array-like (default = None)*) – truth values, it must have the same dimension as inpt. If None, the layer does not compute the cost, but simply tranform the input

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

Maxpool layer

class layers.maxpool_layer.**Maxpool_layer**(*size*, *stride=None*, *pad=False*, *input_shape=None*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

Maxpool layer

size [tuple or int] Size of the kernel to slide over the input image. If a tuple, it must contains two integers, (kx, ky). If a int, size = kx = ky.

stride [tuple or int (default = None)] Represents the horizontal and vertical stride of the kernel (sx, sy). If None or 0, stride is assigned the same values as *size*.

input_shape [tuple (default = None)] Input shape of the layer. The default value is used when the layer is part of a network.

pad [bool, (default = False)] If False the image is cut to fit the size and stride dimensions, if True the image is padded following keras SAME padding, see references for details.

```
>>> import os
>>>
>>> import pylab as plt
>>> from PIL import Image
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1. / (im.max() - im.
↪min()) * 1.)).astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1. / (im.max() - im.
↪min()) * 255.)).astype(np.uint8)
>>>
>>> filename = os.path.join(os.path.dirname('__file__'), '..', '..', 'data',
↪ 'dog.jpg')
>>> inpt = np.asarray(Image.open(filename), dtype=float)
>>> inpt.setflags(write=1)
```

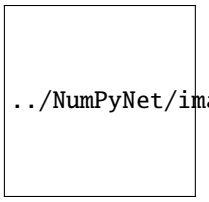
(continues on next page)

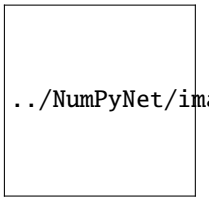
(continued from previous page)

```
>>> inpt = img_2_float(inpt)
>>>
>>> inpt = np.expand_dims(inpt, axis=0)  # Add the batch shape.
>>> b, w, h, c = inpt.shape
>>>
>>> size = (3, 3)
>>> stride = (2, 2)
>>> pad = False
>>>
>>> layer = Maxpool_layer(input_shape=inpt.shape, size=size, stride=stride,␣
↪padding=pad)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt)
>>>
>>> forward_out = layer.output
>>>
>>> print(layer)
>>>
>>> # BACKWARD
>>>
>>> delta = np.zeros(inpt.shape, dtype=float)
>>> layer.delta = np.ones(layer.out_shape, dtype=float)
>>> layer.backward(delta)
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>> fig.suptitle('MaxPool Layer
```

size [{}, stride][{}, padding][{} '.format(size, stride, pad))]

```
>>>
>>> ax1.imshow(float_2_img(inpt[0]))
>>> ax1.set_title('Original Image')
>>> ax1.axis('off')
>>>
>>> ax2.imshow(float_2_img(forward_out[0]))
>>> ax2.set_title('Forward')
>>> ax2.axis('off')
>>>
>>> ax3.imshow(float_2_img(delta[0]))
>>> ax3.set_title('Backward')
>>> ax3.axis('off')
>>>
>>> fig.tight_layout()
>>> plt.show()
```





- https://docs.scipy.org/doc/numpy/reference/generated/numpy.lib.stride_tricks.as_strided.html
- https://stackoverflow.com/questions/42463172/how-to-perform-max-mean-pooling-on-a-2d-array-using-numpy
- https://stackoverflow.com/questions/42463172/how-to-perform-max-mean-pooling-on-a-2d-array-using-numpys

backward(*delta*)

Backward function of maxpool layer: it access avery position where in the input image there's a chosen maximum and add the correspondent self.delta value. Since we work with a 'view' of delta, the same pixel may appear more than one time, and an atomic acces to it's value is needed to correctly modifiy it.

Parameters **delta** (*array-like*) – Global delta to be backpropagated with shape (batch, out_w, out_h, out_c).

Return type self

forward(*inpt*)

Forward function of the maxpool layer: It slides a kernel over every input image and return the maximum value of every sub-window. the function _asStride returns a view of the input arrary with shape (batch, out_w, out_h , c, kx, ky), where, for every image in the batch we have: out_w * out_h * c sub matrixes kx * ky, containing pixel values.

Parameters **inpt** (*array-like*) – Input batch of images, with shape (batch, input_w, input_h, input_c).

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

Route layer

class layers.route_layer.**Route_layer**(*input_layers*, *by_channels=True*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

Route layer For Now the idea is: it takes the seleted layers output and concatenate them along the batch axis OR the channels axis

YOLOv3 implementation always concatenate by channels

By definition, this layer can't be used without a Network model.

Parameters

- **input_layers** (*int or list of int.*) – indexes of the layers in the network for which the outputs have to concatenated.
- **by_channels** (*bool, (default = True).*) – It determines along which dimension the concatenation is performed. For examples if two input with size (b1, w, h , c) and (b2, w, h, c) are concatenated with by_channels=False, then the final output shape will be (b1 + b2, w, h, c). Otherwise, if the shapes are (b, w, h, c1) and (b, w, h, c2) and axis=3, the final output size will be (b, w, h, c1 + c2) (YOLOv3 model). Notice that the all the other dimensions must be equal.

Example

TODO

TODO

backward(*delta*, *network*)

Sum self.delta to the correct layer delta on the network

Parameters

- **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.
- **network** (*Network object type.*) – The network model to which this layer belongs to.

Return type self

forward(*network*)

Concatenate along chosen axis the outputs of selected network layers In main CNN applications, like YOLOv3, the concatenation happens long channels axis

Parameters **network** (*Network object type.*) – The network model to which this layer belongs to.

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

Shortcut layer

class layers.shortcut_layer.**Shortcut_layer**(*activation=<class 'NumPyNet.activations.Activations'>*, *alpha=1.0*, *beta=1.0*, ***kwargs*)

Bases: NumPyNet.layers.base.BaseLayer

Shortcut layer: activation of the linear combination of the output of two layers

layer1 * alpha + layer2 * beta = output

Now working only with same shapes input

activation [str or Activation object] Activation function of the layer.

alpha [float, (default = 1.)] first weight of the combination.

beta [float, (default = 1.)] second weight of the combination.

```
>>> import pylab as plt
>>>
>>> from NumPyNet import activations
>>>
>>> img_2_float = lambda im : ((im - im.min()) * (1./(im.max() - im.min())␣
↪* 1.)).astype(float)
>>> float_2_img = lambda im : ((im - im.min()) * (1./(im.max() - im.min())␣
↪* 255.)).astype(np.uint8)
>>>
>>> # Set seed to have same input
>>> np.random.seed(123)
>>>
>>> layer_activ = activations.Relu()
>>>
>>> batch = 2
>>>
>>> alpha = 0.75
>>> beta  = 0.5
>>>
>>> # Random input
>>> inpt1      = np.random.uniform(low=-1., high=1., size=(batch, 100, 100,␣
↪3))
>>> inpt2      = np.random.uniform(low=-1., high=1., size=inpt1.shape)
>>> b, w, h, c = inpt1.shape
>>>
>>>
>>> # model initialization
>>> layer = Shortcut_layer(activation=layer_activ,
>>>                        alpha=alpha, beta=beta)
>>>
>>> # FORWARD
>>>
>>> layer.forward(inpt1, inpt2)
>>> forward_out = layer.output.copy()
>>>
>>> print(layer)
>>>
>>> # BACKWARD
>>>
>>> delta      = np.zeros(shape=inpt1.shape, dtype=float)
>>> delta_prev = np.zeros(shape=inpt2.shape, dtype=float)
>>>
>>> layer.delta = np.ones(shape=layer.out_shape, dtype=float)
>>> layer.backward(delta, delta_prev)
>>>
>>> # Visualizations
>>>
>>> fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(10, 5))
>>> fig.subplots_adjust(left=0.1, right=0.95, top=0.95, bottom=0.15)
>>> fig.suptitle('Shortcut Layer
```

alpha [{}, beta][{}, activation][{} '.format(alpha, beta, layer_activ.name))]

```
>>>
>>> ax1.imshow(float_2_img(inpt1[0]))
>>> ax1.set_title('Original Image')
>>> ax1.axis('off')
>>>
>>> ax2.imshow(float_2_img(forward_out[0]))
>>> ax2.set_title('Forward')
>>> ax2.axis('off')
>>>
>>> ax3.imshow(float_2_img(delta[0]))
>>> ax3.set_title('Backward')
>>> ax3.axis('off')
>>>
>>> fig.tight_layout()
>>> plt.show()
```

TODO

backward(*delta*, *prev_delta*, *copy=False*)

Backward function of the Shortcut layer

Parameters

- **delta** (*array-like*) – delta array of shape (batch, w, h, c). Global delta to be backpropagated.
- **delta_prev** (*array-like*) – second delta to be backporpagated.
- **copy** (*bool (default=False)*) – States if the activation function have to return a copy of the input or not.

Return type self

forward(*inpt*, *prev_output*, *copy=False*)

Forward function of the Shortcut layer: activation of the linear combination between input.

Parameters

- **inpt** (*array-like*) – Input batch of images in format (batch, in_w, in_h, in _c)
- **prev_output** (*array-like*) – second input of the layer

Return type self

property out_shape

Get the output shape

Returns **out_shape** – Tuple as (batch, out_w, out_h, out_c)

Return type tuple

2.1.2 NumPyNet utility

Box

class box.**Box**(*coords=None*)

Bases: object

Detection box class

Parameters **coords** (*tuple (default=None)*) – Box Coordinates as (x, y, w, h)

Example

```
>>> import pylab as plt
>>> from matplotlib.patches import Rectangle
>>>
>>> b1 = Box((.5, .3, .2, .1))
>>> x_1, y_1, w_1, h_1 = b1.box
>>> left_1, top_1, right_1, bottom_1 = b1.coords
>>>
>>> print('Box1: {}'.format(b1))
>>>
>>> b2 = Box((.4, .5, .2, .5))
>>> x_2, y_2, w_2, h_2 = b2.box
>>> left_2, top_2, right_2, bottom_2 = b2.coords
>>>
>>> print('Box2: {}'.format(b2))
>>>
>>> print('Intersection: {:.3f}'.format(b1.intersection(b2)))
>>> print('Union: {:.3f}'.format(b1.union(b2)))
>>> print('IOU: {:.3f}'.format(b1.iou(b2)))
>>> print('rmse: {:.3f}'.format(b1.rmse(b2)))
>>>
>>> plt.figure()
>>> axis = plt.gca()
>>> axis.add_patch(Rectangle(xy=(left_1, top_1),
>>>                          width=w_1, height=h_1,
>>>                          alpha=.5, linewidth=2, color='blue'))
>>> axis.add_patch(Rectangle(xy=(left_2, top_2),
>>>                          width=w_2, height=h_2,
>>>                          alpha=.5, linewidth=2, color='red'))
```

property area

Compute the are of the box

Returns **area** – Area of the current box.

Return type float

property box

Get the box coordinates

Returns **coords** – Box coordinates as (x, y, w, h)

Return type tuple

`property center`

In the current storage the x,y are the center of the box

Returns **center** – Center of the current box.

Return type tuple

`property coords`

Return box coordinates in clock order (left, top, right, bottom)

Returns **coords** – Coordinates as (left, top, right, bottom)

Return type tuple

`property dimensions`

In the current storage the w,h are the dimensions of the rectangular box

Returns **dims** – Dimensions of the current box as (width, height).

Return type tuple

`intersection`(*other*)

Common area between boxes

Parameters **other** (Box) – 2nd term of the evaluation

Returns **intersection** – Intersection area of two boxes

Return type float

`iou`(*other*)

Intersection over union

Parameters **other** (Box) – 2nd term of the evaluation

Returns **iou** – Intersection over union between boxes

Return type float

`rmse`(*other*)

Root mean square error of the boxes

Parameters **other** (Box) – 2nd term of the evaluation

Returns **rmse** – Root mean square error of the boxes

Return type float

`union`(*other*)

Full area without intersection

Parameters **other** (Box) – 2nd term of the evaluation

Returns **union** – Union area of the two boxes

Return type float

DataGenerator

class data.**DataGenerator**(*load_func*, *batch_size*, *source_path=None*, *source_file=None*, *label_path=None*, *label_file=None*, *source_extension=''*, *label_extension=''*, *seed=123*, ***load_func_kwargs*)

Bases: `object`

Data generator in detached thread.

Parameters

- **load_func** (*function or lambda*) – Function to apply for the preprocessing on a single data/label pair
- **batch_size** (*int*) – Dimension of batch to load
- **source_path** (*str (default=None)*) – Path to the source files
- **source_file** (*str (default=None)*) – Filename in which is stored the list of source files
- **label_path** (*str (default=None)*) – Path to the label files
- **label_file** (*str (default=None)*) – Filename in which is stored the list of label files
- **source_extension** (*str (default='')*) – Extension of the source files
- **label_extension** (*str (default='')*) – Extension of the label files
- **seed** (*int*) – Random seed
- ****load_func_kwargs** (*dict*) – Optional parameters to use in the load_func

Example

```
>>> import pylab as plt
>>>
>>> train_gen = DataGenerator(load_func=load_segmentation, batch_size=2,
>>>                           source_path='/path/to/train/images',
>>>                           label_path='/path/to/mask/images',
>>>                           source_extension='.png',
>>>                           label_extension='.png'
>>>                           )
>>> train_gen.start()
>>>
>>> fig, ((ax00, ax01), (ax10, ax11)) = plt.subplots(nrows=2, ncols=2)
>>>
>>> for i in range(10):
>>>   grabbed = False
>>>
>>>   while not grabbed:
>>>
>>>     (data1, data2), (label1, label2), grabbed = train_gen.load_data()
>>>
>>>   ax00.imshow(data1.get(), cmap='gray')
>>>   ax00.axis('off')
>>>
>>>   ax01.imshow(label1.get(), cmap='gray')
>>>   ax01.axis('off')
```

(continues on next page)

(continued from previous page)

```
>>>
>>>    ax10.imshow(data2.get(), cmap='gray')
>>>    ax10.axis('off')
>>>
>>>    ax11.imshow(label2.get(), cmap='gray')
>>>    ax11.axis('off')
>>>
>>>    plt.pause(1e-2)
>>>
>>>  plt.show()
>>>
>>>  train_gen.stop()
```

load_data()

Get a batch of images and labels

Returns

- **data** (*obj*) – Loaded data
- **label** (*obj*) – Loaded label
- **stopped** (*bool*) – Check if the end of the list is achieved

property num_data

Get the number of data

start()

Start the thread

stop()

Stop the thread

data.**load_segmentation**(*source_image_filename*, *mask_image_filename*)

Load Segmentation data.

Parameters

- **source_image_filename** (*str*) – Filename of the source image
- **mask_image_filename** (*str*) – Filename of the corresponding mask image in binary format

Returns

- **src_image** (*Image*) – Loaded Image object
- **mask_image** (*Image*) – Image label as mask image

Notes

Note: In Segmentation model we have to feed the model with a simple image and the labels will be given by the mask (binary) of the same image in which the segmentation parts are highlight No checks are performed on the compatibility between source image and corresponding mask file. The only checks are given on the image size (channels are excluded)

data.**load_super_resolution**(*hr_image_filename*, *patch_size=(48, 48)*, *scale=4*)

Load Super resolution data.

Parameters

- **hr_image_filename** (*string*) – Filename of the high resolution image
- **patch_size** (*tuple (default=(48, 48))*) – Dimension to cut
- **scale** (*int (default=4)*) – Downsampling scale factor

Returns

- **data** (*Image obj*) – Loaded Image object
- **label** (*Image obj*) – Generated Image label

Notes

Note: In SR models the labels are given by the HR image while the input data are obtained from the same image after a downsampling/resizing. The upsample scale factor learned by the SR model will be the same used inside this function.

Detection

class detection.**Detection**(*num_classes=None*, *mask_size=None*)

Bases: object

Detection object

Parameters

- **num_classes** (*int (default=None)*) – Number of classes to monitor
- **mask_size** (*int (default=None)*) – Size of the possible mask values

Notes

Note: The detection object stores the detection probability of each class and its "objectness". Moreover in the member "bbox" are store the detection box infos as Box object, aka (x, y, w, h)

property box

Return the box object as tuple

static do_nms_obj(*detections*, *thresh*)

Sort the detection according to the probability of each class and perform the IOU as filter for the boxes

Parameters

- **detections** (*array_like (1D array)*) – Array of detection objects.
- **thresh** (*float*) – Threshold to apply for IoU filtering. If IoU is greater than thresh the corresponding objectness and probabilities are set to null.

Returns **dets** – Array of detection objects processed.

Return type array_like (1D array)

static do_nms_sort(*detections*, *thresh*)

Sort the detection according to the objectness and perform the IOU as filter for the boxes.

Parameters

- **detections** (*array_like (1D array)*) – Array of detection objects.
- **thresh** (*float*) – Threshold to apply for IoU filtering. If IoU is greater than thresh the corresponding objectness and probabilities are set to null.

Returns **dets** – Array of detection objects processed.

Return type array_like (1D array)

property objectness

Return the objectness of the detection

property prob

Return the probability of detection for each class

static top_k_predictions(*output*)

Compute the indices of the sorted output

Parameters **output** (*array_like (1D array)*) – Array of predictions expressed as floats. Its value will be sorted in ascending order and the corresponding array of indices is given in output.

Returns **indexes** – Array of indexes which sort the output values in ascending order.

Return type list (int32 values)

Fast math operations

fmath.**atanh**(*x*)

Fast math version of 'atanh' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**exp**(*x*)

Fast math version of 'exp' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**hardtanh**(*x*)

Fast math version of 'hardtanh' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**log**(*x*)

Fast math version of 'log' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**log10**(*x*)

Fast math version of 'log10' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**log2**(*x*)

Fast math version of 'log2' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**pow**(*a*, *b*)

Fast math version of 'pow' function

Parameters

- **a** (*float*) – Base
- **b** (*float*) – Exponent

Returns **res** – Result of the function

Return type float

fmath.**pow2**(*x*)

Fast math version of 'pow2' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**rsqrt**(*x*)

Fast math version of 'rsqrt' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**sqrt**(*x*)

Fast math version of 'sqrt' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

fmath.**tanh**(*x*)

Fast math version of 'tanh' function

Parameters **x** (*float*) – Value to evaluate

Returns **res** – Result of the function

Return type float

Image

class image.**Image**(*filename=None*)

Bases: object

Constructor of the image object. If filename the load function loads the image file.

Parameters **filename** (*str (default=None)*) – Image filename

add_single_batch()

Add batch dimension for testing layer

Return type self

property channels

Get the image number of channels

crop(*dsize*, *size*)

Crop the image according to the given dimensions [dsize[0] : dsize[0] + size[0], dsize[1] : dsize[1] + size[1]]

Parameters

- **dsize** (*2D iterable*) – (X, Y) of the crop
- **size** (*2D iterable*) – (width, height) of the crop

Returns **cropped** – Cropped image

Return type *Image*

draw_detections(*dets*, *thresh*, *names*)

Draw the detections into the current image

Parameters

- **dets** (*Detection list*) – List of pre-computed detection objects
- **thresh** (*float*) – Probability threshold to filter the boxes
- **names** (*iterable*) – List of object names as strings

Return type self

flip(*axis=- 1*)

Flip the image along given axis (0 - horizontal, 1 - vertical)

Parameters **axis** (*int (default=0)*) – Axis to flip

Return type self

from_frame(*array*)

Use opencv frame array as the image

from_numpy_matrix(*array*)

Use numpy array as the image

Parameters **array** (*array_like*) – buffer of the input image as (width, height, channel)

Return type self

get()

Return the data object as a numpy array

Returns **data** – Image data as numpy array

Return type array-like

property height

Get the image height

letterbox(*net_dim*)

resize image with unchanged aspect ratio using padding

Parameters **net_dim** (*2D iterable*) – width and height outputs

Returns **resized** – Resized Image

Return type *Image*

load(*filename*)

Read Image from file

Parameters **filename** (*str*) – Image filename path

Return type self

mean_std_norm()

Normalize the current image as

```
image = (image - mean) / variance
```

Return type self

remove_single_batch()

Remove batch dimension for testing layer

Return type self

rescale(*var*, *process=normalization.normalize*)

Divide or multiply by train variance-image

Parameters

- **variances** (*array_like*) – Array of variances to apply to the image
- **process** (*normalization (int)*) – Switch between normalization and denormalization

Return type self

resize(*dsize=None*, *scale_factor=(None, None)*)

Resize the image according to the new shape given

Parameters

- **dsize** (*2D iterable (default=None)*) – Destination size of the image
- **scale_factor** (*2D iterable (default=(None, None))*) – width scale factor, height scale factor

Returns **res** – Resized Image

Return type *Image*

Notes

Note: The resize is performed using the LANCZOS interpolation.

rgb2rgba()

Add alpha channel to the original image

Return type self

Notes

Note: Pay attention to the value of the alpha channel! OpenCV does not set its values to null but they are and empty (garbage) array.

rotate(*angle*)

Rotate the image according to the given angle in degree fmt.

Parameters **angle** (*float*) – Angle in degree fmt

Returns **rotated** – Rotated image

Return type *Image*

Note:

Note: This rotation preserves the original size so some original parts can be removed from the rotated image. See 'rotate_bound' for a conservative rotation.

References

https://www.pyimagesearch.com/2017/01/02/rotate-images-correctly-with-opencv-and-python/

rotate_bound(*angle*)

Rotate the image according to the given angle in degree fmt.

Parameters **angle** (*float*) – Angle in degree fmt

Returns **rotated** – Rotated image

Return type *Image*

Note:

Note: This rotation preserves the original image, so the output can be greater than the original size. See 'rotate' for a rotation which preserves the size.

References

https://www.pyimagesearch.com/2017/01/02/rotate-images-correctly-with-opencv-and-python/

save(*filename*)

save the image

Parameters **filename** (*str*) – Output filename of the image

Return type True if everything is ok

scale(*scaling*, *process=normalization.normalize*)

Scale image values

Parameters

- **scale** (*float*) – Scale factor to apply to the image
- **process** (*normalization (int, default = normalize)*) – Switch between normalization (0) and denormalization (1)

Return type self

scale_between(*minimum*, *maximum*)

Rescale image value between min and max

Parameters

- **minimum** (*float (default = 0.)*) – Min value
- **maximum** (*float (default = 1.)*) – Max value

Return type self

property shape

Get the image dimensions

show(*window_name*, *ms=0*, *fullscreen=None*)

show the image

Parameters

- **window_name** (*str*) – Name of the plot
- **ms** (*int (default=0)*) – Milliseconds to wait

Returns **check** – True if everything is ok

Return type bool

standardize(*means*, *process=normalization.normalize*)

Remove or add train mean-image from current image

Parameters

- **means** (*array_like*) – Array of means to apply to the image
- **process** (*normalization (int, default = normalize)*) – Switch between normalization (0) and denormalization (1)

Return type self

transpose()

Transpose width and height

Return type self

property width

Get the image width

Metrics

metrics.**mean_absolute_error**(*y_true*, *y_pred*)

Compute average absolute error score of a classification.

Parameters

- **y_true** (*2d array-like*) – Ground truth (correct) labels expressed as image.
- **y_pred** (*2d array-like*) – Predicted labels, as returned by the NN

Returns **score** – Average absolute error between the two inputs

Return type float

metrics.**mean_accuracy_score**(*y_true*, *y_pred*)

Compute average accuracy score of a classification.

Parameters

- **y_true** (*2d array-like*) – Ground truth (correct) labels expressed as image.
- **y_pred** (*2d array-like*) – Predicted labels, as returned by the NN

Returns **score** – Average accuracy between the two inputs

Return type float

metrics.**mean_hellinger**(*y_true*, *y_pred*)

Compute average hellinger score of a classification.

Parameters

- **y_true** (*2d array-like*) – Ground truth (correct) labels expressed as image.
- **y_pred** (*2d array-like*) – Predicted labels, as returned by the NN

Returns **score** – Average hellinger error between the two inputs

Return type float

metrics.**mean_iou_score**(*y_true*, *y_pred*)

Compute average IoU score of a classification. IoU is computed as Intersection Over Union between true and predict labels.

It's a tipical metric in segmentation problems, so we encourage to use it when you are dealing image processing tasks.

Parameters

- **y_true** (*2d array-like*) – Ground truth (correct) labels expressed as image.
- **y_pred** (*2d array-like*) – Predicted labels, as returned by the NN

Returns **score** – Average IoU between the two inputs

Return type float

metrics.**mean_logcosh**(*y_true*, *y_pred*)

Compute average logcosh score of a classification.

Parameters

- **y_true** (*2d array-like*) – Ground truth (correct) labels expressed as image.
- **y_pred** (*2d array-like*) – Predicted labels, as returned by the NN

Returns **score** – Average logcosh error between the two inputs

Return type float

metrics.**mean_square_error**(*y_true*, *y_pred*)

Compute average square error score of a classification.

Parameters

- **y_true** (*2d array-like*) – Ground truth (correct) labels expressed as image.
- **y_pred** (*2d array-like*) – Predicted labels, as returned by the NN

Returns **score** – Average square error between the two inputs

Return type float

Network

class network.**Network**(*batch*, *input_shape=None*, *train=True*)

Bases: object

Neural Network object

Parameters

- **batch** (*int*) – Batch size
- **input_shape** (*tuple*) – Input dimensions
- **train** (*bool (default=True)*) – Turn on/off the parameters tuning

Notes

Warning: Up to now the trainable variable is useless since the layer doesn't take it into account!

```
LAYERS = {'activation':  <class
'NumPyNet.layers.activation_layer.Activation_layer'>, 'avgpool':  <class
'NumPyNet.layers.avgpool_layer.Avgpool_layer'>, 'batchnorm':  <class
'NumPyNet.layers.batchnorm_layer.BatchNorm_layer'>, 'connected':  <class
'NumPyNet.layers.connected_layer.Connected_layer'>, 'convolutional':  <class
'NumPyNet.layers.convolutional_layer.Convolutional_layer'>, 'cost':  <class
'NumPyNet.layers.cost_layer.Cost_layer'>, 'dropout':  <class
'NumPyNet.layers.dropout_layer.Dropout_layer'>, 'input':  <class
'NumPyNet.layers.input_layer.Input_layer'>, 'l1norm':  <class
'NumPyNet.layers.l1norm_layer.L1Norm_layer'>, 'l2norm':  <class
'NumPyNet.layers.l2norm_layer.L2Norm_layer'>, 'logistic':  <class
'NumPyNet.layers.logistic_layer.Logistic_layer'>, 'lstm':  <class
'NumPyNet.layers.lstm_layer.LSTM_layer'>, 'maxpool':  <class
'NumPyNet.layers.maxpool_layer.Maxpool_layer'>, 'rnn':  <class
'NumPyNet.layers.rnn_layer.RNN_layer'>, 'route':  <class
'NumPyNet.layers.route_layer.Route_layer'>, 'shortcut':  <class
'NumPyNet.layers.shortcut_layer.Shortcut_layer'>, 'shuffler':  <class
'NumPyNet.layers.shuffler_layer.Shuffler_layer'>, 'simplernn':  <class
'NumPyNet.layers.simple_rnn_layer.SimpleRNN_layer'>, 'softmax':  <class
'NumPyNet.layers.softmax_layer.Softmax_layer'>, 'upsample':  <class
'NumPyNet.layers.upsample_layer.Upsample_layer'>, 'yolo':  <class
'NumPyNet.layers.yolo_layer.Yolo_layer'>}
```

add(*layer*)

Add a new layer to the network model. Layers are progressively appended to the tail of the model.

Parameters **layer** (*Layer object*) – Layer object to append to the current architecture

Return type self

Notes

Note: If the architecture is empty a default InputLayer is used to start the model.

Warning: The input layer type must be one of the types stored into the LAYERS dict, otherwise a LayerError is raised.

compile(*optimizer=<class 'NumPyNet.optimizer.Optimizer'>*, *metrics=None*)

Compile the neural network model setting the optimizer to each layer and the evaluation metrics

Parameters

- **optimizer** (Optimizer) – Optimizer object to use during the training
- **metrics** (*list (default=None)*) – List of metrics functions to use for the model evaluation.

Notes

Note: The optimizer is copied into each layer object which requires a parameters optimization.

evaluate(*X*, *truth*, *verbose=False*)

Return output and loss of the model

Parameters

- **X** (*array-like*) – Input data
- **truth** (*array-like*) – Ground truth or labels
- **verbose** (*bool (default=False)*) – Turn on/off the verbosity given by the training progress bar

Returns

- **loss** (*float*) – The current loss of the model
- **output** (*array-like*) – Output of the model as numpy array

fit(*X*, *y*, *max_iter=100*, *shuffle=True*, *verbose=True*)

Fit/training function

Parameters

- **X** (*array-like*) – Input data
- **y** (*array-like*) – Ground truth or labels
- **max_iter** (*int (default=100)*) – Maximum number of iterations/epochs to perform
- **shuffle** (*bool (default=True)*) – Turn on/off the random shuffling of the data
- **verbose** (*bool (default=True)*) – Turn on/off the verbosity given by the training progress bar

Return type self

fit_generator(*Xy_generator*, *max_iter=100*)

Fit function using a train generator

Parameters

- **Xy_generator** (DataGenerator) – Data generator object
- **max_iter** (*int (default=100)*) – Maximum number of iterations/epochs to perform

Return type self

References

DataGenerator object in data.py

property input_shape

Get the input shape

load(*cfg_filename*, *weights=None*)

Load network model from config file in INI fmt

Parameters

- **cfg_filename** (*str*) – Filename or path of the neural network configuration file in INI format
- **weights** (*str (default=None)*) – Filename of the weights

Return type self

load_model(*model_filename*)

Load network model object as pickle

Parameters **model_filename** (*str*) – Filename or path of the model (binary) file

Return type self

Notes

Note: The model loading is performed using pickle. If the model was previously dumped with the save_model function everything should be ok.

load_weights(*weights_filename*)

Load weight from filename in binary fmt

Parameters **weights_filename** (*str*) – Filename of the input weights file

Return type self

Notes

Note: The weights are read and set to each layer which has the load_weights member function.

next()

Get the next layer

Notes

This should fix python2* problems with __iter__ and __next__

property num_layers

Get the number of layers in the model

property out_shape

Get the output shape

predict(*X*, *truth=None*, *verbose=True*)

Predict the given input

Parameters

- **X** (*array-like*) – Input data
- **truth** (*array-like (default=None)*) – Ground truth or labels
- **verbose** (*bool (default=True)*) – Turn on/off the verbosity given by the training progress bar

Returns **output** – Output of the model as numpy array

Return type array-like

save_model(*model_filename*)

Dump the current network model as pickle

Parameters **model_filename** (*str*) – Filename or path for the model dumping

Return type self

Notes

Note: The model is dumped using pickle binary format.

save_weights(*filename*)

Dump current network weights

Parameters **filename** (*str*) – Filename of the output weights file

Return type self

Notes

Note: The weights are extracted from each layer which has the save_weights member function.

summary()

Print the network model summary

Return type None

Optimizer

class optimizer.**Adadelta**(*rho=0.9*, *epsilon=1e-06*, **args*, ***kwargs*)

Bases: *optimizer.Optimizer*

AdaDelta optimization algorithm

Update the parameters according to the rule

```
c = rho * c + (1. - rho) * gradient * gradient
update = gradient * sqrt(d + epsilon) / (sqrt(c) + epsilon)
parameter -= learning_rate * update
d = rho * d + (1. - rho) * update * update
```

Parameters

- **rho** (*float (default=0.9)*) – Decay factor
- **epsilon** (*float (default=1e-6)*) – Precision parameter to overcome numerical overflows
- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the given parameters according to the class optimization algorithm

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Returns **params** – The updated parameters

Return type list

class optimizer.**Adagrad**(*epsilon=1e-06*, **args*, ***kwargs*)

Bases: *optimizer.Optimizer*

Adagrad optimizer specialization

Update the parameters according to the rule

```
c += gradient * gradient
parameter -= learning_rate * gradient / (sqrt(c) + epsilon)
```

Parameters

- **epsilon** (*float (default=1e-6)*) – Precision parameter to overcome numerical overflows
- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the given parameters according to the class optimization algorithm

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Returns **params** – The updated parameters

Return type list

class optimizer.**Adam**(*beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, **args*, ***kwargs*)

Bases: *optimizer.Optimizer*

Adam optimization algorithm

Update the parameters according to the rule

```
at  = learning_rate * sqrt(1 - B2**iterations) / (1 - B1**iterations)
m = B1 * m + (1 - B1) * gradient
v = B2 * m + (1 - B2) * gradient * gradient
parameter -= at * m / (sqrt(v) + epsilon)
```

Parameters

- **beta1** (*float (default=0.9)*) – B1 factor
- **beta2** (*float (default=0.999)*) – B2 factor
- **epsilon** (*float (default=1e-8)*) – Precision parameter to overcome numerical overflows
- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the given parameters according to the class optimization algorithm

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Returns **params** – The updated parameters

Return type list

class optimizer.**Adamax**(*beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, **args*, ***kwargs*)

Bases: *optimizer.Optimizer*

Adamax optimization algorithm

Update the parameters according to the rule

```
at  = learning_rate / (1 - B1**iterations)
m = B1 * m + (1 - B1) * gradient
v = max(B2 * v, abs(gradient))
parameter -= at * m / (v + epsilon)
```

Parameters

- **beta1** (*float (default=0.9)*) – B1 factor
- **beta2** (*float (default=0.999)*) – B2 factor
- **epsilon** (*float (default=1e-8)*) – Precision parameter to overcome numerical overflows
- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the given parameters according to the class optimization algorithm

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Returns **params** – The updated parameters

Return type list

class optimizer.**Momentum**(*momentum=0.9*, **args*, ***kwargs*)

Bases: *optimizer.Optimizer*

Stochastic Gradient Descent with Momentum specialiation

Update the parameters according to the rule

```
v = momentum * v - lr * gradient
parameter += v - learning_rate * gradient
```

Parameters

- **momentum** (*float (default=0.9)*) – Momentum value
- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the given parameters according to the class optimization algorithm

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Returns **params** – The updated parameters

Return type list

class optimizer.**NesterovMomentum**(*momentum=0.9*, **args*, ***kwargs*)

Bases: *optimizer.Optimizer*

Stochastic Gradient Descent with Nesterov Momentum specialiation

Update the parameters according to the rule

```
v = momentum * v - lr * gradient
parameter += momentum * v - learning_rate * gradient
```

Parameters

- **momentum** (*float (default=0.9)*) – Momentum value
- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the given parameters according to the class optimization algorithm

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Returns **params** – The updated parameters

Return type list

class optimizer.**Optimizer**(*lr=0.001*, *decay=0.0*, *lr_min=0.0*, *lr_max=inf*, **args*, ***kwargs*)

Bases: object

Abstract base class for the optimizers

Parameters

- **lr** (*float (default=2e-2)*) – Learning rate value
- **decay** (*float (default=0.)*) – Learning rate decay
- **lr_min** (*float (default=0.)*) – Minimum of learning rate domain
- **lr_max** (*float (default=np.inf)*) – Maximum of learning rate domain
- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the optimizer parameters

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Return type self

class optimizer.**RMSprop**(*rho=0.9*, *epsilon=1e-06*, **args*, ***kwargs*)

Bases: *optimizer.Optimizer*

RMSprop optimization algorithm

Update the parameters according to the rule

```
c = rho * c + (1. - rho) * gradient * gradient
parameter -= learning_rate * gradient / (sqrt(c) + epsilon)
```

Parameters

- **rho** (*float (default=0.9)*) – Decay factor
- **epsilon** (*float (default=1e-6)*) – Precision parameter to overcome numerical overflows
- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the given parameters according to the class optimization algorithm

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Returns **params** – The updated parameters

Return type list

class optimizer.**SGD**(**args*, ***kwargs*)

Bases: *optimizer.Optimizer*

Stochastic Gradient Descent specialization

Update the parameters according to the rule

```
parameter -= learning_rate * gradient
```

Parameters

- ***args** (*list*) – Class specialization variables.
- ****kwargs** (*dict*) – Class Specialization variables.

update(*params*, *gradients*)

Update the given parameters according to the class optimization algorithm

Parameters

- **params** (*list*) – List of parameters to update
- **gradients** (*list*) – List of corresponding gradients

Returns **params** – The updated parameters

Return type list

Parser

class parser.**data_config**(*filename*)

Bases: object

Data configuration parser

Parameters **filename** (*str*) – Configuration data filename or path

Return type data_config object

Notes

Note: The data configuration stores the global parameters for a given model (ex. cfg filename, weight filename, ...) The file must be saved in a dictionary format like "cfg = config_filename.cfg"

get(*key*, *default=None*)

Getter function

Parameters

- **key** (*str*) – config dictionary key
- **default** (*dtype (default=None)*) – the default value if the key is not found in the data config

parser.**get_labels**(*filename*, *classes=- 1*)

Read the labels file

Parameters

- **filename** (*str*) – Labels filename or path
- **classes** (*int (default = -1)*) – Number of labels to read. If it is equal to -1 the full list of labels is read

Returns **labels** – The first 'classes' labels in the file.

Return type list

class parser.**net_config**(*filename*)

Bases: object

Network config parser

Parameters **filename** (*str*) – Network config filename or path

Return type net_config object

Notes

Note: The network configuration file must be stored in INI format. Since multiple layers can have the same type the dictionary must be overloaded by a custom OrderedDict

get(*section*, *key*, *default=None*)

Getter function

Parameters

- **section** (*str*) – Layer name + position
- **key** (*str*) – config dictionary key
- **default** (*dtype (default=None)*) – the default value if the key is not found in the data config

get_params(*section*)

Return all params in section as dict

Parameters **section** (*str*) – Layer name + position

Returns **params** – Dictionary of all (keys, values) in section

Return type dict

class multidict

Bases: collections.OrderedDict

clear() → None. Remove all items from od.

copy() → a shallow copy of od

fromkeys(*value=None*)

Create a new ordered dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

move_to_end(*key*, *last=True*)

Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem(*last=True*)

Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*key*, *default=None*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

next()

parser.**read_map**(*filename*)

Read the map file

Parameters **filename** (*str*) – Map filename or path

Returns **rows** – List of the maps read

Return type list

Notes

Note: This functioni is used by the Yolo layer

Utils

utils.**check_is_fitted**(*obj*, *variable='delta'*)

Check if for the current layer is available the backward function.

Parameters

- **obj** (*layer type*) – The object used as self
- **variable** (*str*) – The variable name which allows the backward status if it is not None

Notes

Note: The backward function can be used ONLY after the forward procedure. This function allows to check if the forward function has been already applied.

class utils.**cost_type**(*value*)

Bases: int, enum.Enum

An enumeration.

hellinger = 6

hinge = 7

logcosh = 8

mae = 2

masked = 1

mse = 0

seg = 3

smooth = 4

wgan = 5

utils.**data_to_timesteps**(*data*, *steps*, *shift=1*)

Prepare data for a Recurrent model, dividing a series of data with shape (Ndata, features) into timesteps, with shapes (Ndata - steps + 1, steps, features) If 'data' has more than two dimension, it'll be reshaped. Pay attention to the final number of 'batch'

Parameters

- **data** (*array-like*) – 2 or 4 dimensional numpy array, with shapes (Ndata, features) or (Ndata, w, h, c).
- **steps** (*int*) – Number of timesteps considered for the Recurrent layer
- **shift** (*int (default=1)*) – Temporal shift.

Returns

- **X** (*array-like*) – A view on the data array of input, for Recurrent layers
- **y** (*array-like*) – Correspondig labels as time shifted values.

utils.**from_categorical**(*categoricals*)

Convert a one-hot encoding format into a vector of labels

Parameters **categoricals** (*array-like 2D*) – One-hot encoding format of a label set

Return type Corresponding labels in 1D array

utils.**print_statistics**(*arr*)

Compute the common statistics of the input array

Parameters **arr** (*array-like*) – Input array

Returns

- **mse** (*float*) – Mean Squared Error, i.e sqrt(mean(x*x))
- **mean** (*float*) – Mean of the array
- **variance** (*float*) – Variance of the array

Notes

Note: The values are printed and returned

utils.**to_categorical**(*arr*)

Converts a vector of labels into one-hot encoding format

Parameters **arr** (*array-like 1D*) – Array of integer labels (without holes)

Return type 2D matrix in one-hot encoding format

VideoCapture

class video.**VideoCapture**(*cam_index=0*, *queue_size=128*)

Bases: object

OpenCV VideoCapture wrap in detached thread.

Parameters

- **cam_index** (*integer or str*) – Filename or cam index
- **queue_size** (*int*) – Integer of maximum number of frame to store into the queue

Example

```
>>> cap = VideoCapture()
>>> time.sleep(.1)
>>>
>>> cv2.namedWindow('Camera', cv2.WINDOW_NORMAL)
>>>
>>> cap.start()
>>>
>>> while cap.running():
>>>
>>>   frame = cap.read()
>>>   frame.show('Camera', ms=1)
>>>   print('FPS: {:.3f}'.format(cap.fps))
>>>
>>> cap.stop()
>>>
>>> cv2.destroyAllWindows()
```

Notes

The object is inspired to the ImUtils implementation.

References

- https://github.com/jrosebr1/imutils

property elapsed

Get the elapsed time from start to up to now

Returns **elapsed** – Elapsed time

Return type float

property fps

Get the frame per seconds

Returns **fps** – Frame per seconds

Return type float

`read()`
Get a frame as Image object

Returns **im** – The loaded image

Return type Image obj

`running()`
Check if new frames are available

Returns **running** – True if there are data into the queue, False otherwise

Return type bool

`start()`
Start the video capture in thread

`stop()`
Stop the thread

2.2 References

- Travis Oliphant. "NumPy: A guide to NumPy", USA: Trelgol Publishing, 2006.
- Bradski, G. "The OpenCV Library", Dr. Dobb's Journal of Software Tools, 2000.

PYTHON MODULE INDEX

INDEX

M

N

O

P

R

S

T

U

V

W