Python Package for Building Deep Spiking Convolutional Neural Networks on SpiNNaker - Documentation

Release 1.0

Sven Gronauer

Mar 08, 2018

CONTENTS:

1 Introduction **1**
- 1.1 Requirements . . . 1

2 Sample Code **3**
- 2.1 Simple 1-Layer ConvNet . . . 3
- 2.2 Deeper ConvNet . . . 3

3 SpikingConvNet **5**
- 3.1 SpikingConvNet package . . . 5

4 Miscellaneous **13**
- 4.1 Build documentation . . . 13
- 4.2 Obtain MNIST-Dataset . . . 13
- 4.3 Clean up directory . . . 14

Python Module Index **15**

Index **17**

CHAPTER

ONE

INTRODUCTION

This Python Package offers a convenient way of defining a Spiking Convolutional Neural Network on SpiNNaker Hardware. The network, however, is trained in unsupervised manner layer by layer.

1.1 Requirements

- Python 2.7.14
- sPyNNaker 8 (1!4.0.0)

CHAPTER

TWO

SAMPLE CODE

This section describes how to setup a simple Deep Spiking Convolutional Neural Network.

2.1 Simple 1-Layer ConvNet

Let's start with training a simple SCNN with one convolutional Layer. By creating firstly the model structure with the following python code:

```
model = SpikingModel(input_tensor=(28,28,1), run_control=rc)
model.add(ConvLayer(4,shape=(5,5), stride=2))
model.add(Classifier())
```

In order to build the network structure on SpiNNaker Hardware, you have to execute commands in the terminal:

```
$python main.py --mode loaddata
$python main.py --mode training --layer 1
$python main.py --mode training --layer svm
$python main.py --mode testing
```

2.2 Deeper ConvNet

Theoritically, as many layers as appreciated can be build. Therefore convolutional layers are added to the model are added in sequential manner.

```
model = SpikingModel(input_tensor=(28,28,1), run_control=rc)
model.add(ConvLayer(4,shape=(5,5), stride=2))
model.add(ConvLayer(4,shape=(5,5), stride=2))
...
model.add(ConvLayer(4,shape=(3,3), stride=2))
model.add(Classifier())
```

```
$python main.py --mode loaddata
$python main.py --mode training --layer 1
$python main.py --mode training --layer 2
...
$python main.py --mode training --layer n
$python main.py --mode training --layer svm
$python main.py --mode testing
```

Note: The training of the Network is done layer by layer, hence the input spikes of the currently trained layer depend on the previous layer. So a new simulation cycle is started the previously calculated layers are flattend to achieve parallel computation.

CHAPTER

THREE

SPIKINGCONVNET

3.1 SpikingConvNet package

3.1.1 Module contents

Package for building Spiking Deep Convolutional Neural Networks on SpiNNaker

- Neuroscientific System Theory
- Technical University Munich
- Creator: Sven Gronauer
- Date: February 2018

contains the necessary infrastructure and algorithms to build an arbitrarily deep network described as a sequential model. The package SpikingConvNet is divided into three modules, which will be proposed in the following:

`Classes Module`

contains the object classes for creating models of the Spiking Neural Networks and the necessary infrastructure for implementing such networks on SpiN- Naker. A SCNN model is built by sequentially adding of convolutional layers to the input layer and a classifier at last. The Spinnaker Network class holds instances and methods to interact with the SpiNNaker board. The structure is based on PyNNs procedure of creating Spiking Networks. Neurons belonging to a particular layer of the neural net- work and share common properties are packed into populations. Projections establish connections between populations. The strength of a connection is expressed by a numerical value, the synaptic weight that either increases the membrane potential of the post-neuron (exhibitory) or lessens the membrane potential (inhibitory). The model is defined on the host-computer and then transferred to the SpiN- Naker board, where the simulation is processed. When the simulation finishes, data are retrieved back to the host-computer and post-processed.

`Algorithms Module`

provides functions for generating sparse connections between neural popula- tions. The so-called projections between populations are employed by con- nection lists, which are automatically built by considering the tensors of the layers to be connected. Stride and kernel size specify the particular tensor of the posterior layer by T_n+1 = (T_n - shape)/stride + 1. In addition, the algorithms module supports the transformation of spiketrains. Training deeper layers requires the network of the previous layers to be rebuilt to obtain previous spiketrains. These spiketrains are transformed into plain times to assign them to a spiking source array and train the actual layer with the STDP update rule by windowing the input pattern over several time in- stances. The training of the network is done layer by layer, hence the input spikes of the currently trained layer depend on the previous layer. So a new simulation cycle is started when a deeper layer is trained. Because the synap- tic weights of previous layers are already determined, the previous layers are rebuild as parallel entities and therefore profit in terms of computational speed.

`Parameters Module`

This file holds all important constants and parameters for controlling and setting up the simulation.

`Utils Module`

Supporting functions to visualise and handle data. The training and test set of the Spiking Neural Network is obtained by the MNIST dataset. Loading a specified subset of MNIST digits defined in parameters.py and supplies the network algorithms a shuffled set of data points. Processed data is visualised with the plotting functions: convolutional kernels can be plotted with heatmaps, spike times of each neuron along the time axis and the membrane voltages over time.

3.1.2 Submodules

3.1.3 SpikingConvNet.algorithms module

Algorithms are provided for

- Generating sparse connections between populations
- Slice a layer into windows and split windows over time

`SpikingConvNet.algorithms.`**`input_flattend_spikes`**(*X_train*, *tensor_input*, *kernel_shape*)

Create flattened SpikeSourceArray for input neurons for instance in rebuilding network

For rebuilding the network structure the input neurons are not windowed over time. Instead, the input layer is flattened and the entire image is presented to the network in each simulation interval. The corresponding spiketimes depend on pixel intensities and are stochastically rate-coded.

Parameters

- **X_train** (*np.array, shape = [n_examples, image_intensities.flatten()]*) – dataset of MNIST input images as 2d-array
- **tensor_input** (*tuple of int*) – Dimensions of input layer
- **kernel_shape** (*tuple of int*) – Kernel shape

Returns **spiketrains** – Each datapoint contains precise spike times for each input neuron

Return type np.array, shape = [n_examples, spike_times]

`SpikingConvNet.algorithms.`**`input_windowed_spikes`**(*X_train*, *tensor_input*, *kernel_shape*, *stride*)

Create windowed SpikeSourceArray for input neurons for instance in training layers of the network

Input patterns are windowed over time, post-neurons are presented only a subset of the input pattern. The corresponding spiketimes depend on pixel intensities and are stochastically rate-coded.

Parameters **X_train** (*np.array, shape = [n_examples, image_intensities.flatten()]*)

Returns **spiketrains** – Each datapoint contains precise spike times for each input neuron

Return type np.array, shape = [n_examples*n_windows, spike_times]

`SpikingConvNet.algorithms.`**`rebuild_fixed_connections`**(*tensor_first*, *tensor_second*, *kernel_shape*, *stride*, *weights_tensor*)

Construct fixed connections between flattened layers

Take previously learned STDP weights and establish fixed weights. The computation is handled now in parallel

Parameters

- **tensor_first** (*tuple of int*) – Dimensions of previous layer
- **tensor_second** (*tuple of int*) – Dimensions of posterior layer
- **kernel_shape** (*tuple of int*) – Kernel shape
- **stride** (*int*) – Specified stride over convolved layer
- **weights_tensor** (*np.array, shape=[n_kernel, kernel_height*kernel_width]*) – Previously trained STDP weights, now initialized as fixed weights

Returns **connection_list** – list for s.FromListConnector()

Return type list of [position_1, position_2, weight , delay]

SpikingConvNet.algorithms.**rebuild_inhibitory_connections**(*tensor_prev*, *tensor_layer*, *inhib_weight*)

Construct inhibitory connections within flattened layers

Parameters

- **tensor_prev** (*tuple of int*) – Dimensions of previous layer
- **tensor_layer** (*tuple of int*) – Dimensions of actual layer
- **inhib_weight** (*float32*) – fixed weight for inhibitory connection

Returns **inhib_connection_list** – list for s.FromListConnector()

Return type list of [position_1, position_2, weight , delay]

SpikingConvNet.algorithms.**spikes_for_classifier**(*rc*, *tensor*, *spiketrains*)

Transform spiketrains to plain two-dimensional dataset

to reduce the power of Support Vector Machine, the quantity of spikes within each simulation interval is counted for each neuron in the last layer

Parameters

- **tensor** (*tuple of int*) – Tensor of last Convolutional layer in network
- **spiketrains** (*SpikeTrain object*) – Retrieved spikes from last layer on SpiNNaker board SpikeTrains objects are extracted from Neo Block Segments

Returns **X** – Each datapoint contains number of spikes for each post-neuron within one sim interval

Return type np.array, shape = [datapoints, n_neurons_last_layer]

SpikingConvNet.algorithms.**windowed_spikes**(*spiketrains_input*, *tensor_first*, *tensor_second*, *kernel_shape*, *stride*)

Create windowed SpikeSourceArray for instance in training deeper layers of the network

Input spiketrains are windowed over time, post-neurons are presented only a subset of the input spiketrains. The corresponding output spiketimes depend on calculated spikes (spiketrains_input) of previous layer.

Parameters

- **spiketrains_input** (*SpikeTrain object*) – Spiketrains from previous layer
- **tensor_first** (*tuple of int*) – Dimensions of previous layer
- **tensor_second** (*tuple of int*) – Dimensions of posterior layer
- **kernel_shape** (*tuple of int*) – Kernel shape
- **stride** (*int*) – Specified stride over convolved layer

Returns **spiketrains** – Each datapoint contains precise spike times for each neuron

Return type np.array, shape = [n_examples*windows, spike_times]

3.1.4 SpikingConvNet.classes module

This module provides classes for creating objects of the neural network model and the necessary infracture for building networks on SpiNNaker

Classes are namely:

- Layer
- InputLayer(Layer)
- ConvLayer(Layer)
- Classifier(Layer)
- SpikingModel
- Spinnaker_Network

class SpikingConvNet.classes.**Classifier**
Bases: *SpikingConvNet.classes.Layer*

Holds a linear Support Vector Machine for classifying

predict(*X_test*, *y_test*)
Determine classification accuracy of SVC with given Testset

train(*X_train*, *y_train*)
Train parameters of SVM model with given Trainset

class SpikingConvNet.classes.**ConvLayer**(*kernels*, *shape*, *stride*)
Bases: *SpikingConvNet.classes.Layer*

Specify a convolutional layer of the network

class SpikingConvNet.classes.**InputLayer**(*tensor*)
Bases: *SpikingConvNet.classes.Layer*

Specifies input layer of network

class SpikingConvNet.classes.**Layer**(*rc*)
Bases: object

Base class for network layers

class SpikingConvNet.classes.**SpikingModel**(*input_tensor*, *run_control=None*)
Bases: object

Describes structure of spiking neural network

A model consists of sequential topology of layers. The first layer is provided as an input layer, followed by convolutional layers. The last layer should be defined by a Classifier.

Parameters

- **input_tensor** (*tuple of int*) – Size of input patterns as 3d-Tensor
- **run_control** (*RunControl object*) – Holds information of program flow

add(*layer*)
Adds sequentially a layer to the network model

print_structure()
Print struture of model in console

class SpikingConvNet.classes.**Spinnaker_Network**(*runcontrol*, *model*, *deepspikes=None*)
Bases: object

Class for implementing neural network model on SpiNNaker

The following steps are processed through calling the class constructor (based on PyNN basic setup structure)

1. Initialize with constructor
2. Load datasets (Train and Testset) from local files
3. Load previously calculated weights for layers (if given in /model)
4. Create populations
5. Build STDP-model
6. Build projections between populations
7. Setup recordings

The following methods must be called from external fuction(s):

- update_kernel_weights() - Determine current weights in STDP trained layer
- retrieve_data() - Receive recorded data from SpiNNaker
- print_parameters() - Display information of neural network

Parameters

- **runcotrol** (*RunControl object*) – Structure that contains basic information for program flow such as passed args from terminal command, backup commands, building options for SpiNNaker network
- **model** (*SpikingModel object*) – predefined model of spiking neural network
- **deepspikes** (*Spiketrain object*) – training a deeper layer requires preprocessed spikes from previous layer, hence training of Spiking Neural Network is done layer by layer

print_parameters()
Print parameters of model and simulation to console

retrieve_data()
Transmit observed data of spikes and voltages from SpiNNaker Board to host computer

Returns

- **spiketrains** (*SpikeTrain object*) – Spiketrains from last layer in neural network
- **list** (*[spikes_in, spikes_1, v_1]*) –
  - spikes_in: spiketimes input layer
  - spikes_1: spikes post-neurons
  - v_1: membrane potentials of post-neurons

update_kernel_weights()
Update the internal variables of weights

Receive actual weight values of trained STDP layer from SpiNNaker board and store to self.w_layer variable

Returns

Return type weights, np.array, shape = [n_kernels, flattend_weights]

3.1.5 SpikingConvNet.parameters module

This file holds all important constans and parameters for controlling the simulation

Note: Any adjustments applied to this file have an impact to all other files in the project!

3.1.6 SpikingConvNet.utils module

Utilities for controling program flow, data handling and data plotting

class SpikingConvNet.utils.**RunControl**(*args*, *trainlayer=0*, *trainsvm=False*, *rebuild=False*)
Bases: object

Object for controlling program flow, contains args from console and sets up the logging utility

Parameters

- **args** (*ArgumentParser object*) – Passed arguments from terminal command
- **trainlayer** (*int, optional*) – If not zero, specifies which layer of network to train
- **trainsvm** (*bool, optional*) – If given, classifier is trained
- **rebuild** (*bool, optional*) – Controls the behaviour of the follow up build of neural network (as a variable of programs state machine)
  - `rebuild==True` in order to train layer n, the spikes of layer n-1 must be determined
  - `rebuild==False` a layer or the Classifier is trained

setup_logger()
Setup logger for tracking infos and errors

SpikingConvNet.utils.**convert_rate_code**(*intensity*, *total_intensity=None*)
Rate code spikes - Assign pixel intensity to stochastic spike times

calculate spike times dependend on pixel intensity of pre-neuron

Parameters

- **intensity** (*int, [0,255]*) – Pixel intensity
- **total_intensity** (*int*) – Sum of intensities of input pattern for normalization

Returns **spiketimes** – Corresponding spike times for pixel intensity

Return type list, of int

SpikingConvNet.utils.**convert_time_code**(*intensity*)
Time code spikes - Assign pixel intensity to deterministic time interval

Parameters **intensity** (*int, [0,255]*) – Pixel intensity

Returns **spiketime** – Corresponding spike time for pixel intensity

Return type int

SpikingConvNet.utils.**dog_filter**(*image*)
Apply Difference of Gaussian Filter to image

Parameters **image** (*np.array, shape=[height, width]*) – Image to be transformed

Returns **norm_dog** – Transformed image

Return type np.array, shape=[height, width]

SpikingConvNet.utils.**load_MNIST_digits_shuffled**(*rc*, *mode*)

Load MNIST dataset

load defined number of examples (see parameters.py) loaded subset of digits is defined in SUBSET_DIGITS shuffle data and return as 2d-arrays

Parameters

- **rc** (*RunControl object*) – contains information of backup behaviour
- **mode** (*str*) –
  - `train_examples` Examples for Trainset are loaded
  - `test_examples` Examples for Testset are loaded

Returns **f** – Object of figure

Return type figure object

SpikingConvNet.utils.**plot_confusion_matrix**(*rc*, *cm*, *normalize=True*, *title='Confusion_matrix'*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*)

This function prints and plots a confusion matrix. Normalization can be applied by setting `normalize=True`.

Parameters

- **rc** (*RunControl object*) – contains information of backup behaviour
- **cm** (*np.array,*) – Contains values of the confusion matrix
- **normalize** (*bool*) – If `True` Matrix is normalized
- **title** (*str*) – defines title of figure and name of saved figure on disk
- **cmap** (*cm object*) – Color scheme of the plot

Returns **f** – Object of figure

Return type figure object

SpikingConvNet.utils.**plot_heatpmap**(*rc*, *list_of_elements*, *title='Default Title'*, *delta=False*)

Plot a matrix of heatmaps

Parameters

- **rc** (*RunControl object*) – contains information of backup behaviour
- **list_of_elements** (*list*) – List of np.arrays with shape=[n,n]
- **title** (*str*) – defines title of figure and name of saved figure on disk
- **delta** (*bool*) – if defined, use dirrential color scheme for heatmap plot

Returns **f** – Object of heatmap

Return type figure object

SpikingConvNet.utils.**plot_membrane_voltages**(*rc*, *v_data*, *simtime*, *title='Membrane potentials'*)

Plot the membrane potential

Parameters

- **rc** (*RunControl object*) – contains information of backup behaviour

- **v_data** (*Voltage Object*) – Contains values of the membrane potentials
- **simtime** (*int*) – Maximum simulation time on the x-axis
- **title** (*str*) – defines title of figure and name of saved figure on disk

Returns **f** – Object of figure

Return type figure object

SpikingConvNet.utils.**plot_spike_activity**(*rc*, *spiketrains*, *tensor*, *title='plot_spike_activity'*)

Plot the activity of each neurons in the given layer in the first simulation interval

The quantity of spikes for each neuron in the layer (given with tensor and its spiketrains) are determined for the first simulation interval and then plotted as heatmap

Parameters

- **rc** (*RunControl object*) – contains information of backup behaviour
- **spiketrains** (*SpikeTrain Object*) – List of np.arrays with shape=[n,n]
- **tensor** (*tuple of int*) – Tensor of Layer
- **title** (*str*) – defines title of figure and name of saved figure on disk

Returns **f** – Object of figure

Return type figure object

SpikingConvNet.utils.**plot_spikes**(*rc*, *pre*, *post=None*, *title='Spikes Plot'*, *path=None*)

Plot spikes of given layers

Parameters

- **rc** (*RunControl object*) – contains information of backup behaviour
- **pre** (*SpikeTrain object*) – Spiketrains of first layer to plot

Returns **norm_dog** – Transformed image

Return type np.array, shape=[height, width]

CHAPTER

FOUR

MISCELLANEOUS

4.1 Build documentation

How to build the documentation:

1. Change directory:

```
$ cd docs/
```

2. If major changes have been submitted to the code, then render:

```
$ sphinx-apidoc -f -o source/ ../SpikingConvNet/
```

3. Execute to create HTML Documentation in `/build/html` directory:

```
$ make html
```

4. Excute to create PDF of Documentation in `/build/latex` directory:

```
$ make latexpdf
```

4.2 Obtain MNIST-Dataset

thanks to *https://pypi.python.org/pypi/python-mnist*

Get the package from PyPi:

```
$ pip install python-mnist
```

or install with `setup.py`:

```
$ cd python-mnist/
$ python setup.py install
```

Code sample:

```
from mnist import MNIST
mndata = MNIST('./dir_with_mnist_data_files')
images, labels = mndata.load_training()
```

4.3 Clean up directory

Delete all .pyc files with command:

```
$ bash cleanup.sh
```

PYTHON MODULE INDEX

S

SpikingConvNet, 5
SpikingConvNet.algorithms, 6
SpikingConvNet.classes, 8
SpikingConvNet.parameters, 10
SpikingConvNet.utils, 10

INDEX

A

add() (SpikingConvNet.classes.SpikingModel method), 8

C

Classifier (class in SpikingConvNet.classes), 8
convert_rate_code() (in module SpikingConvNet.utils), 10
convert_time_code() (in module SpikingConvNet.utils), 10
ConvLayer (class in SpikingConvNet.classes), 8

D

dog_filter() (in module SpikingConvNet.utils), 10

I

input_flattend_spikes() (in module SpikingConvNet.algorithms), 6
input_windowed_spikes() (in module SpikingConvNet.algorithms), 6
InputLayer (class in SpikingConvNet.classes), 8

L

Layer (class in SpikingConvNet.classes), 8
load_MNIST_digits_shuffled() (in module SpikingConvNet.utils), 11

P

plot_confusion_matrix() (in module SpikingConvNet.utils), 11
plot_heatpmap() (in module SpikingConvNet.utils), 11
plot_membrane_voltages() (in module SpikingConvNet.utils), 11
plot_spike_activity() (in module SpikingConvNet.utils), 12
plot_spikes() (in module SpikingConvNet.utils), 12
predict() (SpikingConvNet.classes.Classifier method), 8
print_parameters() (SpikingConvNet.classes.Spinnaker_Network method), 9
print_structure() (SpikingConvNet.classes.SpikingModel method), 8

R

rebuild_fixed_connections() (in module SpikingConvNet.algorithms), 6
rebuild_inhibitory_connections() (in module SpikingConvNet.algorithms), 7
retrieve_data() (SpikingConvNet.classes.Spinnaker_Network method), 9
RunControl (class in SpikingConvNet.utils), 10

S

setup_logger() (SpikingConvNet.utils.RunControl method), 10
spikes_for_classifier() (in module SpikingConvNet.algorithms), 7
SpikingConvNet (module), 5
SpikingConvNet.algorithms (module), 6
SpikingConvNet.classes (module), 8
SpikingConvNet.parameters (module), 10
SpikingConvNet.utils (module), 10
SpikingModel (class in SpikingConvNet.classes), 8
Spinnaker_Network (class in SpikingConvNet.classes), 8

T

train() (SpikingConvNet.classes.Classifier method), 8

U

update_kernel_weights() (SpikingConvNet.classes.Spinnaker_Network method), 9

W

windowed_spikes() (in module SpikingConvNet.algorithms), 7