pyeddl

Salva Carrión

Jun 18, 2019

USER GUIDE

PyEddl is a Python library that wraps the C++ European Distributed Deep Learning Library (EDDLL). The code is open source, and available on github.

CHAPTER

ONE

INSTALLATION

Pyeddl has a couple of prerequisites that need to be installed first, but once met, the rest of picky requirements are automatically handle by the installer.

1.1 Requirements

- Python3
- CMake 3.14 or higher
- A modern compiler with C++11 support

1.2 Installation

To build and install *pyeddl*, clone or download this repository and then, from within the repository, run:

```
python3 setup.py install
```

or

```
pip3 install .
```

1.3 Tests

To execute all unit tests, run the following command:

```
python3 setup.py test
```

CHAPTER

TWO

FREQUENTLY ASKED QUESTIONS

2.1 Can I contribute to the library?

Yes, please! Indeed, if you don't want, you don't have to worry neither about the widgets nor the documentation if you don't want. The only requirement is you add a new model is that it passes through all the tests.

Fork and contribute all as you want! https://github.com/salvacarrion/EDDL

CHAPTER

THREE

GETTING STARTED

3.1 Multi-layer Perceptron

```python
from pyeddl.layers import Tensor, Input, Dense, Activation, Drop
from pyeddl.models import Model
from pyeddl.datasets import mnist


# Get dataset
(x_train, y_train),(x_test, y_test) = mnist.load_data()

# Input
batch = 1000
in_layer = Input(shape=(batch, 784))

# Layers
l1 = Activation(Dense(in_layer, 1024), 'relu')
l2 = Activation(Dense(l1, 1024), 'relu')
out_layer = Activation(Dense(Drop(l2, 0.5), 10), 'softmax')

m = Model(in_layer, out_layer)

# Plot model
m.plot("model.pdf")

# Get info
m.summary()

# Create optimizer, loss and metric
opt = SGD(lr=0.01, mu=0.9)
losses = [SoftCrossEntropy()]
metrics = [CategoricalAccuracy()]

# Define computing services (CPU, GPU, FPGA)
cs = ComputingService(CPU_threads=4)

# Build network
m.build(opt, losses, metrics, cs)

# Train model
m.fit(x_train, y_train, batch=batch, epochs=1)

# Evaluate model
m.evaluate(x_train, y_train)
```

3.2 Convolutional

```python
from pyeddl.layers import Tensor, Input, Dense, Activation, Drop
from pyeddl.models import Model
from pyeddl.datasets import mnist


# Get dataset
(x_train, y_train),(x_test, y_test) = mnist.load_data()

# Input
batch = 1000
in_layer = Input(shape=(batch, 784))

# Layers
l = Reshape(in_layer, [batch, 1, 28, 28])
l = MaxPool(Activation(Conv(l, 16, [3, 3]), 'relu'), [2, 2])
l = MaxPool(Activation(Conv(l, 32, [3, 3]), 'relu'), [2, 2])
l = MaxPool(Activation(Conv(l, 64, [3, 3]), 'relu'), [2, 2])
l = MaxPool(Activation(Conv(l, 128, [3, 3]), 'relu'), [2, 2])
l = Reshape(l, [batch, -1])
l = Activation(Dense(l, 32), 'relu')
out_tensor = Activation(Dense(l, 10), 'softmax')

m = Model(in_layer, out_layer)

# Plot model
m.plot("model.pdf")

# Get info
m.summary()

# Create optimizer, loss and metric
opt = SGD(lr=0.01, mu=0.9)
losses = [SoftCrossEntropy()]
metrics = [CategoricalAccuracy()]

# Define computing services (CPU, GPU, FPGA)
cs = ComputingService(CPU_threads=4)

# Build network
m.build(opt, losses, metrics, cs)

# Train model
m.fit(x_train, y_train, batch=batch, epochs=1)

# Evaluate model
m.evaluate(x_train, y_train)
```

CHAPTER

FOUR

MODEL (PYEDDL.MODELS)

`Model`	The *Model* class adds training & evaluation routines to a *Network*.

4.1 Model

Model

class `pyeddl.model.`**Model**(*cmodel=None*)

The *Model* class adds training & evaluation routines to a *Network*.

`__init__`(*cmodel=None*)

Initialize self. See help(type(self)) for accurate signature.

`compile`(*optimizer*, *losses=None*, *metrics=None*, *loss_weights=None*, *sample_weight_mode=None*, *weighted_metrics=None*, *target_tensors=None*, *device='cpu'*, ***kwargs*)

Configures the model for training. # Arguments

optimizer: String (name of optimizer) or optimizer instance. loss: String (name of objective function) or objective function.

If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses.

metrics: List of metrics to be evaluated by the model during training and testing. Typically you will use *metrics=['accuracy']*. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as *metrics={'output_a': 'accuracy'}*.

loss_weights: Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the *weighted sum* of all individual losses, weighted by the *loss_weights* coefficients. If a list, it is expected to have a 1:1 mapping to the model's outputs. If a dict, it is expected to map output names (strings) to scalar coefficients.

sample_weight_mode: If you need to do timestep-wise sample weighting (2D weights), set this to *"temporal"*. *None* defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different *sample_weight_mode* on each output by passing a dictionary or a list of modes.

weighted_metrics: List of metrics to be evaluated and weighted by sample_weight or class_weight during training and testing.

target_tensors: By default, Keras will create placeholders for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensors (in turn, Keras will not expect external Numpy data for these targets at training time), you can specify them via the *target_tensors* argument. It can be a single tensor (for a single-output model), a list of tensors, or a dict mapping output names to target tensors.

****kwargs: When using the Theano/CNTK backends, these arguments** are passed into *K.function*. When using the TensorFlow backend, these arguments are passed into *tf.Session.run*.

# Raises

ValueError: In case of invalid arguments for *optimizer*, *loss*, *metrics* or *sample_weight_mode*.

`evaluate`(*x=None*, *y=None*, *batch_size=None*, *verbose=1*, *sample_weight=None*, *steps=None*, *callbacks=None*, *max_queue_size=10*, *workers=1*, *use_multiprocessing=False*)

Returns the loss value & metrics values for the model in test mode. Computation is done in batches. # Arguments

x: Input data. It could be:

- A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
- A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
- None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).

y: Target data. Like the input data *x*, it could be either Numpy array(s), framework-native tensor(s), list of Numpy arrays (if the model has multiple outputs) or None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.

batch_size: Integer or *None*. Number of samples per gradient update. If unspecified, *batch_size* will default to 32. Do not specify the *batch_size* is your data is in the form of symbolic tensors, generators

verbose: 0 or 1. Verbosity mode. 0 = silent, 1 = progress bar.

sample_weight: Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape *(samples, sequence_length)*, to apply a different weight to every timestep of every sample. In this case you should make sure to specify *sample_weight_mode="temporal"* in *compile()*.

steps: Integer or *None*. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of *None*.

callbacks: List of callbacks to apply during evaluation. max_queue_size: Integer. Maximum size for the generator queue.

If unspecified, *max_queue_size* will default to 10.

workers: Integer. Maximum number of processes to spin up when using process-based threading. If unspecified, *workers* will default to 1. If 0, will execute the generator on the main thread.

use_multiprocessing: Boolean. If *True*, use process-based threading. If unspecified, *use_multiprocessing* will default to *False*. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

# Raises ValueError: in case of invalid arguments.

# Returns Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics_names* will give you the display labels for the scalar outputs.

`fit` (*x=None*, *y=None*, *batch_size=None*, *epochs=1*, *verbose=1*, *callbacks=None*, *validation_split=0.0*, *validation_data=None*, *shuffle=True*, *class_weight=None*, *sample_weight=None*, *initial_epoch=0*, *steps_per_epoch=None*, *validation_steps=None*, *validation_freq=1*, *max_queue_size=10*, *workers=1*, *use_multiprocessing=False*, ***kwargs*)

Trains the model for a fixed number of epochs (iterations on a dataset). Arguments:

x: Input data. It could be:

- A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
- A dict mapping input names to the corresponding array/tensors, if the model has named inputs.

y: Target data. Like the input data *x*, it could be either Numpy array(s), framework-native tensor(s), list of Numpy arrays (if the model has multiple outputs) or None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.

batch_size: Integer or *None*. Number of samples per gradient update. If unspecified, *batch_size* will default to 32. Do not specify the *batch_size* if your data is in the form of symbolic tensors, generators, or *Sequence* instances (since they generate batches).

epochs: Integer. Number of epochs to train the model. An epoch is an iteration over the entire *x* and *y* data provided. Note that in conjunction with *initial_epoch*, *epochs* is to be understood as "final epoch". The model is not trained for a number of iterations given by *epochs*, but merely until the epoch of index *epochs* is reached.

verbose: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.

callbacks: List of callbacks to apply during training and validation validation_split: Float between 0 and 1.

Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the *x* and *y* data provided, before shuffling. This argument is not supported when *x* is a generator or *Sequence* instance.

validation_data: Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. *validation_data* will override *validation_split*. *validation_data* could be:

- tuple *(x_val, y_val)* of Numpy arrays or tensors
- tuple *(x_val, y_val, val_sample_weights)* of Numpy arrays
- dataset or a dataset iterator

For the first two cases, *batch_size* must be provided. For the last case, *validation_steps* must be provided.

shuffle: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when *steps_per_epoch* is not *None*.

class_weight: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

sample_weight: Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape *(samples, sequence_length)*, to apply a different weight to every timestep of every sample. In this case you should make sure to specify *sample_weight_mode="temporal"* in *compile()*. This argument is not supported when *x* generator, or *Sequence* instance, instead provide the sample_weights as the third element of *x*.

initial_epoch: Integer. Epoch at which to start training (useful for resuming a previous training run).

steps_per_epoch: Integer or *None*. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default *None* is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined.

validation_steps: Only relevant if *steps_per_epoch* is specified. Total number of steps (batches of samples) to validate before stopping.

validation_steps: Only relevant if *validation_data* is provided and is a generator. Total number of steps (batches of samples) to draw before stopping when performing validation at the end of every epoch.

validation_freq: Only relevant if validation data is provided. Integer or list/tuple/set. If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. *validation_freq=2* runs validation every 2 epochs. If a list, tuple, or set, specifies the epochs on which to run validation, e.g. *validation_freq=[1, 2, 10]* runs validation at the end of the 1st, 2nd, and 10th epochs.

max_queue_size: Integer. Used for generator. Maximum size for the generator queue. If unspecified, *max_queue_size* will default to 10.

workers: Maximum number of processes to spin up when using process-based threading. If unspecified, *workers* will default to 1. If 0, will execute the generator on the main thread.

use_multiprocessing: Boolean. If *True*, use process-based threading. If unspecified, *use_multiprocessing* will default to *False*. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

**kwargs: Used for backwards compatibility.

Returns: A *History* object. Its *History.history* attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises: RuntimeError: If the model was never compiled. ValueError: In case of mismatch between the provided input data

and what the model expects.

`train_on_batch` (*x*, *y*, *sample_weight=None*, *class_weight=None*)
Runs a single gradient update on a single batch of data.

Args:

x: Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.

y: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.

sample_weight: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().

class_weight: Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

Returns: Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute *model.metrics_names* will give you the display labels for the scalar outputs.

CHAPTER

FIVE

CONVOLUTIONAL (PYEDDL.LAYERS.CONV)

`Conv2D`	2D convolution layer (spatial convolution over images).
`Conv2DTranspose`	Transposed convolution layer (sometimes called Deconvolution).

5.1 Conv2D

class pyeddl.layers.conv.**Conv2D**(*filters*, *kernel_size*, *strides=(1, 1)*, *padding='valid'*, *data_format=None*, *dilation_rate=(1, 1)*, *activation=None*, *use_bias=True*, *kernel_initializer='glorot_uniform'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, ***kwargs*)

2D convolution layer (spatial convolution over images).

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If *use_bias* is True, a bias vector is created and added to the outputs. Finally, if *activation* is not *None*, it is applied to the outputs as well. When using this layer as the first layer in a model, provide the keyword argument *input_shape* (tuple of integers, does not include the batch axis), e.g. *input_shape=(128, 128, 3)* for 128x128 RGB pictures in *data_format="channels_last"*.

Args:

filters: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

kernel_size: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

strides: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any *dilation_rate* value != 1.

padding: one of *"valid"* or *"same"* (case-insensitive). data_format: A string,

one of *"channels_last"* or *"channels_first"*. The ordering of the dimensions in the inputs. *"channels_last"* corresponds to inputs with shape *(batch, height, width, channels)* while *"channels_first"* corresponds to inputs with shape *(batch, channels, height, width)*. If you never set it, then it will be "channels_last".

dilation_rate: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any *dilation_rate* value != 1 is incompatible with specifying any stride value != 1.

activation: Activation function to use If you don't specify anything, no activation is applied (ie. "linear" activation: *a(x) = x*).

use_bias: Boolean, whether the layer uses a bias vector. kernel_initializer: Initializer for the *kernel* weights matrix bias_initializer: Initializer for the bias vector kernel_regularizer: Regularizer function applied to

the *kernel* weights matrix

bias_regularizer: Regularizer function applied to the bias vector activity_regularizer: Regularizer function applied to

the output of the layer (its "activation").

kernel_constraint: Constraint function applied to the kernel matrix bias_constraint: Constraint function applied to the bias vector

Input shape: 4D tensor with shape: *(batch, channels, rows, cols)* if *data_format* is *"channels_first"* or 4D tensor with shape: *(batch, rows, cols, channels)* if *data_format* is *"channels_last"*.

Output shape: 4D tensor with shape: *(batch, filters, new_rows, new_cols)* if *data_format* is *"channels_first"* or 4D tensor with shape: *(batch, new_rows, new_cols, filters)* if *data_format* is *"channels_last"*. *rows* and *cols* values might have changed due to padding.

`__init__`(*filters*, *kernel_size*, *strides=(1, 1)*, *padding='valid'*, *data_format=None*, *dilation_rate=(1, 1)*, *activation=None*, *use_bias=True*, *kernel_initializer='glorot_uniform'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

5.2 Conv2DTranspose

class `pyeddl.layers.conv.`**`Conv2DTranspose`**(*filters*, *kernel_size*, *strides=(1, 1)*, *padding='valid'*, *output_padding=None*, *data_format=None*, *dilation_rate=(1, 1)*, *activation=None*, *use_bias=True*, *kernel_initializer='glorot_uniform'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, ***kwargs*)
Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution. When using this layer as the first layer in a model, provide the keyword argument *input_shape* (tuple of integers, does not include the batch axis), e.g. *input_shape=(128, 128, 3)* for 128x128 RGB pictures in *data_format="channels_last"*.

Args:

filters: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

kernel_size: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

strides: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any *dilation_rate* value != 1.

padding: one of *"valid"* or *"same"* (case-insensitive). output_padding: An integer or tuple/list of 2 integers,

specifying the amount of padding along the height and width of the output tensor. Can be a single integer to specify the same value for all spatial dimensions. The amount of output padding along a given dimension must be lower than the stride along that same dimension. If set to *None* (default), the output shape is inferred.

data_format: A string, one of *"channels_last"* or *"channels_first"*. The ordering of the dimensions in the inputs. *"channels_last"* corresponds to inputs with shape *(batch, height, width, channels)* while *"channels_first"* corresponds to inputs with shape *(batch, channels, height, width)*. It defaults to the *image_data_format* value found in your

dilation_rate: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any *dilation_rate* value != 1 is incompatible with specifying any stride value != 1.

activation: Activation function to use If you don't specify anything, no activation is applied (ie. "linear" activation: *a(x) = x*).

use_bias: Boolean, whether the layer uses a bias vector. kernel_initializer: Initializer for the *kernel* weights matrix bias_initializer: Initializer for the bias vector kernel_regularizer: Regularizer function applied to

the *kernel* weights matrix

bias_regularizer: Regularizer function applied to the bias vector activity_regularizer: Regularizer function applied to

the output of the layer (its "activation").

kernel_constraint: Constraint function applied to the kernel matrix bias_constraint: Constraint function applied to the bias vector

Input shape: 4D tensor with shape: *(batch, channels, rows, cols)* if *data_format* is *"channels_first"* or 4D tensor with shape: *(batch, rows, cols, channels)* if *data_format* is *"channels_last"*.

Output shape: 4D tensor with shape: *(batch, filters, new_rows, new_cols)* if *data_format* is *"channels_first"* or 4D tensor with shape: *(batch, new_rows, new_cols, filters)* if *data_format* is *"channels_last"*. *rows* and *cols* values might have changed due to padding. If *output_padding* is specified: ``‘ new_rows = ((rows - 1) * strides[0] + kernel_size[0]

- 2 * padding[0] + output_padding[0])

new_cols = ((cols - 1) * strides[1] + kernel_size[1]

- 2 * padding[1] + output_padding[1])

‘‘‘

References

- **[A guide to convolution arithmetic for deep learning](** https://arxiv.org/abs/1603.07285v1)
- **[Deconvolutional Networks](** https://www.matthewzeiler.com/mattzeiler/deconvolutionalnetworks.pdf)

`__init__`(*filters*, *kernel_size*, *strides=(1, 1)*, *padding='valid'*, *output_padding=None*, *data_format=None*, *dilation_rate=(1, 1)*, *activation=None*, *use_bias=True*, *kernel_initializer='glorot_uniform'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

CHAPTER

SIX

CORE (PYEDDL.LAYERS.CORE)

`Activation`	Applies an activation function to an output.
`Dense`	Just your regular densely-connected NN layer.
`Dropout`	Applies Dropout to the input.
`Input`	Layer to be used as an entry point into a model.
`Reshape`	Reshapes an output to a certain shape.

6.1 Activation

class `pyeddl.layers.core.`**`Activation`**(*activation*, ***kwargs*)

Applies an activation function to an output.

Args: activation: name of activation function to use

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.

`__init__`(*activation*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

6.2 Dense

class `pyeddl.layers.core.`**`Dense`**(*units*, *activation=None*, *use_bias=True*, *kernel_initializer='glorot_uniform'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, ***kwargs*)

Just your regular densely-connected NN layer.

Dense implements the operation: *output = activation(dot(input, kernel) + bias)* where *activation* is the element-wise activation function passed as the *activation* argument, *kernel* is a weights matrix created by the layer, and *bias* is a bias vector created by the layer (only applicable if *use_bias* is *True*). Note: if the input to the layer has a rank greater than 2, then it is flattened prior to the initial dot product with *kernel*.

Example: # as first layer in a sequential model: model = Sequential() model.add(Dense(32, input_shape=(16,))) # now the model will take as input arrays of shape (*, 16) # and output arrays of shape (*, 32) # after the first layer, you don't need to specify # the size of the input anymore: model.add(Dense(32))

Args: units: Positive integer, dimensionality of the output space. activation: Activation function to use

If you don't specify anything, no activation is applied (ie. "linear" activation: *a(x) = x*).

use_bias: Boolean, whether the layer uses a bias vector. kernel_initializer: Initializer for the *kernel* weights matrix bias_initializer: Initializer for the bias vector kernel_regularizer: Regularizer function applied to

the *kernel* weights matrix

bias_regularizer: Regularizer function applied to the bias vector activity_regularizer: Regularizer function applied to

the output of the layer (its "activation").

kernel_constraint: Constraint function applied to the *kernel* weights matrix

bias_constraint: Constraint function applied to the bias vector

Input shape: nD tensor with shape: *(batch_size, ..., input_dim)*. The most common situation would be a 2D input with shape *(batch_size, input_dim)*.

Output shape: nD tensor with shape: *(batch_size, ..., units)*. For instance, for a 2D input with shape *(batch_size, input_dim)*, the output would have shape *(batch_size, units)*.

`__init__`(*units*, *activation=None*, *use_bias=True*, *kernel_initializer='glorot_uniform'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

6.3 Dropout

class `pyeddl.layers.core.`**`Dropout`**(*rate*, *noise_shape=None*, *seed=None*, ***kwargs*)
Applies Dropout to the input.

Dropout consists in randomly setting a fraction *rate* of input units to 0 at each update during training time, which helps prevent overfitting.

Args: rate: float between 0 and 1. Fraction of the input units to drop. noise_shape: 1D integer tensor representing the shape of the

binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape *(batch_size, timesteps, features)* and you want the dropout mask to be the same for all timesteps, you can use *noise_shape=(batch_size, 1, features)*.

seed: A Python integer to use as random seed.

References

- **[Dropout: A Simple Way to Prevent Neural Networks from Overfitting]**(http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf)

`__init__`(*rate*, *noise_shape=None*, *seed=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

6.4 Input

class `pyeddl.layers.core.`**`Input`**(*input_shape=None*, *batch_size=None*, *batch_input_shape=None*, *dtype=None*, *input_tensor=None*, *sparse=False*, *name=None*)
Layer to be used as an entry point into a model.

It can either wrap an existing tensor (pass an *input_tensor* argument) or create its a placeholder tensor (pass arguments *input_shape* or *batch_input_shape* as well as *dtype*).

Args: input_shape: Shape tuple, not including the batch axis. batch_size: Optional input batch size (integer or None). batch_input_shape: Shape tuple, including the batch axis. dtype: Datatype of the input. input_tensor: Optional tensor to use as layer input

instead of creating a placeholder.

sparse: Boolean, whether the placeholder created is meant to be sparse.

name: Name of the layer (string).

`__init__`(*input_shape=None*, *batch_size=None*, *batch_input_shape=None*, *dtype=None*, *input_tensor=None*, *sparse=False*, *name=None*)
Initialize self. See help(type(self)) for accurate signature.

6.5 Reshape

class `pyeddl.layers.core.`**`Reshape`**(*target_shape*, ***kwargs*)
Reshapes an output to a certain shape.

Args:

target_shape: target shape. Tuple of integers. Does not include the batch axis.

Input shape: Arbitrary, although all dimensions in the input shaped must be fixed. Use the keyword argument *input_shape* (tuple of integers, does not include the batch axis) when using this layer as the first layer in a model.

Output shape: *(batch_size,) + target_shape*

# Example ‘‘‘python

# as first layer in a Sequential model model = Sequential() model.add(Reshape((3, 4), input_shape=(12,))) # now: model.output_shape == (None, 3, 4) # note: *None* is the batch dimension # as intermediate layer in a Sequential model model.add(Reshape((6, 2))) # now: model.output_shape == (None, 6, 2) # also supports shape inference using *-1* as dimension model.add(Reshape((-1, 2, 2))) # now: model.output_shape == (None, 3, 2, 2)

‘‘‘

`__init__`(*target_shape*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

CHAPTER SEVEN

MERGE (PYEDDL.LAYERS.MERGE)

Add	Layer that adds a list of inputs.
Concatenate	Layer that concatenates a list of inputs.

7.1 Add

class pyeddl.layers.merge.**Add**(***kwargs*)

Layer that adds a list of inputs.

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

Example:

‘‘‘**python** import pyeddl input1 = pyeddl.layers.Input(shape=(16,)) x1 = pyeddl.layers.Dense(8, activation=’relu’)(input1) input2 = pyeddl.layers.Input(shape=(32,)) x2 = pyeddl.layers.Dense(8, activation=’relu’)(input2) # equivalent to added = pyeddl.layers.add([x1, x2]) added = pyeddl.layers.Add()([x1, x2]) out = pyeddl.layers.Dense(4)(added) model = pyeddl.models.Model(inputs=[input1, input2], outputs=out)

‘‘‘

7.2 Concatenate

class pyeddl.layers.merge.**Concatenate**(*axis=-1*, ***kwargs*)

Layer that concatenates a list of inputs.

It takes as input a list of tensors, all of the same shape except for the concatenation axis, and returns a single tensor, the concatenation of all inputs.

Args: axis: Axis along which to concatenate. **kwargs: standard layer keyword arguments.

__init__(*axis=-1*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

CHAPTER

EIGHT

POOLING (PYEDDL.LAYERS.POOLING)

MaxPooling2D	Max pooling operation for spatial data.
AveragePooling2D	Average pooling operation for spatial data.

8.1 MaxPooling2D

class pyeddl.layers.pooling.**MaxPooling2D** (*pool_size=(2, 2)*, *strides=None*, *padding='valid'*, *data_format=None*, ***kwargs*)

Max pooling operation for spatial data.

Args:

pool_size: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.

strides: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to *pool_size*.

padding: One of *"valid"* or *"same"* (case-insensitive). data_format: A string,

one of *channels_last* (default) or *channels_first*. The ordering of the dimensions in the inputs. *channels_last* corresponds to inputs with shape *(batch, height, width, channels)* while *channels_first* corresponds to inputs with shape *(batch, channels, height, width)*. It defaults to the *image_data_format* value found in your

Input shape:

- **If *data_format='channels_last'*:** 4D tensor with shape: *(batch_size, rows, cols, channels)*
- **If *data_format='channels_first'*:** 4D tensor with shape: *(batch_size, channels, rows, cols)*

Output shape:

- **If *data_format='channels_last'*:** 4D tensor with shape: *(batch_size, pooled_rows, pooled_cols, channels)*
- **If *data_format='channels_first'*:** 4D tensor with shape: *(batch_size, channels, pooled_rows, pooled_cols)*

__init__ (*pool_size=(2, 2)*, *strides=None*, *padding='valid'*, *data_format=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

8.2 AveragePooling2D

class `pyeddl.layers.pooling.`**`AveragePooling2D`**(*pool_size=(2, 2)*, *strides=None*, *padding='valid'*, *data_format=None*, ***kwargs*)

Average pooling operation for spatial data.

Args:

pool_size: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.

strides: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to *pool_size*.

padding: One of *"valid"* or *"same"* (case-insensitive). data_format: A string,

one of *channels_last* (default) or *channels_first*. The ordering of the dimensions in the inputs. *channels_last* corresponds to inputs with shape *(batch, height, width, channels)* while *channels_first* corresponds to inputs with shape *(batch, channels, height, width)*.

Input shape:

- **If *data_format='channels_last'*:** 4D tensor with shape: *(batch_size, rows, cols, channels)*
- **If *data_format='channels_first'*:** 4D tensor with shape: *(batch_size, channels, rows, cols)*

Output shape:

- **If *data_format='channels_last'*:** 4D tensor with shape: *(batch_size, pooled_rows, pooled_cols, channels)*
- **If *data_format='channels_first'*:** 4D tensor with shape: *(batch_size, channels, pooled_rows, pooled_cols)*

`__init__`(*pool_size=(2, 2)*, *strides=None*, *padding='valid'*, *data_format=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

CHAPTER

NINE

LOSSES (`PYEDDL.LOSSES`)

Built-in loss functions.

mean_squared_error	Mean Squared Error
categorical_crossentropy	Categorical Cross-Entropy
categorical_soft_crossentropy	Categorical Soft Cross-Entropy

9.1 Mean Squared Error

`pyeddl.losses.`**`mean_squared_error`**`()`

Mean Squared Error

9.2 Cross Entropy

`pyeddl.losses.`**`categorical_crossentropy`**`()`

Categorical Cross-Entropy

9.3 Soft Cross Entropy

`pyeddl.losses.`**`categorical_soft_crossentropy`**`()`

Categorical Soft Cross-Entropy

CHAPTER

TEN

METRICS (PYEDDL.METRICS)

mean_squared_error	Mean Squared Error
categorical_accuracy	Categorical Accuracy

10.1 Mean Squared Error

pyeddl.metrics.**mean_squared_error**()

Mean Squared Error

10.2 Categorical accuracy

pyeddl.metrics.**categorical_accuracy**()

Categorical Accuracy

CHAPTER

ELEVEN

OPTIMIZERS (PYEDDL.OPTIM)

The classes presented in this section are optimizers to modify the SGD updates during the training of a model.

The update functions control the learning rate during the SGD optimization

SGD	Stochastic gradient descent optimizer.

11.1 Stochastic Gradient Descent

This is the optimizer by default in all models.

class pyeddl.optim.**SGD**(*lr=0.01*, *momentum=0.0*, *decay=0.0*, *nesterov=False*, ***kwargs*)

Stochastic gradient descent optimizer.

Includes support for momentum, learning rate decay, and Nesterov momentum.

Args: lr: float >= 0. Learning rate. momentum: float >= 0. Parameter that accelerates SGD

in the relevant direction and dampens oscillations.

decay: float >= 0. Learning rate decay over each update. nesterov: boolean. Whether to apply Nesterov momentum.

__init__(*lr=0.01*, *momentum=0.0*, *decay=0.0*, *nesterov=False*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

CHAPTER

TWELVE

DATASETS (`PYEDDL.DATASETS`)

mnist	MNIST handwritten digits dataset.
cifar10	CIFAR10 small images classification dataset.

12.1 MNIST

MNIST handwritten digits dataset.

`pyeddl.datasets.mnist.`**`load_data`**`(`*path='mnist.npz'*`)`

Loads the MNIST dataset.

Arguments: path: path where to cache the dataset locally

Returns: Tuple of Numpy arrays: *(x_train, y_train), (x_test, y_test)*.

License: Yann LeCun and Corinna Cortes hold the copyright of MNIST dataset, which is a derivative work from original NIST datasets. MNIST dataset is made available under the terms of the [Creative Commons Attribution-Share Alike 3.0 license.](https://creativecommons.org/licenses/by-sa/3.0/)

12.2 CIFAR-10

CIFAR10 small images classification dataset.

`pyeddl.datasets.cifar10.`**`load_data`**`()`

Loads CIFAR10 dataset.

Returns: Tuple of Numpy arrays: *(x_train, y_train), (x_test, y_test)*.

CHAPTER THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX