MemTorch

Release 1.1.0

Feb 14, 2021

Contents

MemTorch is a simulation framework for memristive deep learning systems that integrates directly with the well-known PyTorch Machine Learning (ML) library.

CHAPTER 1

Python API

1.1 memtorch.bh

Submodule containing various behavioral models.

1.1.1 memtorch.bh.memristor

Submodule containing various behavioral memristor models, that extend base-class-label.

1.1.1.1 memtorch.bh.memristor.Memristor

class memtorch.bh.memristor.Memristor.**Memristor**(*r_off*, *r_on*, *time_series_resolution*, *pos_write_threshold=0*, *neg_write_threshold=0*)

Bases: abc.ABC

Parameters

- **r_off** (*float*) – Off (maximum) resistance of the device (ohms).
- **r_on** (*float*) – On (minimum) resistance of the device (ohms).
- **time_series_resolution** (*float*) – Time series resolution (s).
- **pos_write_threshold** (*float*) – Positive write threshold voltage (V).
- **neg_write_threshold** (*float*) – Negative write threshold voltage (V).

get_resistance()

Method to determine the resistance of a memristive device.

Returns The devices resistance (ohms).

Return type float

plot_bipolar_switching_behaviour(*memristor*, *voltage_signal_amplitude*, *voltage_signal_frequency*, *log_scale=True*, *return_result=False*)

Method to plot the DC bipolar switching behaviour of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

plot_hysteresis_loop(*memristor*, *duration*, *voltage_signal_amplitude*, *voltage_signal_frequency*, *log_scale=False*, *return_result=False*)

Method to plot the hysteresis loop of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **duration** (*float*) – Duration (s).
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

set_conductance(*conductance*)

Method to manually set the conductance of a memristive device.

Parameters **conductance** (*float*) – Conductance to set.

simulate(*voltage_signal*)

Method to determine the equivalent conductance of a memristive device when a given voltage signal is applied.

Parameters **voltage_signal** (*torch.Tensor*) – A discrete voltage signal with resolution time_series_resolution.

Returns A tensor containing the equivalent device conductance for each simulated timestep.

Return type torch.Tensor

memtorch.bh.memristor.Memristor.**plot_bipolar_switching_behaviour**(*memristor_model*, *voltage_signal_amplitude*, *voltage_signal_frequency*, *log_scale=True*, *return_result=False*)

Method to plot the DC bipolar switching behaviour of a given device.

Parameters

- **memristor_model** (memtorch.bh.memristor.Memristor.Memristor) – Memristor model.
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

memtorch.bh.memristor.Memristor.**plot_hysteresis_loop**(*memristor_model*, *duration*, *voltage_signal_amplitude*, *voltage_signal_frequency*, *log_scale=False*, *return_result=False*)

Method to plot the hysteresis loop of a given device.

Parameters

- **memristor_model** (memtorch.bh.memristor.Memristor.Memristor) – Memristor model.
- **duration** (*float*) – Duration (s).
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

Returns Voltage and current signals.

Return type tuple

1.1.1.2 memtorch.bh.memristor.LinearIonDrift

class memtorch.bh.memristor.LinearIonDrift.**LinearIonDrift**(*time_series_resolution=0.0001*, *u_v=1e-14*, *d=1e-08*, *r_on=100*, *r_off=16000.0*, *pos_write_threshold=0.55*, *neg_write_threshold=-0.55*, *p=1*, ***kwargs*)

Bases: *memtorch.bh.memristor.Memristor.Memristor*

Linear Ion behvaioural drift model.

Parameters

- **time_series_resolution** (*float*) – Time series resolution (s).
- **u_v** (*float*) – Dopant drift mobility of the device material.
- **d** (*float*) – Device length (m).
- **r_on** (*float*) – On (minimum) resistance of the device (ohms).
- **r_off** (*float*) – Off (maximum) resistance of the device (ohms).
- **pos_write_threshold** (*float*) – Positive write threshold voltage (V).
- **neg_write_threshold** (*float*) – Negative write threshold voltage (V).

- **p** (*int*) – Joglekar window p constant.

current(*voltage*)
Method to determine the current of the model given an applied voltage.

Parameters **voltage** (*float*) – The current applied voltage (V).

Returns The observed current (A).

Return type float

dxdt(*current*)
Method to determine the derivative of the state variable, dx/dt.

Parameters **current** (*float*) – The observed current (A).

Returns The derivative of the state variable, dx/dt.

Return type float

plot_bipolar_switching_behaviour(*voltage_signal_amplitude=5*, *voltage_signal_frequency=2.5*, *log_scale=True*, *return_result=False*)
Method to plot the DC bipolar switching behaviour of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

plot_hysteresis_loop(*duration=4*, *voltage_signal_amplitude=5*, *voltage_signal_frequency=2.5*, *return_result=False*)
Method to plot the hysteresis loop of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **duration** (*float*) – Duration (s).
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

set_conductance(*conductance*)
Method to manually set the conductance of a memristive device.

Parameters **conductance** (*float*) – Conductance to set.

simulate(*voltage_signal*, *return_current=False*)
Method to determine the equivalent conductance of a memristive device when a given voltage signal is applied.

Parameters **voltage_signal** (*torch.Tensor*) – A discrete voltage signal with resolution time_series_resolution.

Returns A tensor containing the equivalent device conductance for each simulated timestep.

Return type torch.Tensor

1.1.1.3 memtorch.bh.memristor.VTEAM

class memtorch.bh.memristor.VTEAM.**VTEAM**(*time_series_resolution=1e-10*, *r_off=1000*, *r_on=50*, *d=3e-09*, *k_on=-10*, *k_off=0.0005*, *alpha_on=3*, *alpha_off=1*, *v_on=-0.2*, *v_off=0.02*, *x_on=0*, *x_off=3e-09*, ***kwargs*)

Bases: *memtorch.bh.memristor.Memristor.Memristor*

VTEAM memristor model (https://asic2.group/tools/memristor-models/).

Parameters

- **time_series_resolution** (*float*) – Time series resolution (s).
- **r_off** (*float*) – Off (maximum) resistance of the device (ohms).
- **r_on** (*float*) – On (minimum) resistance of the device (ohms).
- **d** (*float*) – Device length (m).
- **k_on** (*float*) – k_on model parameter.
- **k_off** (*float*) – k_off model parameter.
- **alpha_on** (*float*) – alpha_on model parameter.
- **alpha_off** (*float*) – alpha_off model parameter.
- **v_on** (*float*) – Positive write threshold voltage (V).
- **v_off** (*float*) – Negative write threshold voltage (V).
- **x_on** (*float*) – x_on model parameter.
- **x_off** (*float*) – x_off model parameter.

current(*voltage*)

Method to determine the current of the model given an applied voltage.

Parameters **voltage** (*float*) – The current applied voltage (V).

Returns The observed current (A).

Return type float

dxdt(*voltage*)

Method to determine the derivative of the state variable.

Parameters **voltage** (*float*) – The current applied voltage (V).

Returns The derivative of the state variable.

Return type float

plot_bipolar_switching_behaviour(*voltage_signal_amplitude=1.5*, *voltage_signal_frequency=50000000.0*, *log_scale=True*, *return_result=False*)

Method to plot the DC bipolar switching behaviour of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.

- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

plot_hysteresis_loop(*duration=2e-07*, *voltage_signal_amplitude=1*, *voltage_signal_frequency=50000000.0*, *return_result=False*)

Method to plot the hysteresis loop of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **duration** (*float*) – Duration (s).
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

set_conductance(*conductance*)

Method to manually set the conductance of a memristive device.

Parameters **conductance** (*float*) – Conductance to set.

simulate(*voltage_signal*, *return_current=False*)

Method to determine the equivalent conductance of a memristive device when a given voltage signal is applied.

Parameters **voltage_signal** (*torch.Tensor*) – A discrete voltage signal with resolution time_series_resolution.

Returns A tensor containing the equivalent device conductance for each simulated timestep.

Return type torch.Tensor

1.1.1.4 memtorch.bh.memristor.Data_Driven

class memtorch.bh.memristor.Data_Driven.**Data_Driven**(*time_series_resolution=1e-08*, *r_off=17000.0*, *r_on=1280*, *A_p=743.47*, *A_n=-68012.28374*, *t_p=6.51*, *t_n=0.31645*, *k_p=0.000511*, *k_n=0.00117*, *r_p=[16719, 0]*, *r_n=[29304.82557, 23692.77225]*, *eta=1*, *a_p=0.24*, *a_n=0.24*, *b_p=3*, *b_n=3*, ***kwargs*)

Bases: *memtorch.bh.memristor.Memristor.Memristor*

A Data-Driven Verilog-A ReRAM Model (https://eprints.soton.ac.uk/411693/).

Parameters

- **time_series_resolution** (*float*) – Time series resolution (s).
- **r_off** (*float*) – Off (maximum) resistance of the device (ohms).

- **r_on** (*float*) – On (minimum) resistance of the device (ohms).
- **A_p** (*float*) – A_p model parameter.
- **A_n** (*float*) – A_n model parameter.
- **t_p** (*float*) – t_p model parameter.
- **t_n** (*float*) – t_n model parameter.
- **k_p** (*float*) – k_p model parameter.
- **k_n** (*float*) – k_n model parameter.
- **r_p** (*float*) – r_p voltage-dependent resistive boundary function coefficients.
- **r_n** (*float*) – r_n voltage-dependent resistive boundary function coefficients.
- **eta** (*int*) – Switching direction to stimulus polarity.
- **a_p** (*float*) – a_p model parameter.
- **a_n** (*float*) – a_n model parameter.
- **b_p** (*float*) – b_p model parameter.
- **b_n** (*float*) – b_n model parameter.

current(*voltage*)

Method to determine the current of the model given an applied voltage.

Parameters **voltage** (*float*) – The current applied voltage (V).

Returns The observed current (A).

Return type float

plot_bipolar_switching_behaviour(*voltage_signal_amplitude=1.5*, *voltage_signal_frequency=10000.0*, *log_scale=True*, *return_result=False*)

Method to plot the DC bipolar switching behaviour of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

plot_hysteresis_loop(*duration=0.001*, *voltage_signal_amplitude=1.5*, *voltage_signal_frequency=10000.0*, *return_result=False*)

Method to plot the hysteresis loop of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **duration** (*float*) – Duration (s).
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)

- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

resistance(*voltage*)
Method to determine the resistance of the model given an applied voltage.

Parameters **voltage** (*float*) – The current applied voltage (V).

Returns The observed resistance (Ω).

Return type float

set_conductance(*conductance*)
Method to manually set the conductance of a memristive device.

Parameters **conductance** (*float*) – Conductance to set.

simulate(*voltage_signal*, *return_current=False*)
Method to determine the equivalent conductance of a memristive device when a given voltage signal is applied.

Parameters **voltage_signal** (*torch.Tensor*) – A discrete voltage signal with resolution time_series_resolution.

Returns A tensor containing the equivalent device conductance for each simulated timestep.

Return type torch.Tensor

1.1.1.5 memtorch.bh.memristor.Stanford_PKU

class memtorch.bh.memristor.Stanford_PKU.**Stanford_PKU**(*time_series_resolution=0.0001*, *r_off=218586*, *r_on=542*, *gap_init=2e-10*, *g_0=2.5e-10*, *V_0=0.25*, *I_0=0.001*, *read_voltage=0.1*, *T_init=298*, *R_th=2100.0*, *gamma_init=16*, *beta=0.8*, *t_ox=1.2e-08*, *F_min=1400000000.0*, *vel_0=10*, *E_a=0.6*, *a_0=2.5e-10*, *delta_g_init=0.02*, *model_switch=0*, *T_crit=450*, *T_smth=500*, ***kwargs*)

Bases: *memtorch.bh.memristor.Memristor.Memristor*

Stanford PKU memristor model (https://nano.stanford.edu/stanford-rram-model).

Parameters

- **time_series_resolution** (*float*) – Time series resolution (s).
- **r_off** (*float*) – Off (maximum) resistance of the device (ohms).
- **r_on** (*float*) – On (minimum) resistance of the device (ohms).
- **gap_init** (*float*) – Initial gap distance (m).
- **g_0** (*float*) – g_0 model parameter.
- **V_0** (*float*) – V_0 model parameter.

- **I_0** (*float*) – I_0 model parameter.
- **read_voltage** (*float*) – Read voltage (V) to determine the device's conductance.
- **T_init** (*float*) – Initial room tempurature.
- **R_th** (*float*) – Thermal resistance.
- **gamma_init** (*float*) – gamma_init model parameter.
- **beta** (*float*) – beta model parameter.
- **t_ox** (*float*) – Oxide thickness (m).
- **F_min** (*float*) – Minimum field requirement to enhance gap formation.
- **vel_0** (*float*) – vel_0 model parameter.
- **E_a** (*float*) – Activation energy.
- **a_0** (*float*) – Atom spacing.
- **delta_g_init** (*float*) – Initial delta_g value.
- **model_switch** (*int*) – Switch to select standard model (0) or dynamic model (1).
- **T_crit** (*float*) – Threshold temperature (K) for significant random variations.
- **T_smth** (*float*) – Activation energy for vacancy generation.

T_current(*voltage*, *current*)
Method to determine the thermal current of the model given an applied voltage and current.

Parameters

- **voltage** (*float*) – The current applied voltage (V).
- **current** (*float*) – The current applied current (A).

Returns The observed thermal current (A).

Return type float

current(*voltage*)
Method to determine the current of the model given an applied voltage.

Parameters **voltage** (*float*) – The current applied voltage (V).

Returns The observed current (A).

Return type float

dg_dt(*voltage*, *current*)
Method to determine the derivative of the gap length.

Parameters

- **voltage** (*float*) – The current applied voltage (V).
- **current** (*float*) – The current applied current (A).

Returns The derivative of the gap length.

Return type float

plot_bipolar_switching_behaviour(*voltage_signal_amplitude=1.5*, *voltage_signal_frequency=0.05*, *log_scale=True*, *return_result=False*)
Method to plot the DC bipolar switching behaviour of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

plot_hysteresis_loop(*duration=0.5*, *voltage_signal_amplitude=1.5*, *voltage_signal_frequency=10*, *log_scale=False*, *return_result=False*)

Method to plot the hysteresis loop of a given device.

Parameters

- **memristor** (memtorch.bh.memristor.Memristor.Memristor) – Memristor.
- **duration** (*float*) – Duration (s).
- **voltage_signal_amplitude** (*float*) – Voltage signal amplitude (V).
- **voltage_signal_frequency** (*float*) – Voltage signal frequency (Hz)
- **log_scale** (*bool*) – Plot the y-axis (current) using a symmetrical log scale (True).
- **return_result** (*bool*) – Voltage and current signals are returned (True).

set_conductance(*conductance*)

Method to manually set the conductance of a memristive device.

Parameters **conductance** (*float*) – Conductance to set.

simulate(*voltage_signal*, *return_current=False*)

Method to determine the equivalent conductance of a memristive device when a given voltage signal is applied.

Parameters **voltage_signal** (*torch.Tensor*) – A discrete voltage signal with resolution time_series_resolution.

Returns A tensor containing the equivalent device conductance for each simulated timestep.

Return type torch.Tensor

1.1.2 memtorch.bh.nonideality

Submodule containing various models, which can be used to introduce non-idealities.

1.1.2.1 memtorch.bh.nonideality.NonIdeality

class memtorch.bh.nonideality.NonIdeality.**NonIdeality**

Bases: enum.Enum

NonIdeality enumeration.

DeviceFaults = 2

FiniteConductanceStates = 1

NonLinear = 3

memtorch.bh.nonideality.NonIdeality.**apply_nonidealities**(*model*, *non_idealities*, ***kwargs*)

Method to apply non-idealities to a torch.nn.Module instance with memristive layers.

Parameters

- **model** (*torch.nn.Module*) – torch.nn.Module instance.
- **nonidealities** (memtorch.bh.nonideality.NonIdeality.NonIdeality, *tuple*) – Non-linearitites to model.

Returns Patched instance.

Return type torch.nn.Module

memtorch.bh.nonideality.NonIdeality.**required**(*kwargs*, *arguments*, *call*)

Method to check is required arguments in **kwargs are present.

Parameters

- **kwargs** (**kwargs) – Keyword-arguments.
- **arguments** (*list of str*) – Arguments which are required to be present.
- **call** (*str*) – Function to call.

1.1.2.2 memtorch.bh.nonideality.FiniteConductanceStates

Used to model a finite number of conductance states.

memtorch.bh.nonideality.FiniteConductanceStates.**apply_finite_conductance_states**(*layer*, *num_conductar*

Method to model a finite number of conductance states for devices within a memristive layer.

Parameters

- **layer** (*memtorch.mn*) – A memrstive layer.
- **num_conductance_states** (*int*) – Number of finite conductance states to model.

Returns The patched memristive layer.

Return type memtorch.mn

1.1.2.3 memtorch.bh.nonideality.DeviceFaults

memtorch.bh.nonideality.DeviceFaults.**apply_cycle_variability**(*layer*, *distribution=<class 'torch.distributions.normal.Normal'>*, *min=0*, *max=inf*, *parallelize=False*, *r_off_kwargs={}*, *r_on_kwargs={}*)

Method to apply cycle-to-cycle variability to a memristive layer.

Parameters

- **layer** (*memtorch.mn*) – A memrstive layer.
- **distribution** (*torch.distributions*) – torch distribution.
- **min** (*float*) – Minimum value to sample.
- **max** (*float*) – Maximum value to sample.

- **parallelize** (*bool*) – The operation is parallelized (True).
- **r_off_kwargs** (*dict*) – r_off kwargs.
- **r_on_kwargs** (*dict*) – r_on kwargs.

memtorch.bh.nonideality.DeviceFaults.**apply_device_faults**(*layer*, *lrs_proportion*, *hrs_proportion*, *electroform_proportion*)

Method to model device failure within a memristive layer.

Parameters

- **layer** (*memtorch.mn*) – A memrstive layer.
- **lrs_proportion** (*float*) – Proportion of devices which become stuck at a low resistance state.
- **hrs_proportion** (*float*) – Proportion of devices which become stuck at a high resistance state.
- **electroform_proportion** (*float*) – Proportion of devices which fail to electroform.

Returns The patched memristive layer.

Return type memtorch.mn

1.1.2.4 memtorch.bh.nonideality.NonLinear

memtorch.bh.nonideality.NonLinear.**apply_non_linear**(*layer*, *sweep_duration=1*, *sweep_voltage_signal_amplitude=1*, *sweep_voltage_signal_frequency=1*, *num_conductance_states=None*, *simulate=False*)

Method to model non_linear iv characteristics for devices within a memristive layer.

Parameters

- **layer** (*memtorch.mn*) – A memrstive layer.
- **sweep_duration** (*float*) – Voltage sweep duration (s).
- **sweep_voltage_signal_amplitude** (*float*) – Voltage sweep amplitude (v).
- **sweep_voltage_signal_frequency** (*float*) – Voltage sweep frequency (Hz).
- **num_conductance_states** (*int*) – Number of finite conductance states to model. None indicates finite states are not to be modeled.
- **simulate** (*bool*) – Each device is simulated during inference (True).

Returns The patched memristive layer.

Return type memtorch.mn

1.1.3 memtorch.bh.crossbar.Crossbar

Class used to model memristor crossbars.

class memtorch.bh.crossbar.Crossbar.**Crossbar**(*memristor_model*, *memristor_model_params*, *shape*, *tile_shape=None*)

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.

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 1R 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)*)

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.

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*)

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

Parameters

- **input** (*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 quanitzation (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in [‘linear’, ‘log’, ‘log_minmax’, ‘minmax’, ‘tanh’], or None.

Returns Output tensor.

Return type torch.tensor

1.1.4 memtorch.bh.crossbar.Tile

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

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

Parameters

- **tensor** (*torch.tensor*) – TBD.
- **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*, *ADC_resolution=None*, *ADC_overflow_rate=0.0*, *quant_method=None*)
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.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quanitzation (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'minmax', 'tanh'], or None.

Returns Output tensor.

Return type torch.tensor

1.1.5 memtorch.bh.crossbar.Program

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

memtorch.bh.crossbar.Program.**gen_programming_signal**(*number_of_pulses*, *pulse_duration*, *refactory_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).
- **refactory_period** (*float*) – Duration of the refactory 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*, *refactory_period=0*, *pos_voltage_level=1.0*, *neg_voltage_level=-1.0*, *simulate_neighbours=True*, *timeout=10*, *timeout_adjustment=1e-09*)

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).
- **refactory_period** (*float*) – Duration of the refactory 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.
- **timeout_adjustment** (*float*) – Adjustment (resistance) to unstick stuck devices.
- **simulate_neighbours** (*bool*) – Simulate neighbours (True).

Returns Programmed device.

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

1.1.6 memtorch.bh.Quantize

Wrapper for the pytorch-playground quant.py script.

memtorch.bh.Quantize.**quantize**(*input*, *bits*, *overflow_rate*, *quant_method='linear'*, *min=None*, *max=None*)

Method to quantize a tensor.

Parameters

- **input** (*tensor*) – Input tensor.
- **bits** (*int*) – Bit width.
- **overflow_rate** (*float*) – Overflow rate threshold for linear quanitzation.
- **quant_method** (*str*) – Quantization method. Must be in ['linear', 'log', 'tanh'].
- **min** (*float*) – Minimum value to clip values to.
- **max** (*float*) – Maximum value to clip values to.

Returns Quantized tensor.

Return type tensor

1.1.7 memtorch.bh.StochasticParameter

Methods to model stochastic parameters.

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 stochatic 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*, *resample_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 **

1.2 memtorch.map

Submodule containing various mapping algorithms.

1.2.1 memtorch.map.Module

Methods to determine relationships between a memristive crossbar and the output for a given memristive module.

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*) – 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

1.2.2 memtorch.map.Parameter

Methods to naively map network parameters to memristive device conductance's.

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*) – If not None, the proportion of weights to retain.

Returns Positive and negative crossbar weights.

Return type torch.Tensor, torch.Tensor

1.3 memtorch.mn

torch.nn equivalent submodule.

1.3.1 memtorch.mn.Module

Methods to convert and tune torch.nn models.

memtorch.mn.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*, *ADC_resolution=None*, *ADC_overflow_rate=0.0*, *quant_method=None*, *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.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quanitzation (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'minmax', 'tanh'], or None.
- **verbose** (*bool*) – Used to determine if verbose output is enabled (True) or disabled (False).

Returns Patched torch.nn.Module.

Return type torch.nn.Module

1.3.2 memtorch.mn.Linear

torch.nn.Linear equivalent.

class memtorch.mn.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*, *ADC_resolution=None*, *ADC_overflow_rate=0.0*, *quant_method=None*, *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.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quanitzation (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in [‘linear’, ‘log’, ‘log_minmax’, ‘minmax’, ‘tanh’], or None.
- **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.

1.3.3 memtorch.mn.Conv1d

torch.nn.Conv1d equivalent.

class memtorch.mn.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*, *ADC_resolution=None*, *ADC_overflow_rate=0.0*, *quant_method=None*, *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.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quanitzation (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in [‘linear’, ‘log’, ‘log_minmax’, ‘minmax’, ‘tanh’], or None.

- **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.

1.3.4 memtorch.mn.Conv2d

torch.nn.Conv2d equivalent.

class memtorch.mn.Conv2d.**Conv2d**(*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*, *ADC_resolution=None*, *ADC_overflow_rate=0.0*, *quant_method=None*, *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_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.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.

- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quanitzation (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'minmax', 'tanh'], or None.
- **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.

1.3.5 memtorch.mn.Conv3d

torch.nn.Conv3d equivalent.

class memtorch.mn.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*, *ADC_resolution=None*, *ADC_overflow_rate=0.0*, *quant_method=None*, *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.
- **ADC_resolution** (*int*) – ADC resolution (bit width). If None, quantization noise is not accounted for.
- **ADC_overflow_rate** (*float*) – Overflow rate threshold for linear quanitzation (if ADC_resolution is not None).
- **quant_method** – Quantization method. Must be in ['linear', 'log', 'log_minmax', 'min-max', 'tanh'], or None.
- **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.

CHAPTER 2

Tutorials

To learn how to use MemTorch, and to reproduce results of 'MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems', we provide numerous Jupyter notebooks.

Tutorial.ipynb details example usage of MemTorch to:

1. Train and benchmark a DNN model using the CIFAR-10 dataset.
2. Convert a pretrained DNN to a Memristive-DNN (MDNN).
3. Patch a converted Memristive-CNN.
4. Introduce non-ideal device characteristics to a generic memristive model.

CaseStudy.ipynb can be used to reproduce the results from the case study presented in 'MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems'.

NovelSimulations.ipynb can be used to reproduce the results from the novel simulations presented in 'MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems'.

2.1 MemTorch Tutorial

2.1.1 1. Training and benchmarking a DNN using CIFAR-10

The VGG-16 DNN architecture is trained and tested using the CIFAR-10 data set. The CIFAR-10 data set consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. The network is trained for 50 epochs with a batch size, $\Im = 256$. The initial learning rate is $\eta = 1e - 2$, which is decayed by an order of magnitude every 20 training epochs. Adam is used to optimize network parameters and Cross Entropy (CE) is used to determine network losses. *memtorch.utils.LoadCIFAR10* is used to load the CIFAR-10 training and test sets. After each epoch the model is bench-marked using the CIFAR-10 test set. The model that achieves the highest test set accuracy is saved as *trained_model.pt*.

```
[ ]: import torch
     from torch.autograd import Variable
```

(continues on next page)

(continued from previous page)

```
import memtorch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from memtorch.utils import LoadCIFAR10
import numpy as np


class Net(nn.Module):
    def __init__(self, inflation_ratio=1):
        super(Net, self).__init__()
        self.conv0 = nn.Conv2d(in_channels=3, out_channels=128*inflation_ratio,␣
↪kernel_size=3, stride=1, padding=1)
        self.bn0 = nn.BatchNorm2d(num_features=128*inflation_ratio)
        self.act0 = nn.ReLU()
        self.conv1 = nn.Conv2d(in_channels=128*inflation_ratio, out_
↪channels=128*inflation_ratio, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(num_features=128*inflation_ratio)
        self.act1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=128*inflation_ratio, out_
↪channels=256*inflation_ratio, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(num_features=256*inflation_ratio)
        self.act2 = nn.ReLU()
        self.conv3 = nn.Conv2d(in_channels=256*inflation_ratio, out_
↪channels=256*inflation_ratio, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(num_features=256*inflation_ratio)
        self.act3 = nn.ReLU()
        self.conv4 = nn.Conv2d(in_channels=256*inflation_ratio, out_
↪channels=512*inflation_ratio, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(num_features=512*inflation_ratio)
        self.act4 = nn.ReLU()
        self.conv5 = nn.Conv2d(in_channels=512*inflation_ratio, out_channels=512,␣
↪kernel_size=3, padding=1)
        self.bn5 = nn.BatchNorm2d(num_features=512)
        self.act5 = nn.ReLU()
        self.fc6 = nn.Linear(in_features=512*4*4, out_features=1024)
        self.bn6 = nn.BatchNorm1d(num_features=1024)
        self.act6 = nn.ReLU()
        self.fc7 = nn.Linear(in_features=1024, out_features=1024)
        self.bn7 = nn.BatchNorm1d(num_features=1024)
        self.act7 = nn.ReLU()
        self.fc8 = nn.Linear(in_features=1024, out_features=10)

    def forward(self, input):
        x = self.act0(self.bn0(self.conv0(input)))
        x = self.act1(self.bn1(F.max_pool2d(self.conv1(x), 2)))
        x = self.act2(self.bn2(self.conv2(x)))
        x = self.act3(self.bn3(F.max_pool2d(self.conv3(x), 2)))
        x = self.act4(self.bn4(self.conv4(x)))
        x = self.act5(self.bn5(F.max_pool2d(self.conv5(x), 2)))
        x = x.view(x.size(0), -1)
        x = self.act6(self.bn6(self.fc6(x)))
        x = self.act7(self.bn7(self.fc7(x)))
        return self.fc8(x)

def test(model, test_loader):
    correct = 0
```

(continues on next page)

(continued from previous page)

```
    for batch_idx, (data, target) in enumerate(test_loader):
        output = model(data.to(device))
        pred = output.data.max(1)[1]
        correct += pred.eq(target.to(device).data.view_as(pred)).cpu().sum()

    return 100. * float(correct) / float(len(test_loader.dataset))

device = torch.device('cpu' if 'cpu' in memtorch.__version__ else 'cuda')
epochs = 50
train_loader, validation_loader, test_loader = LoadCIFAR10(batch_size=256,␣
↪validation=False)
model = Net().to(device)
criterion = nn.CrossEntropyLoss()
learning_rate = 1e-2
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
best_accuracy = 0
for epoch in range(0, epochs):
    print('Epoch: [%d]\t\t' % (epoch + 1), end='')
    if epoch % 20 == 0:
        learning_rate = learning_rate * 0.1
        for param_group in optimizer.param_groups:
            param_group['lr'] = learning_rate

    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data.to(device))
        loss = criterion(output, target.to(device))
        loss.backward()
        optimizer.step()

    accuracy = test(model, test_loader)
    print('%2.2f%%' % accuracy)
    if accuracy > best_accuracy:
        torch.save(model.state_dict(), 'trained_model.pt')
        best_accuracy = accuracy
```

2.1.2 2. Conversion of a DNN to a MDNN

We use MemTorch to demonstrate the conversion of a DNN to MDNN. A memristive device model is defined and characterized below, which is used to replace all *torch.nn.Linear* layers within the DNN, trained in Step 1, with equivalent crossbar architectures using *memtorch.mn.Module.patch_model*.

```
[ ]: reference_memristor = memtorch.bh.memristor.VTEAM
reference_memristor_params = {'time_series_resolution': 1e-10}
memristor = reference_memristor(**reference_memristor_params)
memristor.plot_hysteresis_loop()
```

memtorch.bh.map.Parameter.naive_map is used to convert the weights within all *torch.nn.Linear* layers to equivalent conductance values, to be programmed to the two memristive devices used to represent each weight (positive and negative) using Eq. (13).

tile_shape is (128, 128), so modular crossbar tiles are used to represent weights. *ADC_resolution* sets the bit width of all emulated Analogue to Digital Converters (ADC); in this particular instance an 8-bit ADC resolution is used. *ADC_overflow* sets the initial overflow rate of each ADC. *quant_method* sets the quantization method used. *transistor*

is *True*, so a 1T1R arrangement is simulated. *programming_routine* is set to *None* to skip device-level simulation of the programming routine. We note if *transistor* is *False programming_routine* must not be *None*. In which case, device-level simulation is performed for each device using *memtorch.bh.crossbar.gen_programming_signal* and *memtorch.bh.memristor.Memristor.simulate*, which uses finite differences to model internal device dynamics. As *scheme* is not defined, a double-column parameter representation scheme is adopted. Finally, *max_input_voltage* is 1.0, so inputs to each layer are encoded between -1.0V and +1.0V.

All patched *torch.nn.Linear* layers are tuned using linear regression. A randomly generated tensor of size (8, *self.in_features*) is propagated through each memristive layer and each legacy layer (accessible using *layer.forward_legacy*). *sklearn.linear_model.LinearRegression* is used to determine the coefficient and intercept between the linear relationship of each set of outputs, which is used to define the *transform_output* lamdba function, that maps the output of each layer to their equivalent representations.

[ ]:
```
import copy
from memtorch.mn.Module import patch_model
from memtorch.map.Parameter import naive_map
from memtorch.bh.crossbar.Program import naive_program


model = Net().to(device)
model.load_state_dict(torch.load('trained_model.pt'), strict=False)
patched_model = patch_model(copy.deepcopy(model),
                            memristor_model=reference_memristor,
                            memristor_model_params=reference_memristor_params,
                            module_parameters_to_patch=[torch.nn.Linear],
                            mapping_routine=naive_map,
                            transistor=True,
                            programming_routine=None,
                            tile_shape=(128, 128),
                            max_input_voltage=1.0,
                            ADC_resolution=8,
                            ADC_overflow_rate=0.,
                            quant_method='linear')
```

[ ]:
```
patched_model.tune_()
```

[ ]:
```
print(test(patched_model, test_loader))
```

2.1.3 3. Modeling non-ideal device characteristics

We use a simple prototype model to demonstrate modeling non-ideal device characteristics. For sake of simplicity, from here-on-in, modular crossbar tiles are not used, and quantization noise is ignored.

[ ]:
```
from memtorch.mn.Module import patch_model
import copy
from memtorch.map.Parameter import naive_map

class Model(torch.nn.Module):

    def __init__(self):
        super(Model, self).__init__()
        self.convolutional_layer = torch.nn.Conv2d(in_channels=3, out_channels=1,␣
↪kernel_size=5)
        self.linear_layer = torch.nn.Linear(in_features=16, out_features=4)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, input):
        x = self.convolutional_layer(input)
        x = x.view(x.size(0), -1)
        return self.linear_layer(x)

torch.manual_seed(0)
model = Model().to(device)
reference_memristor_params = {'time_series_resolution': 1e-10, 'r_off': 200, 'r_on':␣
↪100}
patched_model = patch_model(copy.deepcopy(model),
                            memristor_model=reference_memristor,
                            memristor_model_params=reference_memristor_params,
                            module_parameters_to_patch=[torch.nn.Linear, torch.nn.
↪Conv2d],
                            mapping_routine=naive_map,
                            transistor=True,
                            programming_routine=None,
                            max_input_voltage=1.0)
```

Device-device variability is introduced to the VTEAM reference memristor model using *memtorch.bh.StochasticParameter*, by sampling R_{OFF} for each device from a normal distribution with $\sigma = 20$ and $x = 200$, and R_{ON} for each device from a normal distribution with $\sigma = 10$ and $x = 100$. Using *np.vectorize*, the R_{OFF} and R_{ON} values for each memristive device are compared.

[ ]:
```
from memtorch.mn.Module import patch_model
import copy
from memtorch.map.Parameter import naive_map
from memtorch.bh.crossbar.Program import naive_program
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

reference_memristor_params = {'time_series_resolution': 1e-10,
                              'r_off': memtorch.bh.StochasticParameter(loc=200,␣
↪scale=20, min=2),
                              'r_on': memtorch.bh.StochasticParameter(loc=100,␣
↪scale=10, min=1)}

patched_model_ = patch_model(copy.deepcopy(model),
                            memristor_model=reference_memristor,
                            memristor_model_params=reference_memristor_params,
                            module_parameters_to_patch=[torch.nn.Linear, torch.nn.
↪Conv2d],
                            mapping_routine=naive_map,
                            transistor=True,
                            programming_routine=None,
                            max_input_voltage=1.0)

A = torch.Tensor(np.vectorize(lambda x: x.r_off)(patched_model_.linear_layer.
↪crossbars[0].devices))
B = torch.Tensor(np.vectorize(lambda x: x.r_on)(patched_model_.linear_layer.
↪crossbars[0].devices))
C = torch.cat((A, B), 0)

plt.subplot(2, 1, 1)
plt.imshow(A.transpose(0, 1), interpolation='nearest', aspect=1, vmin=C.min(), vmax=C.
↪max(), cmap='seismic')
```

(continues on next page)

(continued from previous page)

```
plt.xticks([])
plt.yticks([])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 0')
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(cax=cax)

plt.subplot(2, 1, 2)
plt.imshow(B.transpose(0, 1), interpolation='nearest', aspect=1, vmin=C.min(), vmax=C.
↪max(), cmap='seismic')
plt.xticks([])
plt.yticks([])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 0')
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(cax=cax)
plt.savefig('var.svg')
plt.show()
```

We model a number (5) of finite discrete conductance states using $memtorch.bh.nonideality.NonIdeality.FiniteConductanceState$
The conductance levels within the positive crossbar were compared before and after a finite discrete conductance states are modeled.

```
[ ]: from memtorch.bh.nonideality.NonIdeality import apply_nonidealities

A = 1 / patched_model.linear_layer.crossbars[0].conductance_matrix
model = apply_nonidealities(copy.deepcopy(patched_model),
                                    non_idealities=[memtorch.bh.nonideality.NonIdeality.
↪FiniteConductanceStates],
                                    conductance_states = 5)
B = 1 / model.linear_layer.crossbars[0].conductance_matrix
C = torch.cat((A, B), 0)

plt.subplot(2, 1, 1)
plt.imshow(A.transpose(0, 1), interpolation='nearest', aspect=1, vmin=C.min(), vmax=C.
↪max(), cmap='seismic')
plt.xticks([])
plt.yticks([])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 0')
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(cax=cax)

plt.subplot(2, 1, 2)
plt.imshow(B.transpose(0, 1), interpolation='nearest', aspect=1, vmin=C.min(), vmax=C.
↪max(), cmap='seismic')
plt.xticks([])
plt.yticks([])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 0')
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(cax=cax)
```

(continues on next page)

(continued from previous page)

```
plt.savefig('finite.svg')
plt.show()
```

We model device failure using $memtorch.bh.nonideality.NonIdeality.DeviceFaults$. The conductance levels within the positive crossbar are once again compared, before and after 50% of devices are stuck at R_{LRS} and 50% of devices are stuck at R_{HRS}.

```
[ ]: from memtorch.bh.nonideality.NonIdeality import apply_nonidealities

A = 1 / patched_model.linear_layer.crossbars[0].conductance_matrix
model = apply_nonidealities(copy.deepcopy(patched_model),
                                    non_idealities=[memtorch.bh.nonideality.NonIdeality.
↪DeviceFaults],
                                    lrs_proportion=0.5,
                                    hrs_proportion=0.5,
                                    electroform_proportion=0)
B = 1 / model.linear_layer.crossbars[0].conductance_matrix
C = torch.cat((A, B), 0)

plt.subplot(2, 1, 1)
plt.imshow(A.transpose(0, 1), interpolation='nearest', aspect=1, vmin=C.min(), vmax=C.
↪max(), cmap='seismic')
plt.xticks([])
plt.yticks([])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 0')
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(cax=cax)

plt.subplot(2, 1, 2)
plt.imshow(B.transpose(0, 1), interpolation='nearest', aspect=1, vmin=C.min(), vmax=C.
↪max(), cmap='seismic')
plt.xticks([])
plt.yticks([])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 0')
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(cax=cax)
plt.savefig('fault.svg')
plt.show()
```

We model non-linear I/V characteristics using $memtorch.bh.nonideality.NonIdeality.NonLinear$ during inference. The output of the single *torch.nn.Linear* layer are compared when devices were simulated during inference with linear and non-linear I/V characteristics. Non-linear I/V characteristics were determined by applying a half-voltage sweep, using a sinusoidal cosine voltage signal with a duration of 5ns, amplitude of 1V, and a frequency of 50 MHz to each device.

```
[ ]: from memtorch.bh.nonideality.NonIdeality import apply_nonidealities
from sklearn.metrics.pairwise import cosine_similarity

A = torch.tensor(np.zeros((100, 4)))
B = torch.tensor(np.zeros((100, 4)))
for i in range(100):
  input = torch.zeros((1,3,8,8)).uniform_(-1, 1)
```

(continues on next page)

(continued from previous page)

```
  A[i, :] = patched_model(input)
  model = apply_nonidealities(copy.deepcopy(patched_model),
                              non_idealities=[memtorch.bh.nonideality.NonIdeality.
↪NonLinear],
                              sweep_duration=5e-9,
                              sweep_voltage_signal_amplitude=1,
                              sweep_voltage_signal_frequency=50e6)
  B[i, :] = model(input)

print(cosine_similarity([A.view(-1).numpy()], [B.view(-1).numpy()]))
```

2.2 A Case Study - Seizure Detection

2.2.1 1. Seizure detection dataset

[ ]:
```
import torch
from torch.utils.data import Dataset
import torch.nn.functional as F
import torch.nn as nn
import pandas as pd
import numpy as np
import sklearn
from sklearn import preprocessing


class SeizureDataset(Dataset):

    def __init__(self, path_to_csv):
        self.features = pd.read_csv(path_to_csv)
        self.labels = self.features.pop('y')
        self.features = preprocessing.scale(self.features.iloc[:, 1:], axis=0)

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, i):
        if self.labels[i] == 1:
            label = 1
        else:
            label = 0

        return np.asarray(self.features[i, :]).astype(np.float), label

csv_path = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00388/data.csv'
dataset = SeizureDataset(path_to_csv=csv_path)
```

2.2.2 2. Network architecture

[ ]:
```
class EEGNet(nn.Module):
    def __init__(self):
        super(EEGNet, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        self.fc1 = nn.Linear(178, 200)
        self.batchnorm1 = nn.BatchNorm1d(200)
        self.fc2 = nn.Linear(200, 200)
        self.batchnorm2 = nn.BatchNorm1d(200)
        self.fc3 = nn.Linear(200, 200)
        self.batchnorm3 = nn.BatchNorm1d(200)
        self.fc4 = nn.Linear(200, 2)
        self.batchnorm4 = nn.BatchNorm1d(2)

    def forward(self, x):
        x = self.batchnorm1(F.relu(self.fc1(x)))
        x = self.batchnorm2(F.relu(self.fc2(x)))
        x = self.batchnorm3(F.relu(self.fc3(x)))
        x = self.batchnorm4(self.fc4(x))
        return F.log_softmax(x, dim=1)
```

2.2.3 3. Training methodology

[ ]:
```
import sklearn
from sklearn.model_selection import KFold


init_lr = 1e-1
batch_size = 1024

def get_device():
    if torch.cuda.is_available():
        device = 'cuda:0'
    else:
        device = 'cpu'
    return device

def adjust_lr(optimizer, epoch):
    lr = init_lr * (0.1 ** (epoch // 20))
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr

    return lr

def train(net, train_loader, test_loader, epochs=10, fold=0):
    print('fold %d' % fold)
    best_f1_score = 0
    for epoch in range(epochs):
        lr = adjust_lr(optimizer, epoch)
        running_loss = 0
        for data in train_loader:
            inputs, labels = data
            inputs = inputs.float()
            if device == 'cuda:0':
                inputs = inputs.cuda()
                labels = labels.cuda()

            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
```

(continues on next page)

(continued from previous page)

```
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        f1_score = test(net, test_loader)
        if f1_score > best_f1_score:
            torch.save(net.state_dict(), 'trained_net_fold_%d.pt' % fold)
            best_f1_score = f1_score

        print('[Epoch %d] lr: %.4f f1_score: %.4f\ttraining loss: %.4f' % (epoch + 1,␣
↪lr, f1_score, running_loss / len(train_loader)))

def test(net, test_loader):
    confusion_matrix = torch.zeros(2, 2)
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data
            inputs = inputs.float()
            if device == 'cuda:0':
                inputs = inputs.cuda()
                labels = labels.cuda()

            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            for t, p in zip(labels.view(-1), predicted.view(-1)):
                confusion_matrix[t.long(), p.long()] += 1

            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    f1_score = 2 * confusion_matrix[0][0] / (2 * confusion_matrix[0][0] + confusion_
↪matrix[0][1] + confusion_matrix[1][0])
    return f1_score.item()

device = get_device()
dataset = SeizureDataset(csv_path)
torch.manual_seed(0)
kf = KFold(n_splits=5, shuffle=True)
train_loaders = []
test_loaders = []
for i, (train_index, test_index) in enumerate(kf.split(dataset)):
    train_ = torch.utils.data.Subset(dataset, train_index)
    test_ = torch.utils.data.Subset(dataset, test_index)
    train_loaders.append(torch.utils.data.DataLoader(train_, batch_size=batch_size,␣
↪shuffle=True))
    test_loaders.append(torch.utils.data.DataLoader(test_, batch_size=batch_size,␣
↪shuffle=False))

torch.manual_seed(torch.initial_seed())
torch.save(test_loaders, 'test_loaders.pth')
assert(len(train_loaders) == len(test_loaders))

# Determine the baseline F1 score
df = pd.DataFrame(columns=['indentifier', 'fold', 'f1_score'])
for indentifier in range(100):
```

(continues on next page)

(continued from previous page)

```
    test_loaders = torch.load('test_loaders.pth')
    for fold in range(len(test_loaders)):
        net = EEGNet()
        if torch.cuda.is_available():
            net = torch.nn.DataParallel(net)

        f1_score = test(net, test_loaders[fold])
        df = df.append({'indentifier': indentifier, 'fold': fold, 'f1_score': f1_
↪score}, ignore_index=True)

    df.to_csv('baseline.csv', index=False)

# Determine the F1 score
for fold in range(len(train_loaders)):
    net = EEGNet().to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=init_lr)
    train(net, train_loaders[fold], test_loaders[fold], epochs=50, fold=fold)

fold_f1_scores = []
for fold in range(len(train_loaders)):
    net = EEGNet().to(device)
    net.load_state_dict(torch.load('trained_net_fold_%d.pt' % fold), strict=True)
    fold_f1_scores.append(test(net, test_loaders[fold]))
    print('f1_score of fold %d: %0.4f' % (fold, fold_f1_scores[fold]))

print('baseline -> mean: %0.4f\tstddev: %0.4f' % (np.mean(df['f1_score'].values), np.
↪std(df['f1_score'].values)))
print('trained  -> mean: %0.4f\tstddev: %0.4f' % (np.mean(fold_f1_scores), np.
↪std(fold_f1_scores)))
```

2.2.4 4. Network conversion

```
[ ]: import memtorch
from memtorch.mn.Module import patch_model
from memtorch.map.Parameter import naive_map
from memtorch.bh.crossbar.Program import naive_program
from memtorch.bh.nonideality.NonIdeality import apply_nonidealities
import copy


test_loaders = torch.load('test_loaders.pth')
reference_memristor = memtorch.bh.memristor.VTEAM
reference_memristor_params = {'time_series_resolution': 1e-6,
                              'alpha_off': 1,
                              'alpha_on': 3,
                              'v_off': 0.5,
                              'v_on': -0.53,
                              'r_off': 2.5e3,
                              'r_on': 100,
                              'k_off': 4.03e-8,
                              'k_on': -80,
                              'd': 10e-9,
                              'x_on': 0,
```

(continues on next page)

(continued from previous page)

```
                                'x_off': 10e-9}

# Determine the first baseline F1 score
df = pd.DataFrame(columns=['indentifier', 'fold', 'f1_score'])
for indentifier in range(100):
    for fold in range(len(test_loaders)):
        net = EEGNet()
        net.load_state_dict(torch.load('trained_net_fold_%d.pt' % fold), strict=True)
        if torch.cuda.is_available():
            net = torch.nn.DataParallel(net)

        patched_net = patch_model(copy.deepcopy(net),
                                  memristor_model=reference_memristor,
                                  memristor_model_params=reference_memristor_params,
                                  module_parameters_to_patch=[torch.nn.Linear],
                                  mapping_routine=naive_map,
                                  transistor=True,
                                  programming_routine=None,
                                  scheme=memtorch.bh.Scheme.DoubleColumn)

        for i, (name, m) in enumerate(list(patched_net.named_modules())):
            if isinstance(m, memtorch.mn.Linear):
                m.crossbars[0].conductance_matrix = m.crossbars[0].conductance_matrix.
↪uniform_(1 / 2.5e3, 1 / 100)
                m.crossbars[1].conductance_matrix = m.crossbars[1].conductance_matrix.
↪uniform_(1 / 2.5e3, 1 / 100)

        patched_net.tune_()
        f1_score = test(patched_net, test_loaders[fold])
        df = df.append({'indentifier': indentifier, 'fold': fold, 'f1_score': f1_
↪score}, ignore_index=True)

    df.to_csv('baseline_variability.csv', index=False)

# Determine the second baseline F1 score
df_2 = pd.DataFrame(columns=['indentifier', 'fold', 'f1_score'])
for indentifier in range(100):
    for fold in range(len(test_loaders)):
        net = EEGNet()
        if torch.cuda.is_available():
            net = torch.nn.DataParallel(net)

        patched_net = patch_model(copy.deepcopy(net),
                                  memristor_model=reference_memristor,
                                  memristor_model_params=reference_memristor_params,
                                  module_parameters_to_patch=[torch.nn.Linear],
                                  mapping_routine=naive_map,
                                  transistor=True,
                                  programming_routine=None,
                                  scheme=memtorch.bh.Scheme.DoubleColumn)

        for i, (name, m) in enumerate(list(patched_net.named_modules())):
            if isinstance(m, memtorch.mn.Linear):
                m.crossbars[0].conductance_matrix = m.crossbars[0].conductance_matrix.
↪uniform_(1 / 2.5e3, 1 / 100)
                m.crossbars[1].conductance_matrix = m.crossbars[1].conductance_matrix.
↪uniform_(1 / 2.5e3, 1 / 100)
```

(continues on next page)

(continued from previous page)

```
        f1_score = test(patched_net, test_loaders[fold])
        df_2 = df_2.append({'indentifier': indentifier, 'fold': fold, 'f1_score': f1_
↪score}, ignore_index=True)

    df_2.to_csv('baseline_variability_no_tune.csv', index=False)

# Determine the F1 score
fold_f1_scores = []
for fold in range(len(test_loaders)):
    net = EEGNet()
    net.load_state_dict(torch.load('trained_net_fold_%d.pt' % fold), strict=False)
    if torch.cuda.is_available():
        net = torch.nn.DataParallel(net)

    patched_net = patch_model(net,
                              memristor_model=reference_memristor,
                              memristor_model_params=reference_memristor_params,
                              module_parameters_to_patch=[torch.nn.Linear],
                              mapping_routine=naive_map,
                              transistor=True,
                              programming_routine=None,
                              scheme=memtorch.bh.Scheme.DoubleColumn)

    patched_net.tune_()
    f1_score = test(patched_net, test_loaders[fold])
    fold_f1_scores.append(f1_score)

tuned_baseline = np.mean(df['f1_score'].values)
print('baseline          -> mean: %0.4f\tstddev: %0.4f' % (np.mean(df_2['f1_score'].
↪values), np.std(df_2['f1_score'].values)))
print('tuned baseline    -> mean: %0.4f\tstddev: %0.4f' % (np.mean(df['f1_score'].
↪values), np.std(df['f1_score'].values)))
print('trained and tuned -> mean: %0.4f\tstddev: %0.4f' % (np.mean(fold_f1_scores),␣
↪np.std(fold_f1_scores)))
```

[ ]:
```
import matplotlib.pyplot as plt


# Plot the hysteresis loop
hysteresis_loop_reference_memristor_params = {'time_series_resolution': 1e-6,
                              'alpha_off': 1,
                              'alpha_on': 3,
                              'v_off': 0.5,
                              'v_on': -0.53,
                              'r_off': memtorch.bh.StochasticParameter(2.5e3, std=50,␣
↪min=2),
                              'r_on': memtorch.bh.StochasticParameter(100, std=25,␣
↪min=1),
                              'k_off': 4.03e-8,
                              'k_on': -80,
                              'd': 10e-9,
                              'x_on': 0,
                              'x_off': 10e-9}

palette = ["#DA4453", "#8CC152", "#4A89DC", "#F6BB42", "#B600B0", "#535353"]
```

(continues on next page)

(continued from previous page)

```
f = plt.figure(figsize=(16/3, 4))
plt.title('Hysteresis Loop')
plt.xlabel('Voltage (V)')
plt.ylabel('Current (A)')
j = 0
for i in range(10):
    j = j + 1
    if j == 6:
        j = 0

    memristor = reference_memristor(**hysteresis_loop_reference_memristor_params)
    voltage_signal, current_signal = memristor.plot_hysteresis_loop(duration=2,␣
↪voltage_signal_amplitude=1, voltage_signal_frequency = 0.5, return_result=True)
    plt.plot(voltage_signal, current_signal, color=palette[j])

plt.grid()
plt.show()
```

2.2.5 5. Device-to-device variability investigation

```
[ ]: # Determine the F1 score
non_linear_reference_memristor_params = {'time_series_resolution': 1e-6,
                                'alpha_off': 1,
                                'alpha_on': 3,
                                'v_off': 0.5,
                                'v_on': -0.53,
                                'r_off': memtorch.bh.StochasticParameter(2.5e3,␣
↪std=sigma*2, min=1),
                                'r_on': memtorch.bh.StochasticParameter(100, std=sigma,␣
↪min=1),
                                'k_off': 4.03e-8,
                                'k_on': -80,
                                'd': 10e-9,
                                'x_on': 0,
                                'x_off': 10e-9}

df = pd.DataFrame(columns=['sigma', 'mean', 'std'])
sigma_values = np.linspace(0, 500, 21)
for sigma in sigma_values:
    f1_scores = []
    for fold in range(len(test_loaders)):
        net = EEGNet()
        net.load_state_dict(torch.load('trained_net_fold_%d.pt' % fold), strict=True)
        if torch.cuda.is_available():
            net = torch.nn.DataParallel(net)

        patched_net = patch_model(copy.deepcopy(net),
                                  memristor_model=reference_memristor,
                                  memristor_model_params=non_linear_reference_
↪memristor_params,
                                  module_parameters_to_patch=[torch.nn.Linear],
                                  mapping_routine=naive_map,
                                  transistor=True,
                                  programming_routine=None,
                                  scheme=memtorch.bh.Scheme.DoubleColumn)
```

(continues on next page)

(continued from previous page)

```
        patched_net.tune_()
        f1_score = test(patched_net, test_loaders[fold])
        f1_scores.append(f1_score)

    df = df.append({'sigma': sigma, 'mean': np.mean(f1_scores), 'std': np.std(f1_
↪scores)}, ignore_index=True)

df.to_csv('variability.csv', index=False)
```

```
[ ]: f = plt.figure(figsize=(16/3, 4))
plt.axhline(y=tuned_baseline, color='k', linestyle='--', zorder=1)
b = plt.bar(df['sigma'], df['mean'], width=12.5, zorder=2)
plt.xlabel('$\sigma$')
plt.ylabel('F1 Score')
for bar in b:
    bar.set_edgecolor('black')
    bar.set_facecolor(palette[0])
    bar.set_linewidth(1)

f.tight_layout()
plt.grid()
plt.ylim([0.9, 1.0])
plt.show()
```

2.2.6 6. Non-linear IV characteristics investigation

```
[ ]: # Determine the F1 score
df = pd.DataFrame(columns=['sigma', 'mean', 'std'])
sigma_values = np.linspace(0, 500, 11)
f1_scores = []
for fold in range(len(test_loaders)):
    net = EEGNet()
    net.load_state_dict(torch.load('trained_net_fold_%d.pt' % fold), strict=False)
    if torch.cuda.is_available():
        net = torch.nn.DataParallel(net)

    patched_net = patch_model(net,
                              memristor_model=reference_memristor,
                              memristor_model_params=reference_memristor_params,
                              module_parameters_to_patch=[torch.nn.Linear],
                              mapping_routine=naive_map,
                              transistor=True,
                              programming_routine=None,
                              scheme=memtorch.bh.Scheme.DoubleColumn)

    patched_net = apply_nonidealities(patched_net,
                    non_idealities=[memtorch.bh.nonideality.NonIdeality.NonLinear],
                    sweep_duration=2,
                    sweep_voltage_signal_amplitude=1,
                    sweep_voltage_signal_frequency=0.5)

    patched_net.tune_()
    f1_score = test(patched_net, test_loaders[fold])
```

(continues on next page)

(continued from previous page)

```
    f1_scores.append(f1_score)

print('mean: %0.4f\tstddev: %0.4f' % (np.mean(f1_scores), np.std(f1_scores)))
```

2.3 Novel Simulations

2.3.1 1. Define and train a VGG Convolutional Neural Network (CNN) using CIFAR-10

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import memtorch
from memtorch.utils import LoadCIFAR10


class Net(nn.Module):
    def __init__(self, inflation_ratio=1):
        super(Net, self).__init__()
        self.conv0 = nn.Conv2d(in_channels=3, out_channels=128*inflation_ratio,␣
↪kernel_size=3, stride=1, padding=1)
        self.bn0 = nn.BatchNorm2d(num_features=128*inflation_ratio)
        self.act0 = nn.ReLU()
        self.conv1 = nn.Conv2d(in_channels=128*inflation_ratio, out_
↪channels=128*inflation_ratio, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(num_features=128*inflation_ratio)
        self.act1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=128*inflation_ratio, out_
↪channels=256*inflation_ratio, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(num_features=256*inflation_ratio)
        self.act2 = nn.ReLU()
        self.conv3 = nn.Conv2d(in_channels=256*inflation_ratio, out_
↪channels=256*inflation_ratio, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(num_features=256*inflation_ratio)
        self.act3 = nn.ReLU()
        self.conv4 = nn.Conv2d(in_channels=256*inflation_ratio, out_
↪channels=512*inflation_ratio, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(num_features=512*inflation_ratio)
        self.act4 = nn.ReLU()
        self.conv5 = nn.Conv2d(in_channels=512*inflation_ratio, out_channels=512,␣
↪kernel_size=3, padding=1)
        self.bn5 = nn.BatchNorm2d(num_features=512)
        self.act5 = nn.ReLU()
        self.fc6 = nn.Linear(in_features=512*4*4, out_features=1024)
        self.bn6 = nn.BatchNorm1d(num_features=1024)
        self.act6 = nn.ReLU()
        self.fc7 = nn.Linear(in_features=1024, out_features=1024)
        self.bn7 = nn.BatchNorm1d(num_features=1024)
        self.act7 = nn.ReLU()
        self.fc8 = nn.Linear(in_features=1024, out_features=10)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, input):
        x = self.act0(self.bn0(self.conv0(input)))
        x = self.act1(self.bn1(F.max_pool2d(self.conv1(x), 2)))
        x = self.act2(self.bn2(self.conv2(x)))
        x = self.act3(self.bn3(F.max_pool2d(self.conv3(x), 2)))
        x = self.act4(self.bn4(self.conv4(x)))
        x = self.act5(self.bn5(F.max_pool2d(self.conv5(x), 2)))
        x = x.view(x.size(0), -1)
        x = self.act6(self.bn6(self.fc6(x)))
        x = self.act7(self.bn7(self.fc7(x)))
        return self.fc8(x)


def test(model, test_loader):
    correct = 0
    for batch_idx, (data, target) in enumerate(test_loader):
        output = model(data.to(device))
        pred = output.data.max(1)[1]
        correct += pred.eq(target.to(device).data.view_as(pred)).cpu().sum()

    return 100. * float(correct) / float(len(test_loader.dataset))

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
epochs = 50
train_loader, validation_loader, test_loader = LoadCIFAR10(batch_size=256,␣
↪validation=False)
model = Net().to(device)
if device == 'cuda':
    model = torch.nn.DataParallel(model)

criterion = nn.CrossEntropyLoss()
learning_rate = 1e-2
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
best_accuracy = 0
for epoch in range(0, epochs):
    print('Epoch: [%d]\t\t' % (epoch + 1), end='')
    if epoch % 20 == 0:
        learning_rate = learning_rate * 0.1
        for param_group in optimizer.param_groups:
            param_group['lr'] = learning_rate

    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data.to(device))
        loss = criterion(output, target.to(device))
        loss.backward()
        optimizer.step()

    accuracy = test(model, test_loader)
    print('%2.2f%%' % accuracy)
    if accuracy > best_accuracy:
        torch.save(model.state_dict(), 'trained_model.pt')
        best_accuracy = accuracy
```

2.3.2 2. Load and test the network

```
[ ]: import copy
     import pandas as pd
     import matplotlib
     import matplotlib.pyplot as plt
     import numpy as np


     model = Net().to(device)
     try:
         model.load_state_dict(torch.load('trained_model.pt'), strict=False)
     except:
         raise Exception('trained_model.pt has not been found.')

     print('Test Set Accuracy: \t%2.2f%%' % test(model, test_loader))
```

2.3.3 3. Import seaborn and define an appropriate color-palette

```
[ ]: import seaborn as sns


     palette = ["#DA4453", "#8CC152", "#4A89DC", "#F6BB42", "#B600B0", "#535353"]
```

2.3.4 4. Device-device variability investigation

```
[ ]: from memtorch.mn.Module import patch_model
     from memtorch.map.Parameter import naive_map
     from memtorch.bh.crossbar.Program import naive_program
     from memtorch.bh.nonideality.NonIdeality import apply_nonidealities


     def trial(r_on, r_off, sigma):
         model_ = copy.deepcopy(model)
         reference_memristor = memtorch.bh.memristor.VTEAM
         reference_memristor_params = {'time_series_resolution': 1e-10,
                                       'r_off': memtorch.bh.StochasticParameter(r_off,␣
     ↪std=sigma*2, min=1),
                                       'r_on': memtorch.bh.StochasticParameter(r_on,␣
     ↪std=sigma, min=1)}

         patched_model = patch_model(copy.deepcopy(model_),
                                     memristor_model=reference_memristor,
                                     memristor_model_params=reference_memristor_params,
                                     module_parameters_to_patch=[torch.nn.Linear, torch.nn.
     ↪Conv2d],
                                     mapping_routine=naive_map,
                                     transistor=True,
                                     programming_routine=None)

         patched_model.tune_()
         return test(patched_model, test_loader)
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame(columns=['sigma', 'test_set_accuracy'])
r_on = 200
r_off = 500
sigma_values = np.linspace(0, 100, 21)
for sigma in sigma_values:
    df = df.append({'sigma': sigma, 'test_set_accuracy': trial(r_on, r_off, sigma)},␣
↪ignore_index=True)

df.to_csv('variability.csv', index=False)
```

```
[ ]: df = pd.read_csv('variability.csv')
f = plt.figure()
plt.axhline(y=10, color='k', linestyle='--', zorder=1)
b = plt.bar(df['sigma'], df['test_set_accuracy'], width=2.5, zorder=2)
plt.xlabel('$\sigma$')
plt.ylabel('CIFAR-10 Test-set Accuracy (%)')
for bar in b:
    bar.set_edgecolor('black')
    bar.set_facecolor(palette[0])
    bar.set_linewidth(1)

f.tight_layout()
plt.grid()
plt.savefig("P1.svg")
plt.show()
```

2.3.5 5. Finite conductance states investigation

```
[ ]: from memtorch.mn.Module import patch_model
from memtorch.map.Parameter import naive_map
from memtorch.bh.crossbar.Program import naive_program
from memtorch.bh.nonideality.NonIdeality import apply_nonidealities


def trial(r_on, r_off, finite_states):
    model_ = copy.deepcopy(model)
    reference_memristor = memtorch.bh.memristor.VTEAM
    reference_memristor_params = {'time_series_resolution': 1e-10,
                                  'r_off': r_off,
                                  'r_on': r_on}

    patched_model = patch_model(copy.deepcopy(model_),
                                memristor_model=reference_memristor,
                                memristor_model_params=reference_memristor_params,
                                module_parameters_to_patch=[torch.nn.Linear, torch.nn.
↪Conv2d],
                                mapping_routine=naive_map,
                                transistor=True,
                                programming_routine=None)

    patched_model = apply_nonidealities(patched_model,
                               non_idealities=[memtorch.bh.nonideality.NonIdeality.
↪FiniteConductanceStates],
                               conductance_states = int(finite_states))
```

(continues on next page)

(continued from previous page)

```
    patched_model.tune_()
    return test(patched_model, test_loader)

df = pd.DataFrame(columns=['finite_states', 'test_set_accuracy'])
r_on = 200
r_off = 500
finite_state_values = np.linspace(1, 10, 10)
for finite_states in finite_state_values:
    df = df.append({'finite_states': finite_states, 'test_set_accuracy': trial(r_on,␣
↪r_off, finite_states)}, ignore_index=True)

df.to_csv('finite_states.csv', index=False)
```

[ ]:
```
df = pd.read_csv('finite_states.csv')
f = plt.figure()
plt.axhline(y=10, color='k', linestyle='--', zorder=1)
b = plt.bar(df['finite_states'], df['test_set_accuracy'], width=0.5, zorder=2)
plt.xlabel('Number of Finite States')
plt.ylabel('CIFAR-10 Test-set Accuracy (%)')
plt.xticks(df['finite_states'])
for bar in b:
    bar.set_edgecolor('black')
    bar.set_facecolor(palette[0])
    bar.set_linewidth(1)

f.tight_layout()
plt.grid()
plt.savefig("P2.svg")
plt.show()
```

2.3.6 6. Device failure investigation

[ ]:
```
from memtorch.mn.Module import patch_model
from memtorch.map.Parameter import naive_map
from memtorch.bh.crossbar.Program import naive_program
from memtorch.bh.nonideality.NonIdeality import apply_nonidealities


def trial(r_on, r_off, lrs_proportion, hrs_proportion):
    model_ = copy.deepcopy(model)
    reference_memristor = memtorch.bh.memristor.VTEAM
    reference_memristor_params = {'time_series_resolution': 1e-10,
                                  'r_off': r_off,
                                  'r_on': r_on}

    patched_model = patch_model(copy.deepcopy(model_),
                                memristor_model=reference_memristor,
                                memristor_model_params=reference_memristor_params,
                                module_parameters_to_patch=[torch.nn.Linear, torch.nn.
↪Conv2d],
                                mapping_routine=naive_map,
                                transistor=True,
                                programming_routine=None)
```

(continues on next page)

(continued from previous page)

```
    patched_model = apply_nonidealities(patched_model,
                                non_idealities=[memtorch.bh.nonideality.NonIdeality.
↪DeviceFaults],
                                lrs_proportion=lrs_proportion,
                                hrs_proportion=hrs_proportion,
                                electroform_proportion=0)

    patched_model.tune_()
    return test(patched_model, test_loader)

df_lrs_hrs = pd.DataFrame(columns=['failure_percentage', 'test_set_accuracy'])
df_lrs = pd.DataFrame(columns=['failure_percentage', 'test_set_accuracy'])
df_hrs = pd.DataFrame(columns=['failure_percentage', 'test_set_accuracy'])
r_on = 200
r_off = 500
failures = np.linspace(0, 0.25, 11)

for failure in failures:
    df_lrs_hrs = df_lrs_hrs.append({'failure_percentage': failure, 'test_set_accuracy
↪': trial(r_on, r_off, failure, failure)}, ignore_index=True)
    df_lrs = df_lrs.append({'failure_percentage': failure, 'test_set_accuracy':␣
↪trial(r_on, r_off, failure, 0)}, ignore_index=True)
    df_hrs = df_hrs.append({'failure_percentage': failure, 'test_set_accuracy':␣
↪trial(r_on, r_off, 0, failure)}, ignore_index=True)

df_lrs_hrs.to_csv('failure_lrs_hrs.csv', index=False)
df_lrs.to_csv('failure_lrs.csv', index=False)
df_hrs.to_csv('failure_hrs.csv', index=False)
```

[ ]:
```
import seaborn as sns
from matplotlib.ticker import FormatStrFormatter


df_lrs_hrs = pd.read_csv('failure_lrs_hrs.csv')
df_lrs = pd.read_csv('failure_lrs.csv')
df_hrs = pd.read_csv('failure_hrs.csv')
f = plt.figure()
plt.axhline(y=10, color='k', linestyle='--', zorder=1)
concat = pd.concat([df_lrs_hrs['failure_percentage'],
                    df_lrs_hrs['test_set_accuracy'],
                    df_lrs['test_set_accuracy'],
                    df_hrs['test_set_accuracy']],
                    axis=1)
concat.columns = ['failure_percentage', 'lrs_hrs', 'lrs', 'hrs']
data = pd.DataFrame(columns=['failure_percentage', 'state', 'test_set_accuracy'])
for index, row in concat.iterrows():
    data = data.append({'failure_percentage': row['failure_percentage'] * 100, 'state
↪': 'lrs_hrs', 'test_set_accuracy': row['lrs_hrs']}, ignore_index=True)
    data = data.append({'failure_percentage': row['failure_percentage'] * 100, 'state
↪': 'lrs', 'test_set_accuracy': row['lrs']}, ignore_index=True)
    data = data.append({'failure_percentage': row['failure_percentage'] * 100, 'state
↪': 'hrs', 'test_set_accuracy': row['hrs']}, ignore_index=True)

data['state'] = data['state'].map({'lrs_hrs': '$R_{ON}$ and $R_{OFF}$', 'lrs': '$R_
↪{ON}$', 'hrs': '$R_{OFF}$'})
```

(continues on next page)

(continued from previous page)

```
h = sns.barplot(x="failure_percentage", hue="state", y="test_set_accuracy", data=data,
↪ zorder=2, edgecolor='black', linewidth='1', palette=sns.color_palette(palette),␣
↪saturation=1)
plt.gca().xaxis.set_major_formatter(FormatStrFormatter('%1.1f'))
plt.xticks(np.arange(11), np.arange(0, 25 + 2.5, step=2.5))
h.legend(loc=1)
plt.xlabel('Device Failure (%)')
plt.ylabel('CIFAR-10 Test-set Accuracy (%)')
f.tight_layout()
plt.grid()
plt.savefig("P3.svg")
plt.show()
```

2.3.7 7. First novel simulation

```
[ ]: from memtorch.mn.Module import patch_model
from memtorch.map.Parameter import naive_map
from memtorch.bh.crossbar.Program import naive_program
from memtorch.bh.nonideality.NonIdeality import apply_nonidealities


def trial(num_conductance_states, g_ratio, sigma):
    model_ = copy.deepcopy(model)
    r_on = 200
    reference_memristor = memtorch.bh.memristor.VTEAM
    reference_memristor_params = {'time_series_resolution': 1e-10,
                                  'r_off': memtorch.bh.StochasticParameter(r_on * g_
↪ratio, std=sigma*2, min=1),
                                  'r_on': memtorch.bh.StochasticParameter(r_on,␣
↪std=sigma, min=1)}
    patched_model = patch_model(copy.deepcopy(model_),
                                memristor_model=reference_memristor,
                                memristor_model_params=reference_memristor_params,
                                module_parameters_to_patch=[torch.nn.Linear, torch.nn.
↪Conv2d],
                                mapping_routine=naive_map,
                                transistor=True,
                                programming_routine=None)
    patched_model = apply_nonidealities(patched_model,
                                non_idealities=[memtorch.bh.nonideality.NonIdeality.
↪FiniteConductanceStates],
                                conductance_states=int(num_conductance_states))
    patched_model.tune_()
    return test(patched_model, test_loader)

std_devs = [0, 20, 100]
g_ratios = [2 ** n for n in range(6)]
conductance_states = np.linspace(2, 10, 9)
for std_dev in std_devs:
    df = pd.DataFrame(columns=['conductance_states', 'g_ratio', 'test_set_accuracy'])
    for g_ratio in g_ratios:
        for num_conductance_states in conductance_states:
            test_set_accuracy = trial(num_conductance_states, g_ratio, std_dev)
            df = df.append({'conductance_states': num_conductance_states,
                            'g_ratio': g_ratio,
```

(continues on next page)

(continued from previous page)

```
                                'test_set_accuracy': test_set_accuracy}, ignore_
↪index=True)

    df.to_csv('S1_std_dev_%d.csv' % std_dev, index=False)
```

[ ]:
```
import seaborn as sns
from matplotlib.ticker import FormatStrFormatter

f = plt.figure(figsize=(16, 4))
plt.subplot(1, len(std_devs), 1)
for plot_index, std_dev in enumerate(std_devs):
    plt.subplot(1, len(std_devs), plot_index + 1)
    plt.axhline(y=10, color='k', linestyle='--', zorder=1)
    data = pd.read_csv('S1_std_dev_%d.csv' % std_dev)
    h = sns.barplot(x="conductance_states", hue="g_ratio", y="test_set_accuracy",␣
↪data=data, zorder=2, edgecolor='black', linewidth='1', palette=sns.color_
↪palette(palette), saturation=1)
    plt.gca().xaxis.set_major_formatter(FormatStrFormatter('%1.0f'))
    plt.xticks(np.arange(0, len(conductance_states)), map(lambda n: "%d" % n,␣
↪conductance_states))
    leg = h.axes.get_legend()
    leg.set_title('RON/ROFF Ratio')
    h.legend(loc=1)
    plt.title('$\sigma$ = %d' % std_dev)
    if plot_index == 0:
        plt.xlabel('Number of Finite States')
        plt.ylabel('CIFAR-10 Test-set Accuracy (%)')
    else:
        plt.xlabel('')
        plt.ylabel('')

    plt.grid()

f.tight_layout()
plt.savefig("S1.svg")
plt.show()
```

2.3.8 8. Second novel simulation

[ ]:
```
from memtorch.mn.Module import patch_model
from memtorch.map.Parameter import naive_map
from memtorch.bh.crossbar.Program import naive_program
from memtorch.bh.nonideality.NonIdeality import apply_nonidealities


def trial(num_conductance_states, lrs_failure_rate, hrs_failure_rate, sigma):
    model_ = copy.deepcopy(model)
    r_on = 200
    r_off = 500
    reference_memristor = memtorch.bh.memristor.VTEAM
    reference_memristor_params = {'time_series_resolution': 1e-10,
                                  'r_off': memtorch.bh.StochasticParameter(r_on * g_
↪ratio, std=sigma*2, min=1),
                                  'r_on': memtorch.bh.StochasticParameter(r_off,␣
↪std=sigma, min=1)}
```

(continues on next page)

(continued from previous page)

```
    patched_model = patch_model(copy.deepcopy(model_),
                                memristor_model=reference_memristor,
                                memristor_model_params=reference_memristor_params,
                                module_parameters_to_patch=[torch.nn.Linear, torch.nn.
↪Conv2d],
                                mapping_routine=naive_map,
                                transistor=True,
                                programming_routine=None)
    patched_model = apply_nonidealities(patched_model,
                                non_idealities=[memtorch.bh.nonideality.NonIdeality.
↪FiniteConductanceStates,
                                                memtorch.bh.nonideality.NonIdeality.
↪DeviceFaults],
                                conductance_states=int(num_conductance_states),
                                lrs_proportion=lrs_failure_rate,
                                hrs_proportion=hrs_failure_rate,
                                electroform_proportion=0)
    patched_model.tune_()
    return test(patched_model, test_loader)

std_devs = [0, 20, 100]
failure_rates = np.linspace(0, 0.25, 6)
conductance_states = np.linspace(2, 10, 9)

# LRS
for std_dev in std_devs:
    df = pd.DataFrame(columns=['conductance_states', 'failure_rate', 'test_set_
↪accuracy'])
    for failure_rate in failure_rates:
        for num_conductance_states in conductance_states:
            test_set_accuracy = trial(num_conductance_states, failure_rate, 0, std_
↪dev)
            df = df.append({'conductance_states': num_conductance_states,
                            'failure_rate': failure_rate,
                            'test_set_accuracy': test_set_accuracy}, ignore_
↪index=True)

    df.to_csv('S2_LRS_std_dev_%d.csv' % std_dev, index=False)

# HRS
for std_dev in std_devs:
    df = pd.DataFrame(columns=['conductance_states', 'failure_rate', 'test_set_
↪accuracy'])
    for failure_rate in failure_rates:
        for num_conductance_states in conductance_states:
            test_set_accuracy = trial(num_conductance_states, 0, failure_rate, std_
↪dev)
            df = df.append({'conductance_states': num_conductance_states,
                            'failure_rate': failure_rate,
                            'test_set_accuracy': test_set_accuracy}, ignore_
↪index=True)

    df.to_csv('S2_HRS_std_dev_%d.csv' % std_dev, index=False)

# LRS and HRS
for std_dev in std_devs:
    df = pd.DataFrame(columns=['conductance_states', 'failure_rate', 'test_set_
↪accuracy'])
```

(continues on next page)

(continued from previous page)

```
    for failure_rate in failure_rates:
        for num_conductance_states in conductance_states:
            test_set_accuracy = trial(num_conductance_states, failure_rate, failure_
↪rate, std_dev)
            df = df.append({'conductance_states': num_conductance_states,
                            'failure_rate': failure_rate,
                            'test_set_accuracy': test_set_accuracy}, ignore_
↪index=True)

    df.to_csv('S2_LRS_HRS_std_dev_%d.csv' % std_dev, index=False)
```

[ ]:
```
import seaborn as sns
from matplotlib.ticker import FormatStrFormatter

f = plt.figure(figsize=(16, 12))
plt.subplot(3, len(std_devs), 1)
# LRS
for plot_index, std_dev in enumerate(std_devs):
    plt.subplot(3, len(std_devs), plot_index + 1)
    plt.axhline(y=10, color='k', linestyle='--', zorder=1)
    data = pd.read_csv('S2_LRS_std_dev_%d.csv' % std_dev)
    h = sns.barplot(x="conductance_states", hue="failure_rate", y="test_set_accuracy",
↪ data=data, zorder=2, edgecolor='black', linewidth='1', palette=sns.color_
↪palette(palette), saturation=1)
    plt.gca().xaxis.set_major_formatter(FormatStrFormatter('%1.0f'))
    plt.xticks(np.arange(0, len(conductance_states)), map(lambda n: "%d" % n,␣
↪conductance_states))
    leg = h.axes.get_legend()
    leg.set_title('Device Failure (%)')
    h.legend(loc=1)
    plt.title('$\sigma$ = %d' % std_dev)
    if plot_index == 0:
        plt.xlabel('Number of Finite States')
        plt.ylabel('[LRS] CIFAR-10 Test-set Accuracy (%)')
    else:
        plt.xlabel('')
        plt.ylabel('')

    plt.grid()

# HRS
for plot_index, std_dev in enumerate(std_devs):
    plt.subplot(3, len(std_devs), plot_index + 1 + len(std_devs))
    plt.axhline(y=10, color='k', linestyle='--', zorder=1)
    data = pd.read_csv('S2_HRS_std_dev_%d.csv' % std_dev)
    h = sns.barplot(x="conductance_states", hue="failure_rate", y="test_set_accuracy",
↪ data=data, zorder=2, edgecolor='black', linewidth='1', palette=sns.color_
↪palette(palette), saturation=1)
    plt.gca().xaxis.set_major_formatter(FormatStrFormatter('%1.0f'))
    plt.xticks(np.arange(0, len(conductance_states)), map(lambda n: "%d" % n,␣
↪conductance_states))
    leg = h.axes.get_legend()
    leg.set_title('Device Failure (%)')
    h.legend(loc=1)
    plt.title('$\sigma$ = %d' % std_dev)
    if plot_index == 0:
```

(continues on next page)

(continued from previous page)

```
        plt.xlabel('Number of Finite States')
        plt.ylabel('[HRS] CIFAR-10 Test-set Accuracy (%)')
    else:
        plt.xlabel('')
        plt.ylabel('')

    plt.grid()

# LRS and HRS
for plot_index, std_dev in enumerate(std_devs):
    plt.subplot(3, len(std_devs), plot_index + 1 + 2 * len(std_devs))
    plt.axhline(y=10, color='k', linestyle='--', zorder=1)
    data = pd.read_csv('S2_LRS_HRS_std_dev_%d.csv' % std_dev)
    h = sns.barplot(x="conductance_states", hue="failure_rate", y="test_set_accuracy",
↪ data=data, zorder=2, edgecolor='black', linewidth='1', palette=sns.color_
↪palette(palette), saturation=1)
    plt.gca().xaxis.set_major_formatter(FormatStrFormatter('%1.0f'))
    plt.xticks(np.arange(0, len(conductance_states)), map(lambda n: "%d" % n,␣
↪conductance_states))
    leg = h.axes.get_legend()
    leg.set_title('Device Failure (%)')
    h.legend(loc=1)
    plt.title('$\sigma$ = %d' % std_dev)
    if plot_index == 0:
        plt.xlabel('Number of Finite States')
        plt.ylabel('[LRS and HRS] CIFAR-10 Test-set Accuracy (%)')
    else:
        plt.xlabel('')
        plt.ylabel('')

    plt.grid()

f.tight_layout()
plt.savefig("S2.svg")
plt.show()
```

Python Module Index

m

Index

N

P

Q

R

S

T

U

V

W