
MemTorch

Release 1.1.6

Feb 25, 2022

Contents

1	Documentation	3
1.1	Python API	3
1.1.1	memtorch.bh	3
1.1.2	memtorch.map	11
1.1.3	memtorch.mn	13
1.2	Tutorials	21
	Python Module Index	23
	Index	25

MemTorch is a simulation framework for memristive deep learning systems that integrates directly with the well-known PyTorch Machine Learning (ML) library. MemTorch is formally described in *MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems*, which is openly accessible [here](#).

The best place to get started is [here](#).

We provide documentation in the form of a complete Python API, and numerous interactive tutorials. In addition, a Gitter chatroom is available for discussions:

1.1 Python API

MemTorch consists of various submodules, as defined below:

1.1.1 memtorch.bh

Submodule containing various memristive device behavioral models and methods to simulate non-ideal device and circuit behavior.

memtorch.bh.memristor

All memristor models and window functions are encapsulated and documented in `memtorch.bh.memristor`.

memtorch.bh.nonideality

All non-idealities modelled by MemTorch are encapsulated and documented in `memtorch.bh.nonideality`.

memtorch.bh.crossbar.Crossbar

Class used to model memristor crossbars and to manage modular crossbar tiles.

```
import torch
import memtorch

crossbar = memtorch.bh.crossbar.Crossbar(memtorch.bh.memristor.VTEAM,
                                         {"r_on": 1e2, "r_off": 1e4},
                                         shape=(100, 100),
                                         tile_shape=(64, 64))
crossbar.write_conductance_matrix(torch.zeros(100, 100).uniform_(1e-2, 1e-4),
                                  ↪transistor=True)
crossbar.devices[0][0][0].set_conductance(1e-4)
crossbar.update(from_devices=True, parallelize=True)
```

Note: `use_bindings` is enabled by default, to accelerate operation using C++/CUDA (if supported) bindings.

Warning: As of version 1.1.6, the `write_conductance_matrix` method exhibits different behavior when `self.use_bindings` is True, **CUDA** operation is enabled, and the **Data_Driven2021** memristor model is used.

When `self.use_bindings` is True, **CUDA** operation is enabled, and the **Data_Driven2021** memristor model is used, the programming voltage is force adjusted by `force_adjustment_voltage` when a device becomes stuck. For all others models, or when **CUDA** operation is not enabled or `self.use_bindings` is false, the conductance state of the device being modelled is adjusted using `force_adjustment` when it becomes stuck.

This behavior will made consistent across Python, C++, and CUDA bindings, in a future release.

```
class memtorch.bh.crossbar.Crossbar.Crossbar(memristor_model,          memris-
                                             tor_model_params,          shape,
                                             tile_shape=None, use_bindings=True,
                                             cuda_malloc_heap_size=50,    ran-
                                             dom_crossbar_init=False)
```

Bases: `object`

Class used to model memristor crossbars.

Parameters

- **memristor_model** (*memtorch.bh.memristor.Memristor.Memristor*) – Memristor model.
- **memristor_model_params** (***kwargs*) – ***kwargs* to instantiate the memristor model with.
- **shape** (*int, int*) – Shape of the crossbar.
- **tile_shape** (*int, int*) – Tile shape to use to store weights. If None, modular tiles are not used.
- **use_bindings** (*bool*) – Used to determine if C++/CUDA bindings are used (True) or not (False).
- **random_crossbar_init** (*bool*) – Determines if the crossbar is to be initialized at random values in between Ron and Roff

update (*from_devices=True, parallelize=False*)

Method to update either the layers `conductance_matrix` or each devices conductance state.

Parameters

- **from_devices** (*bool*) – The conductance matrix can either be updated from all devices (True), or each device can be updated from the conductance matrix (False).
- **parallelize** (*bool*) – The operation is parallelized (True).

write_conductance_matrix (*conductance_matrix*, *transistor=True*, *programming_routine=None*, *programming_routine_params={}*)

Method to directly program (alter) the conductance of all devices within the crossbar.

Parameters

- **conductance_matrix** (*torch.FloatTensor*) – Conductance matrix to write.
- **transistor** (*bool*) – Used to determine if a 1T1R (True) or 0T1R arrangement (False) is simulated.
- **programming_routine** – Programming routine (method) to use.
- **programming_routine_params** (***kwargs*) – Programming routine keyword arguments.

class memtorch.bh.crossbar.Crossbar.Scheme

Bases: *enum.Enum*

Scheme enumeration.

DoubleColumn = 2

SingleColumn = 1

memtorch.bh.crossbar.Crossbar.**init_crossbar** (*weights*, *memristor_model*, *memristor_model_params*, *transistor*, *mapping_routine*, *programming_routine*, *programming_routine_params={}*, *p_l=None*, *scheme=<Scheme.DoubleColumn: 2>*, *tile_shape=(128, 128)*, *use_bindings=True*, *cuda_malloc_heap_size=50*, *random_crossbar_init=False*)

Method to initialise and construct memristive crossbars.

Parameters

- **weights** (*torch.Tensor*) – Weights to map.
- **memristor_model** (*memtorch.bh.memristor.Memristor.Memristor*) – Memristor model.
- **memristor_model_params** (***kwargs*) – ***kwargs* to instantiate the memristor model with.
- **transistor** (*bool*) – Used to determine if a 1T1R (True) or 1R arrangement (False) is simulated.
- **mapping_routine** (*function*) – Mapping routine to use.
- **programming_routine** (*function*) – Programming routine to use.
- **programming_routine_params** (***kwargs*) – Programming routine keyword arguments.
- **p_l** (*float*) – If not None, the proportion of weights to retain.
- **scheme** (*memtorch.bh.Scheme*) – Scheme enum.
- **tile_shape** (*int*, *int*) – Tile shape to use to store weights. If None, modular tiles are not used.

- **use_bindings** (*bool*) – Used to determine if C++/CUDA bindings are used (True) or not (False).
- **random_crossbar_init** (*boolean*) – Determines if the crossbar is to be initialized at random values in between Ron and Roff

Returns The constructed crossbars and forward() function.

Return type `tuple`

```
memtorch.bh.crossbar.Crossbar.simulate_matmul (input, crossbar, nl=True,  
                                              tiles_map=None, crossbar_shape=None,  
                                              max_input_voltage=None,  
                                              ADC_resolution=None,  
                                              ADC_overflow_rate=0.0,  
                                              quant_method=None,  
                                              use_bindings=True)
```

Method to simulate non-linear IV device characterisitcs for a 2-D crossbar architecture given scaled inputs.

Parameters

- **input** (*torch.Tensor*) – Scaled input tensor.
- **crossbar** (*memtorch.bh.Crossbar*) – Crossbar containing devices to simulate.
- **nl** (*bool*) – Use lookup tables rather than simulating each device (True).
- **tiles_map** (*torch.Tensor*) – Tiles map for devices if tile_shape is not None.
- **crossbar_shape** (*int*, *int*) – Crossbar shape if tile_shape is not None.
- **max_input_voltage** (*float*) – Maximum input voltage used to encode inputs. If None, inputs are unbounded.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quantization (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in memtorch.bh.Quantize.quant_methods.
- **use_bindings** (*bool*) – Used to determine if C++/CUDA bindings are used (True) or not (False).

Returns Output tensor.

Return type `torch.Tensor`

memtorch.bh.crossbar.Program

Methods to program (alter) the conductance devices within a crossbar or modular crossbar tiles.

```
memtorch.bh.crossbar.Program.gen_programming_signal (number_of_pulses,  
                                                    pulse_duration, refac-  
                                                    tory_period, voltage_level,  
                                                    time_series_resolution)
```

Method to generate a programming signal using a sequence of pulses.

Parameters

- **number_of_pulses** (*int*) – Number of pulses.

- **pulse_duration** (*float*) – Duration of the programming pulse (s).
- **refractory_period** (*float*) – Duration of the refractory period (s).
- **voltage_level** (*float*) – Voltage level (V).
- **time_series_resolution** (*float*) – Time series resolution (s).

Returns Tuple containing the generated time and voltage signals.

Return type *tuple*

```
memtorch.bh.crossbar.Program.naive_program(crossbar, point, conductance, rel_tol=0.01,
                                           pulse_duration=0.001, refractory_period=0,
                                           pos_voltage_level=1.0, neg_voltage_level=-
                                           1.0, timeout=5, force_adjustment=0.001,
                                           force_adjustment_rel_tol=0.1,
                                           force_adjustment_pos_voltage_threshold=0,
                                           force_adjustment_neg_voltage_threshold=0,
                                           failure_iteration_threshold=1000, simulate_neighbours=True)
```

Method to program (alter) the conductance of a given device within a crossbar.

Parameters

- **crossbar** (*memtorch.bh.crossbar.Crossbar*) – Crossbar containing the device to program.
- **point** (*tuple*) – Point to program (row, column).
- **conductance** (*float*) – Conductance to program.
- **rel_tol** (*float*) – Relative tolerance between the desired conductance and the device's conductance.
- **pulse_duration** (*float*) – Duration of the programming pulse (s).
- **refractory_period** (*float*) – Duration of the refractory period (s).
- **pos_voltage_level** (*float*) – Positive voltage level (V).
- **neg_voltage_level** (*float*) – Negative voltage level (V).
- **timeout** (*int*) – Timeout (seconds) until stuck devices are unstuck.
- **force_adjustment** (*float*) – Adjustment (resistance) to unstick stuck devices.
- **force_adjustment_rel_tol** (*float*) – Relative tolerance threshold between a stuck device's conductance and high and low conductance states to force adjust.
- **force_adjustment_pos_voltage_threshold** (*float*) – Positive voltage level threshold (V) to enable force adjustment.
- **force_adjustment_neg_voltage_threshold** (*float*) – Negative voltage level threshold (V) to enable force adjustment.
- **failure_iteration_threshold** (*int*) – Failure iteration threshold.
- **simulate_neighbours** (*bool*) – Simulate neighbours (True).

Returns Programmed device.

Return type *memtorch.bh.memristor.Memristor.Memristor*

memtorch.bh.crossbar.Tile**class** memtorch.bh.crossbar.Tile.**Tile** (*tile_shape*, *patch_num=None*)Bases: `object`

Class used to create modular crossbar tiles to represent 2D matrices.

Parameters

- **tile_shape** (*int*, *int*) – Tile shape to use to store weights.
- **patch_num** (*int*) – Patch number.

update_array (*new_array*)

Method to update the tile's weights.

Parameters **new_array** (*torch.Tensor*) – New array to construct the tile with.memtorch.bh.crossbar.Tile.**gen_tiles** (*tensor*, *tile_shape*, *input=False*, *use_bindings=True*)

Method to generate a set of modular tiles representative of a tensor.

Parameters

- **tensor** (*torch.Tensor*) – Tensor to represent using modular crossbar tiles.
- **tile_shape** (*int*, *int*) – Tile shape to use to store weights.
- **input** (*bool*) – Used to determine if a tensor is an input (True).

Returns Tiles and `tile_map`.**Return type** `torch.Tensor`, `torch.Tensor`

```
memtorch.bh.crossbar.Tile.tile_matmul (mat_a_tiles, mat_a_tiles_map, mat_a_shape,  
                                         mat_b_tiles, mat_b_tiles_map, mat_b_shape,  
                                         source_resistance=None, line_resistance=None,  
                                         ADC_resolution=None, ADC_overflow_rate=0.0,  
                                         quant_method=None, transistor=True,  
                                         use_bindings=True, cuda_malloc_heap_size=50)
```

Method to perform 2D matrix multiplication, given two sets of tiles.

Parameters

- **mat_a_tiles** (*torch.Tensor*) – Tiles representing matrix A.
- **mat_a_tiles_map** (*torch.Tensor*) – Tiles map for matrix A.
- **mat_a_shape** (*int*, *int*) – Shape of matrix A.
- **mat_b_tiles** (*torch.Tensor*) – Tiles representing matrix B.
- **mat_b_tiles_map** (*torch.Tensor*) – Tiles map for matrix B.
- **mat_b_shape** (*int*, *int*) – Shape of matrix B.
- **source_resistance** (*float*) – The resistance between word/bit line voltage sources and crossbar(s).
- **line_resistance** (*float*) – The interconnect line resistance between adjacent cells.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quantization (if ADC_resolution is not None).

- **quant_method** (*str*) – Quantization method. Must be in memtorch.bh.Quantize.quant_methods.
- **transistor** (*bool*) – TBD.
- **use_bindings** (*bool*) – Use C++/CUDA bindings to parallelize tile_matmul operations (True).
- **cuda_malloc_heap_size** (*int*) – cudaLimitMallocHeapSize (in MB) to determine allocatable kernel heap memory if CUDA is used.

Returns Output tensor.

Return type torch.Tensor

```
memtorch.bh.crossbar.Tile.tile_matmul_row(mat_a_row_tiles,          mat_a_tiles_map,
                                          mat_b_tiles,          mat_b_tiles_map,
                                          mat_b_shape,          source_resistance=None,
                                          line_resistance=None, ADC_resolution=None,
                                          ADC_overflow_rate=0.0, quant_method=None,
                                          transistor=True)
```

Method to perform row-wise tile matrix multiplication, given two sets of tiles, using a pythonic approach.

Parameters

- **mat_a_row_tiles** (*torch.Tensor*) – Tiles representing a row of matrix A.
- **mat_a_tiles_map** (*torch.Tensor*) – Tiles map for matrix A.
- **mat_b_tiles** (*torch.Tensor*) – Tiles representing matrix B.
- **mat_b_tiles_map** (*torch.Tensor*) – Tiles map for matrix B.
- **mat_b_shape** (*int, int*) – Shape of matrix B.
- **source_resistance** (*float*) – The resistance between word/bit line voltage sources and crossbar(s).
- **line_resistance** (*float*) – The interconnect line resistance between adjacent cells.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quantization (if ADC_resolution is not None).
- **quant_method** (*str*) – Quantization method. Must be in memtorch.bh.Quantize.quant_methods.
- **transistor** (*bool*) – TBD.

Returns Output tensor.

Return type torch.Tensor

```
memtorch.bh.crossbar.Tile.tiled_inference(input, m, transistor)
```

Method to perform tiled inference.

Parameters

- **input** (*torch.Tensor*) – Input tensor (2-D).
- **m** (*memtorch.mn*) – Memristive MemTorch layer.

Returns Output tensor.

Return type torch.Tensor

memtorch.bh.Quantize

Wrapper for C++ quantization bindings.

`memtorch.bh.Quantize.quantize` (*tensor*, *quant*, *overflow_rate=0.0*, *quant_method=None*, *min=nan*, *max=nan*, *override_original=False*)

Method to quantize a tensor.

Parameters

- **tensor** (*torch.Tensor*) – Input tensor.
- **quant** (*int*) – Bit width (if *quant_method* is not *None*) or the number of discrete quantization levels (if *quant_method* is *None*).
- **overflow_rate** (*float*, *optional*) – Overflow rate threshold for linear quantization.
- **quant_method** (*str*, *optional*) – Quantization method. Must be in *quant_methods*.
- **min** (*float* or *tensor*, *optional*) – Minimum value(s) to clip numbers to.
- **max** (*float* or *tensor*, *optional*) – Maximum value(s) to clip numbers to.
- **override_original** (*bool*, *optional*) – Whether to override the original tensor (True) or not (False).

Returns Quantized tensor.

Return type *torch.Tensor*

memtorch.bh.StochasticParameter

Methods to model stochastic parameters.

memtorch.bh.StochasticParameter is most commonly used to define stochastic parameters when defining behavioural memristor models, as follows:

```
import torch
import memtorch

crossbar = memtorch.bh.crossbar.Crossbar(memtorch.bh.memristor.VTEAM,
                                          {"r_on": memtorch.bh.
↪StochasticParameter(min=1e3, max=1e2), "r_off": 1e4},
                                          shape=(100, 100),
                                          tile_shape=(64, 64))
```

class `memtorch.bh.StochasticParameter.Dict2Obj` (*dictionary*)

Bases: `object`

Class used to instantiate a object given a dictionary.

`memtorch.bh.StochasticParameter.StochasticParameter` (*distribution=<class 'torch.distributions.normal.Normal'>*, *min=0*, *max=inf*, *function=True*, ***kwargs*)

Method to model a stochastic parameter.

Parameters

- **distribution** (*torch.distributions*) – torch distribution.
- **min** (*float*) – Minimum value to sample.

- **max** (*float*) – Maximum value to sample.
- **function** (*bool*) – A sampled value is returned (False). A function to return a sampled value or mean is returned (True).

Returns A sampled value of the stochastic parameter, or a sample-value generator.

Return type *float* or function

```
memtorch.bh.StochasticParameter.unpack_parameters(local_args, r_rel_tol=None,
                                                    r_abs_tol=None, resam-
                                                    ple_threshold=5)
```

Method to sample from stochastic sample-value generators

Parameters

- **local_args** (*locals()*) – Local arguments with stochastic sample-value generators from which to sample from.
- **r_rel_tol** (*float*) – Relative threshold tolerance.
- **r_abs_tol** (*float*) – Absolute threshold tolerance.
- **resample_threshold** (*int*) – Number of times to resample *r_off* and *r_on* when their proximity is within the threshold tolerance before raising an exception.

Returns *locals()* with sampled stochastic parameters.

Return type **

1.1.2 memtorch.map

Submodule containing various mapping, scaling, and encoding methods.

memtorch.map.Input

Encapsulates internal methods to encode (scale) input values as bit-line voltages. Methods can either be specified when converting individual layers:

```
from memtorch.map.Input import naive_scale

m = memtorch.mn.Linear(torch.nn.Linear(10, 10),
                        memtorch.bh.memristor.VTEAM,
                        {},
                        tile_shape=(64, 64),
                        scaling_routine=naive_scale)
```

or when converting *torch.nn.Module* instances:

```
import copy
from memtorch.mn.Module import patch_model
from memtorch.map.Input import naive_scale
import Net

model = Net()
patched_model = patch_model(copy.deepcopy(model),
                             memtorch.bh.memristor.VTEAM,
                             {},
                             module_parameters_to_patch=[torch.nn.Linear],
                             scaling_routine=naive_scale)
```

`memtorch.map.Input.naive_scale(module, input, force_scale=False)`

Naive method to encode input values as bit-line voltages.

Parameters

- **module** (`torch.nn.Module`) – Memristive layer to tune.
- **input** (`torch.tensor`) – Input tensor to encode.
- **force_scale** (`bool, optional`) – Used to determine if inputs are scaled (True) or not (False) if they no not exceed `max_input_voltage`.

Returns Encoded voltages.

Return type `torch.Tensor`

Note: `force_scale` is used to specify whether inputs smaller than or equal to `max_input_voltage` are scaled or not.

`memtorch.map.Module`

Encapsulates internal methods to determine relationships between readout currents of memristive crossbars and desired outputs.

Warning: Currently, only `naive_tune` is supported. In a future release, externally-defined methods will be supported.

`memtorch.map.Module.naive_tune(module, input_shape, verbose=True)`

Method to determine a linear relationship between a memristive crossbar and the output for a given memristive module.

Parameters

- **module** (`torch.nn.Module`) – Memristive layer to tune.
- **input_shape** (`int, int`) – Shape of the randomly generated input used to tune a crossbar.
- **verbose** (`bool, optional`) – Used to determine if verbose output is enabled (True) or disabled (False).

Returns Function which transforms the output of the crossbar to the expected output.

Return type function

`memtorch.map.Parameter`

Encapsulates internal methods to naively map network parameters to memristive device conductance values. Methods can either be specified when converting individual layers:

```
from memtorch.map.Parameter import naive_map

m = memtorch.mn.Linear(torch.nn.Linear(10, 10),
                        memtorch.bh.memristor.VTEAM,
                        {},
                        tile_shape=(64, 64),
                        mapping_routine=naive_map)
```


or when converting `torch.nn.Module` instances:

```
import copy
from memtorch.mn.Module import patch_model
from memtorch.map.Parameter import naive_map
import Net

model = Net()
patched_model = patch_model(copy.deepcopy(model),
                             memtorch.bh.memristor.VTEAM,
                             {},
                             module_parameters_to_patch=[torch.nn.Linear],
                             mapping_routine=naive_map)
```

`memtorch.map.Parameter.naive_map` (*weight*, *r_on*, *r_off*, *scheme*, *p_l*=None)

Method to naively map network parameters to memristive device conductances, using two crossbars to represent both positive and negative weights.

Parameters

- **weight** (*torch.Tensor*) – Weight tensor to map.
- **r_on** (*float*) – Low resistance state.
- **r_off** (*float*) – High resistance state.
- **scheme** (*memtorch.bh.crossbar.Scheme*) – Weight representation scheme.
- **p_l** (*float*, *optional*) – If not None, the proportion of weights to retain.

Returns Positive and negative crossbar weights.

Return type `torch.Tensor`, `torch.Tensor`

1.1.3 memtorch.mn

Memristive `torch.nn` equivalent submodule.

memtorch.mn.Module

Encapsulates `memtorch.bmn.Module.patch_model`, which can be used to convert `torch.nn` models.

```
import copy
import Net
from memtorch.mn.Module import patch_model
from memtorch.map.Parameter import naive_map
from memtorch.map.Input import naive_scale

model = Net()
reference_memristor = memtorch.bh.memristor.VTEAM
patched_model = patch_model(copy.deepcopy(model),
                             memristor_model=reference_memristor,
                             memristor_model_params={},
                             module_parameters_to_patch=[torch.nn.Linear, torch.nn.Conv2d],
                             mapping_routine=naive_map,
                             transistor=True,
                             programming_routine=None,
                             tile_shape=(128, 128),
```

(continues on next page)

(continued from previous page)

```

max_input_voltage=0.3,
scaling_routine=naive_scale,
ADC_resolution=8,
ADC_overflow_rate=0.,
quant_method='linear')

```

Warning: It is strongly suggested to copy the original model using `copy.deepcopy` prior to conversion, as some values are overridden by-reference.

```

memtorch.nn.Module.patch_model(model, memristor_model, memristor_model_params, module_parameters_to_patch={}, mapping_routine=<function naive_map>, transistor=True, programming_routine=None, programming_routine_params={'rel_tol': 0.1}, p_l=None, scheme=<Scheme.DoubleColumn: 2>, tile_shape=None, max_input_voltage=None, scaling_routine=<function naive_scale>, scaling_routine_params={}, source_resistance=None, line_resistance=None, ADC_resolution=None, ADC_overflow_rate=0.0, quant_method=None, use_bindings=True, random_crossbar_init=False, verbose=True, **kwargs)

```

Method to convert a torch.nn model to a memristive model.

Parameters

- **model** (*torch.nn.Module*) – torch.nn.Module to patch.
- **memristor_model** (*memtorch.bh.memristor.Memristor.Memristor*) – Memristor model.
- **memristor_model_params** (****kwargs**) – Memristor model keyword arguments.
- **module_parameters_to_patch** (*module_paramter_patches*) – Model parameters to patch.
- **mapping_routine** (*function*) – Mapping routine to use.
- **transistor** (*bool*) – Used to determine if a 1T1R (True) or 1R arrangement (False) is simulated.
- **programming_routine** (*function*) – Programming routine to use.
- **programming_routine_params** (****kwargs**) – Programming routine keyword arguments.
- **p_l** (*float*) – If not None, the proportion of weights to retain.
- **scheme** (*memtorch.bh.Scheme*) – Weight representation scheme.
- **tile_shape** (*(int, int)*) – Tile shape to use to store weights. If None, modular tiles are not used.
- **max_input_voltage** (*float*) – Maximum input voltage used to encode inputs. If None, inputs are unbounded.
- **scaling_routine** (*function*) – Scaling routine to use in order to scale batch inputs.
- **scaling_routine_params** (****kwargs**) – Scaling routine keyword arguments.

- **source_resistance** (*float*) – The resistance between word/bit line voltage sources and crossbar(s).
- **line_resistance** (*float*) – The interconnect line resistance between adjacent cells.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quantization (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'min-max', 'tanh'], or None.
- **use_bindings** (*bool*) – Used to determine if C++/CUDA bindings are used (True) or not (False).
- **random_crossbar_init** (*bool*) – Determines if the crossbar is to be initialized at random values in between Ron and Roff
- **verbose** (*bool*) – Used to determine if verbose output is enabled (True) or disabled (False).

Returns Patched torch.nn.Module.

Return type torch.nn.Module

The following layer/module types are currently supported:

memtorch.nn.Linear

torch.nn.Linear equivalent.

```
class memtorch.nn.Linear.Linear(linear_layer, memristor_model, memristor_model_params,
                                mapping_routine=<function naive_map>, transistor=True,
                                programming_routine=None, programming_routine_params={},
                                p_l=None, scheme=<Scheme.DoubleColumn: 2>, tile_shape=None,
                                max_input_voltage=None, scaling_routine=<function naive_scale>,
                                scaling_routine_params={}, source_resistance=None,
                                line_resistance=None, ADC_resolution=None,
                                ADC_overflow_rate=0.0, quant_method=None,
                                use_bindings=True, random_crossbar_init=False,
                                verbose=True, *args, **kwargs)
```

Bases: torch.nn.modules.linear.Linear

nn.Linear equivalent.

Parameters

- **linear_layer** (*torch.nn.Linear*) – Linear layer to patch.
- **memristor_model** (*memtorch.bh.memristor.Memristor.Memristor*) – Memristor model.
- **memristor_model_params** (****kwargs**) – Memristor model keyword arguments.
- **mapping_routine** (*function*) – Mapping routine to use.
- **transistor** (*bool*) – Used to determine if a 1T1R (True) or 1R arrangement (False) is simulated.
- **programming_routine** (*function*) – Programming routine to use.

- **programming_routine_params** (****kwargs**) – Programming routine keyword arguments.
- **p_l** (*float*) – If not None, the proportion of weights to retain.
- **scheme** (*memtorch.bh.Scheme*) – Weight representation scheme.
- **tile_shape** (*(int, int)*) – Tile shape to use to store weights. If None, modular tiles are not used.
- **max_input_voltage** (*float*) – Maximum input voltage used to encode inputs. If None, inputs are unbounded.
- **scaling_routine** (*function*) – Scaling routine to use in order to scale batch inputs.
- **scaling_routine_params** (****kwargs**) – Scaling routine keyword arguments.
- **source_resistance** (*float*) – The resistance between word/bit line voltage sources and crossbar(s).
- **line_resistance** (*float*) – The interconnect line resistance between adjacent cells.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quantization (if ADC_resolution is not None).
- **quant_method** (*string*) – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'minmax', 'tanh'], or None.
- **use_bindings** (*bool*) – Used to determine if C++/CUDA bindings are used (True) or not (False).
- **random_crossbar_init** (*bool*) – Determines if the crossbar is to be initialized at random values in between Ron and Roff
- **verbose** (*bool*) – Used to determine if verbose output is enabled (True) or disabled (False).

forward (*input*)

Method to perform forward propagations.

Parameters **input** (*torch.Tensor*) – Input tensor.

Returns Output tensor.

Return type *torch.Tensor*

tune (*input_shape=4098*)

Tuning method.

memtorch.mn.Conv1d

torch.nn.Conv1d equivalent.

```
class memtorch.nn.Conv1d.Conv1d(convolutional_layer, memristor_model, memristor_model_params, mapping_routine=<function naive_map>, transistor=True, programming_routine=None, programming_routine_params={}, p_l=None, scheme=<Scheme.DoubleColumn: 2>, tile_shape=None, max_input_voltage=None, scaling_routine=<function naive_scale>, scaling_routine_params={}, source_resistance=None, line_resistance=None, ADC_resolution=None, ADC_overflow_rate=0.0, quant_method=None, use_bindings=True, random_crossbar_init=False, verbose=True, *args, **kwargs)
```

Bases: torch.nn.modules.conv.Conv1d

nn.Conv1d equivalent.

Parameters

- **convolutional_layer** (*torch.nn.Conv1d*) – Convolutional layer to patch.
- **memristor_model** (*memtorch.bh.memristor.Memristor.Memristor*) – Memristor model.
- **memristor_model_params** (****kwargs**) – Memristor model keyword arguments.
- **mapping_routine** (*function*) – Mapping routine to use.
- **transistor** (*bool*) – Used to determine if a 1T1R (True) or 1R arrangement (False) is simulated.
- **programming_routine** (*function*) – Programming routine to use.
- **programming_routine_params** (****kwargs**) – Programming routine keyword arguments.
- **p_l** (*float*) – If not None, the proportion of weights to retain.
- **scheme** (*memtorch.bh.Scheme*) – Weight representation scheme.
- **tile_shape** (*(int, int)*) – Tile shape to use to store weights. If None, modular tiles are not used.
- **max_input_voltage** (*float*) – Maximum input voltage used to encode inputs. If None, inputs are unbounded.
- **scaling_routine** (*function*) – Scaling routine to use in order to scale batch inputs.
- **scaling_routine_params** (****kwargs**) – Scaling routine keyword arguments.
- **source_resistance** (*float*) – The resistance between word/bit line voltage sources and crossbar(s).
- **line_resistance** (*float*) – The interconnect line resistance between adjacent cells.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quantization (if ADC_resolution is not None).
- **quant_method** (*string*) – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'minmax', 'tanh'], or None.
- **use_bindings** (*bool*) – Used to determine if C++/CUDA bindings are used (True) or not (False).

- **random_crossbar_init** (*bool*) – Determines if the crossbar is to be initialized at random values in between Ron and Roff
- **verbose** (*bool*) – Used to determine if verbose output is enabled (True) or disabled (False).

forward (*input*)

Method to perform forward propagations.

Parameters **input** (*torch.Tensor*) – Input tensor.

Returns Output tensor.

Return type *torch.Tensor*

tune (*input_batch_size=8, input_shape=32*)

Tuning method.

memtorch.nn.Conv2d

torch.nn.Conv2d equivalent.

```
class memtorch.nn.Conv2d.Conv2d(convolutional_layer, memristor_model, memristor_model_params, mapping_routine=<function naive_map>, transistor=True, programming_routine=None, programming_routine_params={}, p_1=None, scheme=<Scheme.DoubleColumn: 2>, tile_shape=None, max_input_voltage=None, scaling_routine=<function naive_scale>, scaling_routine_params={}, source_resistance=None, line_resistance=None, ADC_resolution=None, ADC_overflow_rate=0.0, quant_method=None, use_bindings=True, random_crossbar_init=False, verbose=True, *args, **kwargs)
```

Bases: *torch.nn.modules.conv.Conv2d*

nn.Conv2d equivalent.

Parameters

- **convolutional_layer** (*torch.nn.Conv2d*) – Convolutional layer to patch.
- **memristor_model** (*memtorch.bh.memristor.Memristor.Memristor*) – Memristor model.
- **memristor_model_params** (**kwargs) – Memristor model keyword arguments.
- **mapping_routine** (*function*) – Mapping routine to use.
- **transistor** (*bool*) – Used to determine if a 1T1R (True) or 1R arrangement (False) is simulated.
- **programming_routine** (*function*) – Programming routine to use.
- **programming_routine_params** (**kwargs) – Programming routine keyword arguments.
- **p_1** (*float*) – If not None, the proportion of weights to retain.
- **scheme** (*memtorch.bh.Scheme*) – Weight representation scheme.
- **tile_shape** ((*int*, *int*)) – Tile shape to use to store weights. If None, modular tiles are not used.

- **max_input_voltage** (*float*) – Maximum input voltage used to encode inputs. If None, inputs are unbounded.
- **scaling_routine** (*function*) – Scaling routine to use in order to scale batch inputs.
- **scaling_routine_params** (****kwargs**) – Scaling routine keyword arguments.
- **source_resistance** (*float*) – The resistance between word/bit line voltage sources and crossbar(s).
- **line_resistance** (*float*) – The interconnect line resistance between adjacent cells.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quantization (if ADC_resolution is not None).
- **quant_method** (*string*) – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'minmax', 'tanh'], or None.
- **use_bindings** (*bool*) – Used to determine if C++/CUDA bindings are used (True) or not (False).
- **random_crossbar_init** (*bool*) – Determines if the crossbar is to be initialized at random values in between Ron and Roff
- **verbose** (*bool*) – Used to determine if verbose output is enabled (True) or disabled (False).

forward (*input*)

Method to perform forward propagations.

Parameters **input** (*torch.Tensor*) – Input tensor.

Returns Output tensor.

Return type *torch.Tensor*

tune (*input_batch_size=8, input_shape=32*)

Tuning method.

memtorch.nn.Conv3d

torch.nn.Conv3d equivalent.

```
class memtorch.nn.Conv3d.Conv3d(convolutional_layer, memristor_model, memristor_model_params, mapping_routine=<function naive_map>, transistor=True, programming_routine=None, programming_routine_params={}, p_l=None, scheme=<Scheme.DoubleColumn: 2>, tile_shape=None, max_input_voltage=None, scaling_routine=<function naive_scale>, scaling_routine_params={}, source_resistance=None, line_resistance=None, ADC_resolution=None, ADC_overflow_rate=0.0, quant_method=None, use_bindings=True, random_crossbar_init=False, verbose=True, *args, **kwargs)
```

Bases: *torch.nn.modules.conv.Conv3d*

nn.Conv3d equivalent.

Parameters

- **convolutional_layer** (*torch.nn.Conv3d*) – Convolutional layer to patch.
- **memristor_model** (*memtorch.bh.memristor.Memristor.Memristor*) – Memristor model.
- **memristor_model_params** (**kwargs) – Memristor model keyword arguments.
- **mapping_routine** (*function*) – Mapping routine to use.
- **transistor** (*bool*) – Used to determine if a 1T1R (True) or 1R arrangement (False) is simulated.
- **programming_routine** (*function*) – Programming routine to use.
- **programming_routine_params** (**kwargs) – Programming routine keyword arguments.
- **p_l** (*float*) – If not None, the proportion of weights to retain.
- **scheme** (*memtorch.bh.Scheme*) – Weight representation scheme.
- **tile_shape** (*(int, int)*) – Tile shape to use to store weights. If None, modular tiles are not used.
- **max_input_voltage** (*float*) – Maximum input voltage used to encode inputs. If None, inputs are unbounded.
- **scaling_routine** (*function*) – Scaling routine to use in order to scale batch inputs.
- **scaling_routine_params** (**kwargs) – Scaling routine keyword arguments.
- **source_resistance** (*float*) – The resistance between word/bit line voltage sources and crossbar(s).
- **line_resistance** (*float*) – The interconnect line resistance between adjacent cells.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quantization (if ADC_resolution is not None).
- **quant_method** (*string*) – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'minmax', 'tanh'], or None.
- **use_bindings** (*bool*) – Used to determine if C++/CUDA bindings are used (True) or not (False).
- **random_crossbar_init** (*bool*) – Determines if the crossbar is to be initialized at random values in between Ron and Roff
- **verbose** (*bool*) – Used to determine if verbose output is enabled (True) or disabled (False).

forward (*input*)

Method to perform forward propagations.

Parameters *input* (*torch.Tensor*) – Input tensor.

Returns Output tensor.

Return type *torch.Tensor*

tune (*input_batch_size=4, input_shape=32*)

Tuning method.

1.2 Tutorials

To learn how to use MemTorch using interactive tutorials, and to reproduce simulations presented in ‘MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems’ [1], we provide numerous Jupyter notebooks.

Jupyter Notebook	Note-Description	Google Colab Link
Tutorial	Introductory Tutorial- Start Here	
Exemplar Simulations	Various Exemplar Simulations, As Presented In [1]	
Case Study	(<i>Legacy</i>) An epileptic Seizure Detection Case Study	
Novel Simulations	(<i>Legacy</i>) Novel Simulations Using CIFAR-10	

The development of more Jupyter notebooks and tutorials is currently ongoing.

[1] C. Lammie, W. Xiang, B. Linares-Barranco, and Azghadi, Mostafa Rahimi, “MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems,” arXiv.org, 2020. <https://arxiv.org/abs/2004.10971>.

m

- `memtorch.bh.crossbar.Crossbar`, [4](#)
- `memtorch.bh.crossbar.Program`, [6](#)
- `memtorch.bh.crossbar.Tile`, [8](#)
- `memtorch.bh.Quantize`, [10](#)
- `memtorch.bh.StochasticParameter`, [10](#)
- `memtorch.map.Input`, [11](#)
- `memtorch.map.Module`, [12](#)
- `memtorch.map.Parameter`, [13](#)
- `memtorch.mn.Conv1d`, [16](#)
- `memtorch.mn.Conv2d`, [18](#)
- `memtorch.mn.Conv3d`, [19](#)
- `memtorch.mn.Linear`, [15](#)
- `memtorch.mn.Module`, [14](#)

C

`Conv1d` (class in `memtorch.mn.Conv1d`), 16

`Conv2d` (class in `memtorch.mn.Conv2d`), 18

`Conv3d` (class in `memtorch.mn.Conv3d`), 19

`Crossbar` (class in `memtorch.bh.crossbar.Crossbar`), 4

D

`Dict2Obj` (class in `memtorch.bh.StochasticParameter`), 10

`DoubleColumn` (memtorch.bh.crossbar.Crossbar.Scheme attribute), 5

F

`forward()` (`memtorch.mn.Conv1d.Conv1d` method), 18

`forward()` (`memtorch.mn.Conv2d.Conv2d` method), 19

`forward()` (`memtorch.mn.Conv3d.Conv3d` method), 20

`forward()` (`memtorch.mn.Linear.Linear` method), 16

G

`gen_programming_signal()` (in module `memtorch.bh.crossbar.Program`), 6

`gen_tiles()` (in module `memtorch.bh.crossbar.Tile`), 8

I

`init_crossbar()` (in module `memtorch.bh.crossbar.Crossbar`), 5

L

`Linear` (class in `memtorch.mn.Linear`), 15

M

`memtorch.bh.crossbar.Crossbar` (module), 4

`memtorch.bh.crossbar.Program` (module), 6

`memtorch.bh.crossbar.Tile` (module), 8

`memtorch.bh.Quantize` (module), 10

`memtorch.bh.StochasticParameter` (module), 10

`memtorch.map.Input` (module), 11

`memtorch.map.Module` (module), 12

`memtorch.map.Parameter` (module), 13

`memtorch.mn.Conv1d` (module), 16

`memtorch.mn.Conv2d` (module), 18

`memtorch.mn.Conv3d` (module), 19

`memtorch.mn.Linear` (module), 15

`memtorch.mn.Module` (module), 14

N

`naive_map()` (in module `memtorch.map.Parameter`), 13

`naive_program()` (in module `memtorch.bh.crossbar.Program`), 7

`naive_scale()` (in module `memtorch.map.Input`), 11

`naive_tune()` (in module `memtorch.map.Module`), 12

P

`patch_model()` (in module `memtorch.mn.Module`), 14

Q

`quantize()` (in module `memtorch.bh.Quantize`), 10

S

`Scheme` (class in `memtorch.bh.crossbar.Crossbar`), 5

`simulate_matmul()` (in module `memtorch.bh.crossbar.Crossbar`), 6

`SingleColumn` (memtorch.bh.crossbar.Crossbar.Scheme attribute), 5

`StochasticParameter()` (in module `memtorch.bh.StochasticParameter`), 10

T

`Tile` (class in `memtorch.bh.crossbar.Tile`), 8

`tile_matmul()` (in module `memtorch.bh.crossbar.Tile`), 8

`tile_matmul_row()` (in module `memtorch.bh.crossbar.Tile`), 9

`tilled_inference()` (in module `memtorch.bh.crossbar.Tile`), [9](#)
`tune()` (`memtorch.nn.Conv1d.Conv1d` method), [18](#)
`tune()` (`memtorch.nn.Conv2d.Conv2d` method), [19](#)
`tune()` (`memtorch.nn.Conv3d.Conv3d` method), [20](#)
`tune()` (`memtorch.nn.Linear.Linear` method), [16](#)

U

`unpack_parameters()` (in module `memtorch.bh.StochasticParameter`), [11](#)
`update()` (`memtorch.bh.crossbar.Crossbar.Crossbar` method), [4](#)
`update_array()` (`memtorch.bh.crossbar.Tile.Tile` method), [8](#)

W

`write_conductance_matrix()` (`memtorch.bh.crossbar.Crossbar.Crossbar` method), [5](#)