Contents

1 srnn.py.py

```
#! /usr/bin/env python
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
# SPDX-License-Identifier: MPL-2.0
#

"""  This is the BPTT spiking recurrent neural network (srnn)
     example using the Google Speech Commands dataset.  """

import pprint
import random
import sys
from pathlib import Path

import numpy as np

# import snoop  # for debugging when something goes wrong.
import torch
import torch.nn.functional as F
import torchaudio
import torchvision
from loguru import logger
from torch import nn
from torch.optim.lr_scheduler import StepLR

# srnn specific modules
import efficient_spiking_networks.srnn_layers.spike_dense as sd
```

```
import efficient_spiking_networks.srnn_layers.spike_neuron as sn
import efficient_spiking_networks.srnn_layers.spike_rnn as sr

# modules that are associated with this example.
from GSC.data import (  # pylint: disable=C0301
    GSCSSubsetSC,
    MelSpectrogram,
    Normalize,
    Pad,
    Rescale,
)
from GSC.utils import generate_noise_files

# Two generator functions; part of our utilities suite
from utilities.gendfind import gen_dfind  # pylint: disable=C0411
from utilities.genfind import gen_find  # pylint: disable=C0411

# Setup pretty printing
pp = pprint.PrettyPrinter(indent=4, compact=True, width=42)

# Setup logger level
logger.remove()
logger.add(sys.stderr, level="INFO")

# Use cuda if it's available
device = torch.device(  # pylint: disable=E1101
    "cuda:0" if torch.cuda.is_available() else "cpu"
)
logger.info(f"{device=}")
```

```
# Setup number of workers dependent upon where the code is run
NUMBER_OF_WORKERS = 4 if device.type == "cpu" else 8
PIN_MEMORY = device.type == "cuda"
logger.info(f"The Dataloader will spawn {NUMBER_OF_WORKERS} worker processes.")
logger.info(f"{PIN_MEMORY=}")

# Specify the several paramaters that we'll use throughout this example.
# Paths to the data
DATAROOT = Path("google")
GSC_URL = "speech_commands_v0.01"  # speech_commands_v0.02
GSC = DATAROOT / "SpeechCommands" / GSC_URL
logger.info("\n".join([f"{DATAROOT=}", f"{GSC_URL=}", f"{GSC=}"]))

# Specify the learning rate, etc.
LEARNING_RATE = 3e-3  # 1.2e-2
EPOCHS = 30
BATCH_SIZE = 32
SIZE = 16000
SR = 16000  # Sampling Rate 16Hz
IS_BIAS = True
logger.info(
    "\n".join(
        [
            f"\n{LEARNING_RATE=}",
            f"{EPOCHS=}",
            f"{BATCH_SIZE=}",
            f"{SIZE=}",
            f"{SR=}",
            f"{IS_BIAS=}",
```

```
        ]
    )
)

# Parameters for converting a wav into a mel-scaled spectrogram
DELTA_ORDER = 2
FMAX = 4000
FMIN = 20
HOP_LENGTH = int(10e-3 * SR)
N_FFT = int(30e-3 * SR)
N_MELS = 40
STACK = True
MELSPEC = MelSpectrogram(
    SR, N_FFT, HOP_LENGTH, N_MELS, FMIN, FMAX, DELTA_ORDER, stack=STACK
)
logger.info(
    "\n".join(
        [
            f"\n{DELTA_ORDER=}",
            f"{FMAX=}",
            f"{FMIN=}",
            f"{HOP_LENGTH=}",
            f"{N_FFT=}",
            f"{N_MELS=}",
            f"{STACK=}",
        ]
    )
)
```

```
# Compose transformations applied to each dataset
PAD = Pad(SIZE)
RESCALE = Rescale()
NORMALIZE = Normalize()
TRANSFORMS = torchvision.transforms.Compose([PAD, MELSPEC, RESCALE])

# Specify our custom autograd function that defines how forward and
# backward passes are performed.
thr_func = sn.ActFunADP.apply
logger.info(f"{thr_func=}")

# Specify our loss function
criterion_f = nn.CrossEntropyLoss()  # nn.NLLLoss()
logger.info(f"{criterion_f=}")

# Retrieve the Google Speech Commands Dataset
gsc_dataset = torchaudio.datasets.SPEECHCOMMANDS(
    DATAROOT, url=GSC_URL, folder_in_archive="SpeechCommands", download=True
)

# Create random noise files for training and validation
silence_folder = GSC / "_silence_"
if not silence_folder.exists():
    # Create the folder where we'"ll write our white noise files
    silence_folder.mkdir(parents=True, exist_ok=True)

    # Compose a list of the GSC background noise files.

    # Four of the six files envoked a warning when read.
```

```
    # This is why we'll not choose from among these six
    # but use one file generate our noise files.
    # background_noise_files = [*gen_find("*.wav", GSC / "_background_noise_")]

    # This is the one wav file we'll use to generate our own white noise files.
    background_noise_file = GSC / "_background_noise_" / "white_noise.wav"

    # 260 validation / 2300 training
    generate_noise_files(
        nb_files=2560,
        noise_file=background_noise_file,
        output_folder=silence_folder,
        file_prefix="rd_silence_",
        sr=SR,
    )

    # Compose a list of the new noise files
    # and write the first 260 names to the
    # silence_validation_list.txt file
    silence_files = [*gen_find("*.wav", silence_folder)]
    with open(
        GSC / "silence_validation_list.txt", mode="w", encoding="utf-8"
    ) as f:
        for filename in silence_files[:260]:
            f.write(f"{filename}\n")

    logger.info("{Successfully created silence random noise files.")

# Create Class Label Dictionary
```

```
# The dictionary's keys:value pairs are category names gleaned from
# the GSC directory structure and integers, i.e. [0-10, 11].  The
# first eleven keys or categories, whether chozen ordinally or drawn
# randomly, recieve as values the first ten integers. The next
# key:value pair is {'unknown':11}. The remaining key or categories
# all recieve the value 11.  The values [0-10] represent testing
# categories.

# Beginning at GSC find directories without a leading underscore.
class_labels = list(
    {Path(dir).parts[-1] for dir in gen_dfind(r"^(?!_).*", GSC)}
)
logger.info(f"Class Labels:\n{pp.pformat(class_labels)}")

# Compose the class dictionary by choosing
# the first eleven categories sequentially.
# CLASS_DICT = dict(
#     {j: i for i, j in enumerate(class_labels[:11])},
#     **{"unknown": 11},
#     **{j: 11 for _, j in enumerate(class_labels[11:])},
# )

# Compose the class dictionary by choosing
# the first eleven categories randomly.
# fmt: off
CLASS_DICT = dict(
    {j: i for i, j in enumerate([class_labels.pop(random.randrange(len(class_labels))) for _ in range(11)])},
# noqa: E501 pylint: disable=C0301
    **{"unknown": 11},
```

```
    **{i: 11 for i in class_labels})
# fmt: on
logger.info(f"CLASS DICT:\n{pp.pformat(CLASS_DICT)}")

# Two CLASS_DICT helper functions
def label_to_index(word: str) -> int:
    """Return the position of the word in labels"""
    return CLASS_DICT[word]

def index_to_label(index: int) -> str:
    """Return the word corresponding to the index in labels
    This is the inverse of label_to_index"""

    return list(CLASS_DICT.keys())[list(CLASS_DICT.values()).index(index)]

# Here begins the Spiking Recurrent Neural Network specific code.

def collate_fn(data):
    """
    This custom collate function is called with a list of data samples.
    It collates the input samples into a batch for yielding from the
    data loader iterator.

    Dividing the batch by its standard deviation yields a distribution with
    standard deviation equal to 1 (and a variance equal to 1^2=1).
    """
    x_batch = np.array([d[0] for d in data])  # pylint: disable=C0103
    std = x_batch.std(axis=(0, 2), keepdims=True)
    x_batch = torch.tensor(x_batch / std)  # pylint: disable=E1101
```

```
        y_batch = torch.tensor([d[1] for d in data])  # pylint: disable=C0103,E1101
        # y_batch = [d[1] for d in data]  # pylint: disable=C0103,E1101

        return x_batch, y_batch

    # Definition of the overall RNN network
    class RecurrentSpikingNetwork(nn.Module):  # pylint: disable=R0903

        """
        Class docstring
        """

        def __init__(
            self,
        ):
            """
            Constructor docstring
            """
            super().__init__()
            N = 256  # pylint: disable=C0103
            # IS_BIAS=False

            # Here is what the network looks like
            self.dense_1 = sd.SpikeDENSE(
                40 * 3,
                N,
                tau_adp_inital_std=50,
                tau_adp_inital=200,
                tau_m=20,
```

```
        tau_m_inital_std=5,
        device=device,
        bias=IS_BIAS,
    )
    self.rnn_1 = sr.SpikeRNN(
        N,
        N,
        tau_adp_inital_std=50,
        tau_adp_inital=200,
        tau_m=20,
        tau_m_inital_std=5,
        device=device,
        bias=IS_BIAS,
    )
    self.dense_2 = sd.ReadoutIntegrator(
        N, 12, tau_m=10, tau_m_inital_std=1, device=device, bias=IS_BIAS
    )

    # self.dense_2 = sr.spike_rnn(
    #     N,
    #     12,
    #     tauM=10,
    #     tauM_inital_std=1,
    #     device=device,
    #     bias=IS_BIAS, #10
    # )

    # Please comment this code
    self.thr = nn.Parameter(torch.Tensor(1))
```

```
        nn.init.constant_(self.thr, 5e-2)

        # Initialize the network layers
        torch.nn.init.kaiming_normal_(self.rnn_1.recurrent.weight)

        torch.nn.init.xavier_normal_(self.dense_1.dense.weight)
        torch.nn.init.xavier_normal_(self.dense_2.dense.weight)

        if IS_BIAS:
            torch.nn.init.constant_(self.rnn_1.recurrent.bias, 0)
            torch.nn.init.constant_(self.dense_1.dense.bias, 0)
            torch.nn.init.constant_(self.dense_2.dense.bias, 0)

    def forward(self, inputs):  # pylint: disable=R0914
        """
        Forward member function docstring
        """
        # What is this that returns 4 values?
        # What is b?
        # Stereo channels?
        (
            b,  # pylint: disable=C0103
            channel,
            seq_length,
            inputs_dim,
        ) = inputs.shape
        self.dense_1.set_neuron_state(b)
        self.dense_2.set_neuron_state(b)
        self.rnn_1.set_neuron_state(b)
```

```
fr_1 = []
fr_2 = []
# fr_3 = []
output = 0

# inputs_s = inputs
# Why multiply by 1?
inputs_s = (
    thr_func(inputs - self.thr) * 1.0
    - thr_func(-self.thr - inputs) * 1.0
)

# For every timestep update the membrane potential
for i in range(seq_length):
    inputs_x = inputs_s[:, :, i, :].reshape(b, channel * inputs_dim)
    (
        mem_layer1,  # mem_layer1 unused! pylint: disable=W0612,C0301
        spike_layer1,
    ) = self.dense_1.forward(inputs_x)
    (
        mem_layer2,  # mem_layer2 unused! pylint: disable=W0612,C0301
        spike_layer2,
    ) = self.rnn_1.forward(spike_layer1)
    # mem_layer3,spike_layer3 = self.dense_2.forward(spike_layer2)
    mem_layer3 = self.dense_2.forward(spike_layer2)

    # #tracking #spikes (firing rate)
    output += mem_layer3
    fr_1.append(spike_layer1.detach().cpu().numpy().mean())
```

```
            fr_2.append(spike_layer2.detach().cpu().numpy().mean())
            # fr_3.append(spike_layer3.detach().cpu().numpy().mean())

        output = F.log_softmax(output / seq_length, dim=1)
        return output, [
            np.mean(np.abs(inputs_s.detach().cpu().numpy())),
            np.mean(fr_1),
            np.mean(fr_2),
        ]

def test(data_loader, is_show=0):
    """
    test function docstring
    """

    test_acc = 0.0
    sum_sample = 0.0
    fr_ = []
    for _, (images, labels) in enumerate(data_loader):
        images = images.view(-1, 3, 101, 40).to(device)

        labels = labels.view((-1)).long().to(device)
        predictions, fr = model(images)  # pylint: disable=C0103
        fr_.append(fr)
        values, predicted = torch.max(  # pylint: disable=W0612,E1101
            predictions.data, 1
        )
        labels = labels.cpu()
        predicted = predicted.cpu().t()
```

```
            test_acc += (predicted == labels).sum()
            sum_sample += predicted.numel()
        mean_fr = np.mean(fr_, axis=0)
        if is_show:
            logger.info(f"Mean FR: {mean_fr}")

        return test_acc.data.cpu().numpy() / sum_sample, mean_fr

def train(
        data_loader, epochs, criterion, optimizer, scheduler=None
):  # pylint: disable=R0914
    """
    train function docstring
    """
    acc_list = []
    best_acc = 0

    path = "../model/"  # .pth'
    for epoch in range(epochs):
        logger.info(f"{epoch=}")
        train_acc = 0
        sum_sample = 0
        train_loss_sum = 0
        for _, (images, labels) in enumerate(data_loader):
            # if i == 0:
            images = images.view(-1, 3, 101, 40).to(device)

            labels = labels.view((-1)).long().to(device)
            optimizer.zero_grad()
```

```
            predictions, _ = model(images)
            values, predicted = torch.max(  # pylint: disable=W0612,E1101
                predictions.data, 1
            )

            logger.debug(f"predictions:\n{pp.pformat(predictions)}]")
            logger.debug(f"labels:\n{pp.pformat(labels)}]")

            train_loss = criterion(predictions, labels)

            logger.debug(f"{predictions=}\n{predicted=}")

            train_loss.backward()
            train_loss_sum += train_loss.item()
            optimizer.step()

            labels = labels.cpu()
            predicted = predicted.cpu().t()

            train_acc += (predicted == labels).sum()
            sum_sample += predicted.numel()

        if scheduler:
            scheduler.step()
        train_acc = train_acc.data.cpu().numpy() / sum_sample
        valid_acc, _ = test(gsc_testing_dataloader, 1)  # what?!
        train_loss_sum += train_loss

        acc_list.append(train_acc)
```

```
        logger.info(f"{optimizer.param_groups[0]['lr']=}")

        if valid_acc > best_acc and train_acc > 0.890:
            best_acc = valid_acc
            torch.save(model, path + str(best_acc)[:7] + "-srnn-v3.pth")
        logger.info(f"{model.thr=}")

        training_loss = train_loss_sum / len(data_loader)
        logger.info(
            f"{epoch=:}, {training_loss=}, {train_acc=:.4f}, {valid_acc=:.4f}"
        )

    return acc_list

# Reading and preprocessing the data

# The training dataset.
# Note that the transformations specified here are applied in
# the __getitem__ dunder method of the custom the GSCSSubsetSC class.

gsc_training_dataset = GSCSSubsetSC(
    root=DATAROOT,
    url=GSC_URL,
    folder_in_archive="SpeechCommands",
    download=True,
    subset="training",
    transform=TRANSFORMS,
    class_dict=CLASS_DICT,
)
```

```
    logger.info(
        f"The training data consists of {len(gsc_training_dataset)} samples."
    )

    waveform, idx = gsc_training_dataset[0]
    logger.info(f"Shape of gsc_training_set waveform: {waveform.shape}")
    logger.info(f"Waveform label: {index_to_label(idx)}")
    # labels = sorted(list(set(index_to_label(datapoint[1]) for datapoint in gsc_training_dataset)))
    # noqa: E501 pylint: disable=C0301
    # logger.info(f"training labels:\n{pp.pformat(labels)}]")

    # The training dataloader
    gsc_training_dataloader = torch.utils.data.DataLoader(
        gsc_training_dataset,
        batch_size=BATCH_SIZE,
        shuffle=False,
        drop_last=False,
        collate_fn=collate_fn,
        num_workers=NUMBER_OF_WORKERS,
        pin_memory=PIN_MEMORY,
    )
    gsc_features, gsc_labels = next(iter(gsc_training_dataloader))
    logger.info(f"Training Feature batch shape: {gsc_features.size()}")
    logger.info(f"Training Labels batch shape: {gsc_labels.size()}")
    logger.info(f"Training labels, i.e. indices:\n{pp.pformat(gsc_labels)}]")
    # logger.info(f"Training labels[{len(gsc_labels)}]:\n{pp.pformat(gsc_labels)}")

    # The testing dataset
    gsc_testing_dataset = GSCSSubsetSC(
```

```
    root=DATAROOT,
    url=GSC_URL,
    folder_in_archive="SpeechCommands",
    download=True,
    subset="testing",
    transform=TRANSFORMS,
    class_dict=CLASS_DICT,
)
logger.info(
    f"The testing data consists of {len(gsc_testing_dataset)} samples."
)

# The testing dataloader
gsc_testing_dataloader = torch.utils.data.DataLoader(
    gsc_testing_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    drop_last=False,
    collate_fn=collate_fn,
    num_workers=NUMBER_OF_WORKERS,
    pin_memory=PIN_MEMORY,
)

# Instantiate the model
model = RecurrentSpikingNetwork()
model.to(device)

# Test before training
test_acc_before_training = test(gsc_testing_dataloader)
```

```
    logger.info(f"{test_acc_before_training=}")

    # Prepare for training
    base_params = (
        [
            model.dense_1.dense.weight,
            model.dense_1.dense.bias,
            model.rnn_1.dense.weight,
            model.rnn_1.dense.bias,
            model.rnn_1.recurrent.weight,
            model.rnn_1.recurrent.bias,
            # model.dense_2.recurrent.weight,
            # model.dense_2.recurrent.bias,
            model.dense_2.dense.weight,
            model.dense_2.dense.bias,
        ]
        if IS_BIAS
        else [
            model.dense_1.dense.weight,
            model.rnn_1.dense.weight,
            model.rnn_1.recurrent.weight,
            model.dense_2.dense.weight,
        ]
    )

    optimizer_f = torch.optim.Adam(
        [
            {"params": base_params, "lr": LEARNING_RATE},
            {"params": model.thr, "lr": LEARNING_RATE * 0.01},
```

```
        {"params": model.dense_1.tau_m, "lr": LEARNING_RATE * 2},
        {"params": model.dense_2.tau_m, "lr": LEARNING_RATE * 2},
        {"params": model.rnn_1.tau_m, "lr": LEARNING_RATE * 2},
        {"params": model.dense_1.tau_adp, "lr": LEARNING_RATE * 2.0},
        #   {'}params': model.dense_2.tau_adp, 'lr': LEARNING_RATE * 10},
        {"params": model.rnn_1.tau_adp, "lr": LEARNING_RATE * 2.0},
    ],
    lr=LEARNING_RATE,
)

# scheduler_f = StepLR(optimizer_f, step_size=20, gamma=.5) # 20
scheduler_f = StepLR(optimizer_f, step_size=10, gamma=0.1)  # 20
# scheduler_f = LambdaLR(optimizer_f,lr_lambda=lambda epoch: 1-epoch/70)
# scheduler_f = ExponentialLR(optimizer_f, gamma=0.85)

# Training
train_acc_training_complete = train(
    gsc_training_dataloader, EPOCHS, criterion_f, optimizer_f, scheduler_f
)
logger.info(f"TRAINING COMPLETE: {train_acc_training_complete=}")

# Testing
test_acc_after_training = test(gsc_testing_dataloader)
logger.info(f"TESTING COMPLETE: {test_acc_after_training}")

# finis
# Local Variables:
# compile-command: "pyflakes srnn.py; pylint-3 -f parseable srnn.py" # NOQA, pylint: disable=C0301
# End:
```

2 srnn_fin.py

```
#! /usr/bin/env python

# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0

"""
This is a functional recurrent spiking neural network

"""

import os
import pprint
import sys

import numpy as np
import torch
import torch.nn.functional as F
import torchvision
from loguru import logger
from torch import nn
from torch.optim.lr_scheduler import StepLR
from torch.utils.data import DataLoader

import efficient_spiking_networks.srnn_layers.spike_dense as sd
import efficient_spiking_networks.srnn_layers.spike_neuron as sn
```

```python
import efficient_spiking_networks.srnn_layers.spike_rnn as sr
from GSC.data import Pad  # pylint: disable=C0301
from GSC.data import MelSpectrogram, Normalize, Rescale, SpeechCommandsDataset
from GSC.utils import generate_random_silence_files

# import snoop
# import deeplake
# from tqdm import tqdm_notebo

# Setup pretty printing
pp = pprint.PrettyPrinter(indent=4, width=41, compact=True)

# Setup logger level
logger.remove()
logger.add(sys.stderr, level="INFO")

sys.path.append("..")

# device = torch.device("cpu")
device = torch.device(  # pylint: disable=E1101
    "cuda:0" if torch.cuda.is_available() else "cpu"
)
logger.info(f"{device=}")

# Setup number of workers dependent upon where the code is run
NUMBER_OF_WORKERS = 4 if device.type == "cpu" else 8
logger.info(f"The Dataloader will spawn {NUMBER_OF_WORKERS} worker processes.")

# Data Directories
```

```
TRAIN_DATA_ROOT = "./DATA/train"
TEST_DATA_ROOT = "./DATA/test"

# Specify the learning rate
LEARNING_RATE = 3e-3  # 1.2e-2

EPOCHS = 1

BATCH_SIZE = 32
SIZE = 16000
SR = 16000  # Sampling Rate 16Hz ?

DELTA_ORDER = 2
FMAX = 4000
FMIN = 20
HOP_LENGTH = int(10e-3 * SR)
N_FFT = int(30e-3 * SR)
N_MELS = 40
STACK = True

# Turn wav files into Melspectrograms
melspec = MelSpectrogram(
    SR, N_FFT, HOP_LENGTH, N_MELS, FMIN, FMAX, DELTA_ORDER, stack=STACK
)

pad = Pad(SIZE)
rescale = Rescale()
normalize = Normalize()
transform = torchvision.transforms.Compose([pad, melspec, rescale])
```

```
# Define the overall RNN network
class RecurrentSpikingNetwork(nn.Module):  # pylint: disable=R0903

    """
    Class docstring
    """

    def __init__(
        self,
    ):
        """
        Constructor docstring
        """
        super().__init__()
        N = 256  # pylint: disable=C0103
        # IS_BIAS=False

        # Here is what the network looks like
        self.dense_1 = sd.SpikeDENSE(
            40 * 3,
            N,
            tau_adp_inital_std=50,
            tau_adp_inital=200,
            tau_m=20,
            tau_m_inital_std=5,
            device=device,
            bias=IS_BIAS,
        )
        self.rnn_1 = sr.SpikeRNN(
```

```
            N,
            N,
            tau_adp_inital_std=50,
            tau_adp_inital=200,
            tau_m=20,
            tau_m_inital_std=5,
            device=device,
            bias=IS_BIAS,
        )
        self.dense_2 = sd.ReadoutIntegrator(
            N, 12, tau_m=10, tau_m_inital_std=1, device=device, bias=IS_BIAS
        )

        # self.dense_2 = sr.spike_rnn(
        #     N,
        #     12,
        #     tauM=10,
        #     tauM_inital_std=1,
        #     device=device,
        #     bias=IS_BIAS, #10
        # )

        # Please comment this code
        self.thr = nn.Parameter(torch.Tensor(1))
        nn.init.constant_(self.thr, 5e-2)

        # Initialize the network layers
        torch.nn.init.kaiming_normal_(self.rnn_1.recurrent.weight)
```

```
        torch.nn.init.xavier_normal_(self.dense_1.dense.weight)
        torch.nn.init.xavier_normal_(self.dense_2.dense.weight)

        if IS_BIAS:
            torch.nn.init.constant_(self.rnn_1.recurrent.bias, 0)
            torch.nn.init.constant_(self.dense_1.dense.bias, 0)
            torch.nn.init.constant_(self.dense_2.dense.bias, 0)

    def forward(self, inputs):  # pylint: disable=R0914
        """
        Forward member function docstring
        """
        # What is this that returns 4 values?
        # What is b?
        # Stereo channels?
        (
            b,  # pylint: disable=C0103
            channel,
            seq_length,
            inputs_dim,
        ) = inputs.shape
        self.dense_1.set_neuron_state(b)
        self.dense_2.set_neuron_state(b)
        self.rnn_1.set_neuron_state(b)

        fr_1 = []
        fr_2 = []
        # fr_3 = []
        output = 0
```

```
        # inputs_s = inputs
        # Why multiply by 1?
        inputs_s = (
            thr_func(inputs - self.thr) * 1.0
            - thr_func(-self.thr - inputs) * 1.0
        )

        # For every timestep update the membrane potential
        for i in range(seq_length):
            inputs_x = inputs_s[:, :, i, :].reshape(b, channel * inputs_dim)
            (
                mem_layer1,  # mem_layer1 unused! pylint: disable=W0612,C0301
                spike_layer1,
            ) = self.dense_1.forward(inputs_x)
            (
                mem_layer2,  # mem_layer2 unused! pylint: disable=W0612,C0301
                spike_layer2,
            ) = self.rnn_1.forward(spike_layer1)
            # mem_layer3,spike_layer3 = self.dense_2.forward(spike_layer2)
            mem_layer3 = self.dense_2.forward(spike_layer2)

            # #tracking #spikes (firing rate)
            output += mem_layer3
            fr_1.append(spike_layer1.detach().cpu().numpy().mean())
            fr_2.append(spike_layer2.detach().cpu().numpy().mean())
            # fr_3.append(spike_layer3.detach().cpu().numpy().mean())

        output = F.log_softmax(output / seq_length, dim=1)
        return output, [
```

```
            np.mean(np.abs(inputs_s.detach().cpu().numpy())),
            np.mean(fr_1),
            np.mean(fr_2),
        ]

# Please comment this code
def collate_fn(data):
    """
    Collate function docscting
    """

    x_batch = np.array([d[0] for d in data])  # pylint: disable=C0103
    std = x_batch.std(axis=(0, 2), keepdims=True)
    x_batch = torch.tensor(x_batch / std)  # pylint: disable=E1101
    y_batch = torch.tensor([d[1] for d in data])  # pylint: disable=C0103,E1101

    return x_batch, y_batch

def test(data_loader, is_show=0):
    """
    test function docstring
    """

    test_acc = 0.0
    sum_sample = 0.0
    fr_ = []
    for _, (images, labels) in enumerate(data_loader):
        images = images.view(-1, 3, 101, 40).to(device)
```

```
            labels = labels.view((-1)).long().to(device)
            predictions, fr = model(images)  # pylint: disable=C0103
            fr_.append(fr)
            values, predicted = torch.max(  # pylint: disable=W0612,E1101
                predictions.data, 1
            )
            labels = labels.cpu()
            predicted = predicted.cpu().t()

            test_acc += (predicted == labels).sum()
            sum_sample += predicted.numel()
        mean_fr = np.mean(fr_, axis=0)
        if is_show:
            logger.info(f"Mean FR: {mean_fr}")

        return test_acc.data.cpu().numpy() / sum_sample, mean_fr

    def train(
        epochs, criterion, optimizer, scheduler=None
    ):  # pylint: disable=R0914
        """
        train function docstring
        """
        acc_list = []
        best_acc = 0

        path = "../model/"  # .pth'
        for epoch in range(epochs):
            train_acc = 0
```

```
    sum_sample = 0
    train_loss_sum = 0
    for _, (images, labels) in enumerate(train_dataloader):
        # if i ==0:
        images = images.view(-1, 3, 101, 40).to(device)

        labels = labels.view((-1)).long().to(device)
        optimizer.zero_grad()

        predictions, _ = model(images)
        values, predicted = torch.max(  # pylint: disable=W0612,E1101
            predictions.data, 1
        )

        logger.debug(f"predictions:\n{pp.pformat(predictions)}]")
        logger.debug(f"labels:\n{pp.pformat(labels)}]")
        train_loss = criterion(predictions, labels)

        logger.debug(f"{predictions=}\n{predicted=}")

        train_loss.backward()
        train_loss_sum += train_loss.item()
        optimizer.step()

        labels = labels.cpu()
        predicted = predicted.cpu().t()

        train_acc += (predicted == labels).sum()
        sum_sample += predicted.numel()
```

```
        if scheduler:
            scheduler.step()
        train_acc = train_acc.data.cpu().numpy() / sum_sample
        valid_acc, _ = test(test_dataloader, 1)
        train_loss_sum += train_loss

        acc_list.append(train_acc)
        logger.info(f"{optimizer.param_groups[0]['lr']=}")

        if valid_acc > best_acc and train_acc > 0.890:
            best_acc = valid_acc
            torch.save(model, path + str(best_acc)[:7] + "-srnn-v3.pth")
        logger.info(f"{model.thr=}")

        training_loss = train_loss_sum / len(train_dataloader)
        logger.info(
            f"{epoch=:}, {training_loss=}, {train_acc=:.4f}, {valid_acc=:.4f}"
        )

    return acc_list

# Definitions complete - let's get going!

# list the directories and folders in TRAIN_DATA_ROOT folder
training_words = os.listdir(TRAIN_DATA_ROOT)

# Isolate the directories in the train_date_root
training_words = [
    x
```

```
    for x in training_words  # pylint: disable=C0103
    if os.path.isdir(os.path.join(TRAIN_DATA_ROOT, x))
]

# Ignore those that begin with an underscore
training_words = [
    x
    for x in training_words  # pylint: disable=C0103
    if os.path.isdir(os.path.join(TRAIN_DATA_ROOT, x))
    if x[0] != "_"
]
logger.info(
    f"traiing words[{len(training_words)}]:\n{pp.pformat(training_words)}]"
)

# list the directories and folders in TEST_DATA_ROOT folder
testing_words = os.listdir(TEST_DATA_ROOT)

# Look for testing_word directories in TRAIN_DATA_ROOT so that we only
# select test data for selected training classes.
testing_words = [
    x
    for x in testing_words  # pylint: disable=C0103
    if os.path.isdir(os.path.join(TRAIN_DATA_ROOT, x))
]

# Ignore those that begin with an underscore
testing_words = [
    x
```

```
        for x in testing_words  # pylint: disable=C0103
        if os.path.isdir(os.path.join(TRAIN_DATA_ROOT, x))
        if x[0] != "_"
    ]
    logger.info(
        f"testing words[{len(testing_words)}]:\n{pp.pformat(testing_words)}]"
    )

    # Create a dictionary whose keys are
    # testing_words(in the TRAIN_DATA_ROOT)
    # and whose values are the words' ordianal
    # position in the original list.

    label_dct = {
        k: i for i, k in enumerate(testing_words + ["_silence_", "_unknown_"])
    }

    # Look for training directories in testing directories.
    for w in training_words:
        label = label_dct.get(w)
        if label is None:
            label_dct[w] = label_dct["_unknown_"]

    # Dictionary of testing words plus training words not in testing words.
    logger.info(pp.pformat(f"{len(label_dct)=}, {label_dct=}"))

    noise_path = os.path.join(TRAIN_DATA_ROOT, "_background_noise_")
    noise_files = []
    for f in os.listdir(noise_path):
```

```
    if f.endswith(".wav"):
        full_name = os.path.join(noise_path, f)
        noise_files.append(full_name)

logger.info(f"noise_files[{len(noise_files)}]:\n{pp.pformat(noise_files)}]")

# generate silence training and validation data

silence_folder = os.path.join(TRAIN_DATA_ROOT, "_silence_")
if not os.path.exists(silence_folder):
    os.makedirs(silence_folder)
    # 260 validation / 2300 training
    generate_random_silence_files(
        2560, noise_files, SIZE, os.path.join(silence_folder, "rd_silence")
    )

    # save 260 files for validation
    silence_files = list(os.listdir(silence_folder))
    silence_lines = [
        "_silence_/" + fname + "\n" for fname in silence_files[:260]
    ]
    silence_filename = os.path.join(
        TRAIN_DATA_ROOT, "silence_validation_list.txt"
    )
    with open(silence_filename, "a", encoding="utf-8") as fp:
        fp.writelines(silence_lines)

# Collect the training, testing and validation data
```

```
    train_dataset = SpeechCommandsDataset(
        TRAIN_DATA_ROOT,
        label_dct,
        transform=transform,
        mode="train",
        max_nb_per_class=None,
    )

    item, label = train_dataset[0]
    logger.info(f"Shape of train item: {item.shape}")
    logger.info(f"Label of train item: {label}")

    train_sampler = torch.utils.data.WeightedRandomSampler(
        train_dataset.weights, len(train_dataset.weights)
    )

    train_dataloader = DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        num_workers=NUMBER_OF_WORKERS,
        sampler=train_sampler,
        collate_fn=collate_fn,
    )

    train_features, train_labels = next(iter(train_dataloader))
    logger.info(f"Train Feature batch shape: {train_features.size()}")
    logger.info(f"Train Labels batch shape: {train_labels.size()}")
    logger.info(f"Train labels:\n{pp.pformat(train_labels)}]")
```

```
valid_dataset = SpeechCommandsDataset(
    TRAIN_DATA_ROOT,
    label_dct,
    transform=transform,
    mode="valid",
    max_nb_per_class=None,
)

valid_dataloader = DataLoader(
    valid_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=NUMBER_OF_WORKERS,
    collate_fn=collate_fn,
)

test_dataset = SpeechCommandsDataset(
    TEST_DATA_ROOT, label_dct, transform=transform, mode="test"
)

item, label = test_dataset[0]
logger.info(f"Shape of test item: {item.shape}")
logger.info(f"Label of test item: {label}")

test_dataloader = DataLoader(
    test_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=NUMBER_OF_WORKERS,
```

```
    collate_fn=collate_fn,
)

test_features, test_labels = next(iter(test_dataloader))
logger.info(f"Test Feature batch shape: {test_features.size()}")
logger.info(f"Test Labels batch shape: {test_labels.size()}")
logger.info(f"Test labels:\n{pp.pformat(test_labels)}]")

# Specify the function that will apply the forward and backward passes
thr_func = sn.ActFunADP.apply
IS_BIAS = True

# Instantiate the model
model = RecurrentSpikingNetwork()
criterion_f = nn.CrossEntropyLoss()  # nn.NLLLoss()

model.to(device)

test_acc_before_training = test(test_dataloader)
logger.info(f"{test_acc_before_training=}")

if IS_BIAS:
    base_params = [
        model.dense_1.dense.weight,
        model.dense_1.dense.bias,
        model.rnn_1.dense.weight,
        model.rnn_1.dense.bias,
        model.rnn_1.recurrent.weight,
        model.rnn_1.recurrent.bias,
```

```
            # model.dense_2.recurrent.weight,
            # model.dense_2.recurrent.bias,
            model.dense_2.dense.weight,
            model.dense_2.dense.bias,
        ]
else:
        base_params = [
            model.dense_1.dense.weight,
            model.rnn_1.dense.weight,
            model.rnn_1.recurrent.weight,
            model.dense_2.dense.weight,
        ]

optimizer_f = torch.optim.Adam(
        [
            {"params": base_params, "lr": LEARNING_RATE},
            {"params": model.thr, "lr": LEARNING_RATE * 0.01},
            {"params": model.dense_1.tau_m, "lr": LEARNING_RATE * 2},
            {"params": model.dense_2.tau_m, "lr": LEARNING_RATE * 2},
            {"params": model.rnn_1.tau_m, "lr": LEARNING_RATE * 2},
            {"params": model.dense_1.tau_adp, "lr": LEARNING_RATE * 2.0},
            #   {'}params': model.dense_2.tau_adp, 'lr': LEARNING_RATE * 10},
            {"params": model.rnn_1.tau_adp, "lr": LEARNING_RATE * 2.0},
        ],
        lr=LEARNING_RATE,
)

# scheduler_f = StepLR(optimizer_f, step_size=20, gamma=.5) # 20
scheduler_f = StepLR(optimizer_f, step_size=10, gamma=0.1)  # 20
```

```
    # scheduler_f = LambdaLR(optimizer_f,lr_lambda=lambda epoch: 1-epoch/70)
    # scheduler_f = ExponentialLR(optimizer_f, gamma=0.85)

    train_acc_training_complete = train(
        EPOCHS, criterion_f, optimizer_f, scheduler_f
    )
    logger.info(f"{train_acc_training_complete=}")

    logger.info("TRAINING COMPLETE")

    test_acc_after_training = test(test_dataloader)
    logger.info(f"{test_acc_after_training}")

    logger.info("TESTING COMPLETE")

    # finis

    # Local Variables:
    # compile-command: "pyflakes srnn_fin.py; pylint-3 -d E0401 -f parseable srnn_fin.py" # NOQA, pylint:
    disable=C0301
    # End:
```

3 data.py

```
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0

""" Module Docstring """

import os
from pathlib import Path
from typing import Optional, Union

import librosa
import numpy as np
import scipy.io.wavfile as wav
import torch
from torch.utils.data import Dataset
from torchaudio.datasets import SPEECHCOMMANDS
from torchaudio.datasets.utils import _load_waveform
from utils import txt2list

class GSCSSubsetSC(SPEECHCOMMANDS):
    """Class Docstring"""

    def __init__(  # pylint: disable=R0913
        self,
        root: Union[str, Path],
        url: str = "speech_commands_v0.02",
```

```
        folder_in_archive: str = "SpeechCommands",
        download: bool = True,
        subset: Optional[str] = None,
        transform: Optional[str] = None,
        class_dict: dict = None,
    ) -> None:
        """Function Docstring"""
        super().__init__(
            root, url=url, folder_in_archive="SpeechCommands", download=True
        )

        # two instance variables specific to this subclass
        self.transform = transform
        self.class_dict = class_dict

        def load_list(filename):
            """Function Docstring"""
            filepath = os.path.join(self._path, filename)
            with open(filepath, mode="r", encoding="utf-8") as fileobj:
                return [
                    os.path.normpath(os.path.join(self._path, line.strip()))
                    for line in fileobj
                ]

        if subset == "validation":
            self._walker = load_list("validation_list.txt") + load_list(
                "silence_validation_list.txt"
            )
        elif subset == "testing":
```

```
            self._walker = load_list("testing_list.txt")
        elif subset == "training":
            excludes = (
                load_list("testing_list.txt")
                + load_list("validation_list.txt")
                + load_list("silence_validation_list.txt")
            )
            excludes = set(excludes)
            self._walker = [w for w in self._walker if w not in excludes]  # noqa: E501 pylint: disable=C0103

    def __getitem__(self, n):
        """Function Docstring"""

        metadata = self.get_metadata(n)
        waveform = _load_waveform(self._archive, metadata[0], metadata[1])
        maximum = torch.max(torch.abs(waveform))  # pylint: disable=E1101

        if maximum > 0:
            waveform /= maximum
        if self.transform is not None:
            waveform = self.transform(waveform.squeeze())
        return (waveform, self.class_dict[metadata[2]],)

class SpeechCommandsDataset(Dataset):
    """Class Docstring"""

    def __init__(  # pylint: disable=R0912,R0913,R0914
        self, data_root, label_dct, mode,
        transform=None, max_nb_per_class=None
```

```
    ):
        """Function Docstring"""

        assert mode in [
            "train",
            "valid",
            "test",
        ], 'mode should be "train", "valid" or "test"'

        self.filenames = []
        self.labels = []
        self.mode = mode
        self.transform = transform

        if (
            self.mode == "train"  # pylint: disable=R1714
            or self.mode == "valid"
        ):
            # Create lists of 'wav' files.
            testing_list = txt2list(
                os.path.join(data_root, "testing_list.txt")
            )
            validation_list = txt2list(
                os.path.join(data_root, "validation_list.txt")
            )
            # silence_validation_list.txt not in gsc dataset
            validation_list += txt2list(
                os.path.join(data_root, "silence_validation_list.txt")
            )
```

```
else:
    testing_list = []
    validation_list = []

for root, dirs, files in os.walk(data_root):  # pylint: disable=W0612
    if "_background_noise_" in root:
        continue
    for filename in files:
        if not filename.endswith(".wav"):
            # Ignore files whose suffix is not 'wav'.
            continue

        # Extract the cwd without a path.
        command = root.split("/")[-1]

        label = label_dct.get(command)
        if label is None:
            print(f"ignored command: {command}")
            break  # Out of here!
        partial_path = "/".join([command, filename])

        # These are Boolean values!
        testing_file = partial_path in testing_list
        validation_file = partial_path in validation_list
        training_file = not testing_file and not validation_file

        if (
            (self.mode == "test")
            or (self.mode == "train" and training_file)
```

```
                    or (self.mode == "valid" and validation_file)
                ):
                    full_name = os.path.join(root, filename)
                    self.filenames.append(full_name)
                    self.labels.append(label)

        if max_nb_per_class is not None:
            selected_idx = []
            for label in np.unique(self.labels):
                label_idx = [
                    i for i, x in enumerate(self.labels) if x == label   # noqa: E501 pylint: disable=C0103
                ]
                if len(label_idx) < max_nb_per_class:
                    selected_idx += label_idx
                else:
                    selected_idx += list(
                        np.random.choice(label_idx, max_nb_per_class)
                    )

            self.filenames = [self.filenames[idx] for idx in selected_idx]
            self.labels = [self.labels[idx] for idx in selected_idx]

        if self.mode == "train":
            label_weights = 1.0/np.unique(self.labels, return_counts=True)[1]
            label_weights /= np.sum(label_weights)
            self.weights = torch.DoubleTensor(  # pylint: disable=E1101
                [label_weights[label] for label in self.labels]
            )
```

```
    def __len__(self):
        """Function Docstring"""
        return len(self.labels)

    def __getitem__(self, idx):
        """Function Docstring"""
        filename = self.filenames[idx]
        item = wav.read(filename)[1].astype(float)
        m = np.max(np.abs(item))  # pylint: disable=C0103
        if m > 0:
            item /= m
        if self.transform is not None:
            item = self.transform(item)

        label = self.labels[idx]

        return item, label

class Pad:  # pylint: disable=R0903
    """Class Docstring"""

    def __init__(self, size):
        """Function Docstring"""
        self.size = size

    def __call__(self, waveform):
        """Function Docstring"""
        wav_size = waveform.shape[0]
        pad_size = (self.size - wav_size) // 2
```

```
            padded_wav = np.pad(
                waveform,
                ((pad_size, self.size - wav_size - pad_size),),
                "constant",
                constant_values=(0, 0),
            )
            return padded_wav

# class RandomNoise:  # pylint: disable=R0903
#     """Class Docstring"""

#     def __init__(self, noise_files, size, coef):
#         """Function Docstring"""
#         self.size = size
#         self.noise_files = noise_files
#         self.coef = coef

#     def __call__(self, waveform):
#         """Function Docstring"""
#         if np.random.random() < 0.8:
#             noise_wav = get_random_noise(self.noise_files, self.size)
#             noise_power = (noise_wav**2).mean()
#             sig_power = (waveform**2).mean()

#             noisy_wav = waveform + self.coef * noise_wav * np.sqrt(
#                 sig_power / noise_power
#             )

#         else:
```

```
#             noisy_wav = waveform

#         return noisy_wav

# class RandomShift:  # pylint: disable=R0903
#     """Class Docstring"""

#     def __init__(self, min_shift, max_shift):
#         """Function Docstring"""
#         self.min_shift = min_shift
#         self.max_shift = max_shift

#     def __call__(self, waveform):
#         """Function Docstring"""
#         shift = np.random.randint(self.min_shift, self.max_shift + 1)
#         shifted_wav = np.roll(waveform, shift)

#         if shift > 0:
#             shifted_wav[:shift] = 0
#         elif shift < 0:
#             shifted_wav[shift:] = 0

#         return shifted_wav

class MelSpectrogram:  # pylint: disable=R0902,R0903
    """Class Docstring"""

    def __init__(  # pylint: disable=R0913
        self,
```

```
            sr,  # pylint: disable=C0103
            n_fft,
            hop_length,
            n_mels,
            fmin,
            fmax,
            delta_order=None,
            stack=True,
        ):
            """Function Docstring"""
            self.sr = sr  # pylint: disable=C0103
            self.n_fft = n_fft
            self.hop_length = hop_length
            self.n_mels = n_mels
            self.fmin = fmin
            self.fmax = fmax
            self.delta_order = delta_order
            self.stack = stack

        def __call__(self, waveform):
            """Function Docstring"""
            spectrogram = librosa.feature.melspectrogram(
                y=waveform,
                sr=self.sr,
                n_fft=self.n_fft,
                hop_length=self.hop_length,
                n_mels=self.n_mels,
                fmax=self.fmax,
                fmin=self.fmin,
```

```
        )

        maximum = np.max(np.abs(spectrogram))
        if maximum > 0:
            feat = np.log1p(spectrogram / maximum)
        else:
            feat = spectrogram

        if self.delta_order is not None and not self.stack:
            feat = librosa.feature.delta(feat, order=self.delta_order)
            return np.expand_dims(feat.T, 0)

        if self.delta_order is not None and self.stack:
            feat_list = [feat.T]
            for k in range(1, self.delta_order + 1):
                feat_list.append(librosa.feature.delta(feat, order=k).T)
            return np.stack(feat_list)

        return np.expand_dims(feat.T, 0)

class Rescale:  # pylint: disable=R0903
    """Class Docstring"""

    def __call__(self, data):
        """Function Docstring"""
        std = np.std(data, axis=1, keepdims=True)
        std[std == 0] = 1

        return data / std
```

```
class Normalize:  # pylint: disable=R0903
    """Class Docstring"""

    def __call__(self, data):
        """Function Docstring"""
        data_ = (data > 0.1) * data
        std = np.std(data_, axis=1, keepdims=True)
        std[std == 0] = 1

        return input / std

# class WhiteNoise:  # pylint: disable=R0903
#     """Class Docstring"""

#     def __init__(self, size, coef_max):
#         """Function Docstring"""
#         self.size = size
#         self.coef_max = coef_max

#     def __call__(self, waveform):
#         """Function Docstring"""
#         noise_wav = np.random.normal(size=self.size)
#         noise_power = (noise_wav**2).mean()
#         sig_power = (waveform**2).mean()

#         coef = np.random.uniform(0.0, self.coef_max)

#         noisy_wav = waveform + coef * noise_wav * np.sqrt(
#             sig_power / noise_power
```

```
#         )

#         return noisy_wav

# Local Variables:
# compile-command: "pyflakes data.py; pylint-3 -d E0401 -f parseable data.py" # NOQA, pylint: disable=C0301
# End:
```

4 optim.py

```
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0

"""PyTorch implementation of Rectified Adam from
https://github.com/LiyuanLucasLiu/RAdam"""

import math

import torch
from torch.optim.optimizer import Optimizer

class RAdam(Optimizer):
    """Class Docstring"""

    def __init__(  # pylint: disable=R0913
        self,
        params,
        lr=1e-3,
        betas=(0.9, 0.999),
        eps=1e-8,
        weight_decay=0,
        degenerated_to_sgd=True,
    ):
        if 0.0 > lr:
            raise ValueError(f"Invalid learning rate: {lr}")
```

```
if 0.0 > eps:
    raise ValueError(f"Invalid epsilon value: {eps}")
if not 0.0 <= betas[0] < 1.0:
    raise ValueError(f"Invalid beta parameter at index 0: {betas[0]}")
if not 0.0 <= betas[1] < 1.0:
    raise ValueError(f"Invalid beta parameter at index 1: {betas[1]}")
self.degenerated_to_sgd = degenerated_to_sgd
if (
    isinstance(params, (list, tuple))
    and len(params) > 0
    and isinstance(params[0], dict)
):
    for param in params:
        if "betas" in param and (
            param["betas"][0] != betas[0]
            or param["betas"][1] != betas[1]
        ):
            param["buffer"] = [[None, None, None] for _ in range(10)]

defaults = {
    "lr": lr,
    "betas": betas,
    "eps": eps,
    "weight_decay": weight_decay,
    "buffer": [[None, None, None] for _ in range(10)],
}

super().__init__(params, defaults)
```

```
    # def __setstate__(self, state):
    #     """Function Docstring"""
    #     super().__setstate__(state)

    def step(self, closure=None):  # pylint: disable=R0912, R0914
        """Function Docstring"""
        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:
            for p in group["params"]:  # pylint: disable=C0103
                if p.grad is None:
                    continue
                grad = p.grad.data.float()
                if grad.is_sparse:
                    raise RuntimeError(
                        "RAdam does not support sparse gradients"
                    )

                p_data_fp32 = p.data.float()

                state = self.state[p]

                if len(state) == 0:
                    state["step"] = 0
                    state[
                        "exp_avg"
                    ] = torch.zeros_like(  # pylint: disable=E1101
```

```
            p_data_fp32
        )
        state[
            "exp_avg_sq"
        ] = torch.zeros_like(  # pylint: disable=E1101
            p_data_fp32
        )
else:
    state["exp_avg"] = state["exp_avg"].type_as(p_data_fp32)
    state["exp_avg_sq"] = state["exp_avg_sq"].type_as(
        p_data_fp32
    )

exp_avg, exp_avg_sq = state["exp_avg"], state["exp_avg_sq"]
beta1, beta2 = group["betas"]

exp_avg_sq.mul_(beta2).addcmul_(1 - beta2, grad, grad)
exp_avg.mul_(beta1).add_(1 - beta1, grad)

state["step"] += 1
buffered = group["buffer"][int(state["step"] % 10)]
if state["step"] == buffered[0]:
    N_sma, step_size = (  # pylint: disable=C0103
        buffered[1],
        buffered[2],
    )
else:
    buffered[0] = state["step"]
    beta2_t = beta2 ** state["step"]
```

```
        N_sma_max = 2 / (1 - beta2) - 1  # pylint: disable=C0103
        N_sma = N_sma_max - 2 * state[  # pylint: disable=C0103
            "step"
        ] * beta2_t / (1 - beta2_t)
        buffered[1] = N_sma

        # more conservative since it's an approximated value
        if N_sma >= 5:
            step_size = math.sqrt(
                (1 - beta2_t)
                * (N_sma - 4)
                / (N_sma_max - 4)
                * (N_sma - 2)
                / N_sma
                * N_sma_max
                / (N_sma_max - 2)
            ) / (1 - beta1 ** state["step"])
        elif self.degenerated_to_sgd:
            step_size = 1.0 / (1 - beta1 ** state["step"])
        else:
            step_size = -1
        buffered[2] = step_size

    # more conservative since it's an approximated value
    if N_sma >= 5:
        if group["weight_decay"] != 0:
            p_data_fp32.add_(
                -group["weight_decay"] * group["lr"], p_data_fp32
            )
```

```
                    denom = exp_avg_sq.sqrt().add_(group["eps"])
                    p_data_fp32.addcdiv_(
                        -step_size * group["lr"], exp_avg, denom
                    )
                    p.data.copy_(p_data_fp32)
                elif step_size > 0:
                    if group["weight_decay"] != 0:
                        p_data_fp32.add_(
                            -group["weight_decay"] * group["lr"], p_data_fp32
                        )
                    p_data_fp32.add_(-step_size * group["lr"], exp_avg)
                    p.data.copy_(p_data_fp32)

        return loss

# finis
# Local Variables:
# compile-command: "pyflakes optim.py; pylint-3 -f parseable optim.py" # NOQA, pylint: disable=C0301
# End:
```

5 utils.pys

```
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0

""" Module Docstring """

import numpy as np
import scipy.io.wavfile as wav

# from matplotlib.gridspec import GridSpec
# import matplotlib.pyplot as plt

def txt2list(filename):
    """This function reads a file containing one filename per line
    and returns a list of lines.

    Could be replaced with:
    for fn in gen_find('"*_list.txt', '/tmp/testdata/'):
        with open(fn) as fp:
            mylist = fp.read().splitlines()

    """
    lines_list = []
    with open(filename, "r") as txt:  # pylint: disable=W1514
        for line in txt:
            lines_list.append(line.rstrip("\n"))
```

```
    return lines_list

# def plot_spk_rec(spk_rec, idx):
#     nb_plt = len(idx)
#     d = int(np.sqrt(nb_plt))
#     gs = GridSpec(d, d)
#     fig = plt.figure(figsize=(30, 20), dpi=150)
#     for i in range(nb_plt):
#         plt.subplot(gs[i])
#         plt.imshow(
#             spk_rec[idx[i]].T,
#             cmap=plt.cm.gray_r,
#             origin="lower",
#             aspect="auto",
#         )
#         if i == 0:
#             plt.xlabel("Time")
#             plt.ylabel("Units")

# def plot_mem_rec(mem, idx):
#     nb_plt = len(idx)
#     d = int(np.sqrt(nb_plt))
#     dim = (d, d)

#     gs = GridSpec(*dim)
#     plt.figure(figsize=(30, 20))
#     dat = mem[idx]

#     for i in range(nb_plt):
```

```
#         if i == 0:
#             a0 = ax = plt.subplot(gs[i])
#         else:
#             ax = plt.subplot(gs[i], sharey=a0)
#         ax.plot(dat[i])

# The following two functions together generated random noise by
# randomly sampling a portion of sound from a randomly chozen
# background noise file. Unvortulately four of the six background
# noise files yield errors when read.

def get_random_noise(noise_files, size):  # pylint: disable=C0116
    noise_idx = np.random.choice(len(noise_files))
    fs, noise_wav = wav.read(noise_files[noise_idx])  # noqa: E501 pylint: disable=W0612,C0103,

    offset = np.random.randint(len(noise_wav) - size)
    noise_wav = noise_wav[offset: offset + size].astype(float)

    return noise_wav

def generate_random_silence_files(  # pylint: disable=C0116
    nb_files, noise_files, size, prefix, sr=16000  # pylint: disable=C0103
):
    for i in range(nb_files):
        silence_wav = get_random_noise(noise_files, size)
        wav.write(prefix + "_" + str(i) + ".wav", sr, silence_wav)

def generate_noise_files(
    nb_files, noise_file, output_folder, file_prefix, sr  # noqa: E501 pylint: disable=C0103
```

```
):
    """Function Docstring"""
    for i in range(nb_files):
        fs, noise_wav = wav.read(  # pylint: disable=C0103,W0612
            noise_file,
        )
        offset = np.random.randint(len(noise_wav) - sr)
        noise_wav = noise_wav[offset: offset + sr].astype(float)
        fn = output_folder / "".join(  # pylint: disable=C0103
            [file_prefix, f"{i}", ".wav"]
        )
        wav.write(fn, sr, noise_wav)

# def split_wav(waveform, frame_size, split_hop_length):
#     splitted_wav = []
#     offset = 0

#     while offset + frame_size < len(waveform):
#         splitted_wav.append(waveform[offset : offset + frame_size])
#         offset += split_hop_length

#     return splitted_wav

# finis
# Local Variables:
# compile-command: "pyflakes utils.py; pylint-3 -f parseable utils.py" # NOQA, pylint: disable=C0301
# End:
```

6 spike_rnn.py

```
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0

"""  module docstring """

__all__ = ["SpikeRNN"]

import torch
from torch import nn
from torch.autograd import Variable

from . import spike_dense as sd
from . import spike_neuron as sn

B_J0: float = sn.B_J0_VALUE

class SpikeRNN(nn.Module):  # pylint: disable=R0902
    """Spike_Rnn class docstring"""

    def __init__(  # pylint: disable=R0913
        self,
        input_dim,
        output_dim,
        tau_m=20,
        tau_adp_inital=100,
```

```
    tau_initializer="normal",
    tau_m_inital_std=5,
    tau_adp_inital_std=5,
    is_adaptive=1,
    device="cpu",
    bias: bool = True,
) -> None:
    """Class constructor member function"""
    super().__init__()
    self.mem: Variable
    self.spike = None
    self.b = None  # pylint: disable=C0103
    self.input_dim = input_dim
    self.output_dim = output_dim
    self.is_adaptive = is_adaptive
    self.device = device

    self.b_j0 = B_J0
    self.dense = nn.Linear(input_dim, output_dim, bias=bias)
    self.recurrent = nn.Linear(output_dim, output_dim, bias=bias)
    self.tau_m = nn.Parameter(torch.Tensor(self.output_dim))
    self.tau_adp = nn.Parameter(torch.Tensor(self.output_dim))

    if tau_initializer == "normal":
        nn.init.normal_(self.tau_m, tau_m, tau_m_inital_std)
        nn.init.normal_(self.tau_adp, tau_adp_inital, tau_adp_inital_std)
    elif tau_initializer == "multi_normal":
        self.tau_m = sd.multi_normal_initilization(
            self.tau_m, tau_m, tau_m_inital_std
```

```
            )
            self.tau_adp = sd.multi_normal_initilization(
                self.tau_adp, tau_adp_inital, tau_adp_inital_std
            )

    def parameters(self):
        """parameters member function docstring"""
        return [
            self.dense.weight,
            self.dense.bias,
            self.recurrent.weight,
            self.recurrent.bias,
            self.tau_m,
            self.tau_adp,
        ]

    def set_neuron_state(self, batch_size):
        """set_neuron_state member function docstring"""

        self.mem = Variable(
            torch.zeros(batch_size, self.output_dim) * self.b_j0
        ).to(self.device)
        self.spike = Variable(torch.zeros(batch_size, self.output_dim)).to(
            self.device
        )
        self.b = Variable(
            torch.ones(batch_size, self.output_dim) * self.b_j0
        ).to(self.device)
```

```
    def forward(self, input_spike):
        """forward member function docstring"""
        d_input = self.dense(input_spike.float()) + self.recurrent(self.spike)
        (
            self.mem,
            self.spike,
            theta,  # pylint: disable=W0612
            self.b,
        ) = sn.mem_update_adp(
            d_input,
            self.mem,
            self.spike,
            self.tau_adp,
            self.b,
            self.tau_m,
            device=self.device,
            isAdapt=self.is_adaptive,
        )

        return self.mem, self.spike

# Local Variables:
# compile-command: "pyflakes spike_rnn.py; pylint-3 -d E0401 -f parseable spike_rnn.py" # NOQA, pylint:
disable=C0301
# End:
```

7 spike_cnn.py

```
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0

"""  module docstring """

__all__ = ["SpikeCov1D", "SpikeCov2D"]

import numpy as np
import torch
from torch import nn

from . import spike_neuron as sn

B_J0 = 1.6

class SpikeCov1D(nn.Module):  # pylint: disable=R0902
    """Spike_Cov1D class docstring"""

    def __init__(  # pylint: disable=R0913,R0914
        self,
        input_size,
        output_dim,
        kernel_size=5,
        strides=1,
        pooling_type=None,
```

```
        pool_size=2,
        pool_strides=2,
        dilation=1,
        tau_m=20,
        tau_adp_inital=100,
        tau_initializer="normal",  # pylint: disable=W0613
        tau_m_inital_std=5,
        tau_adp_inital_std=5,
        is_adaptive=1,
        device="cpu",
    ):
        """Class constructor member function docstring"""
        super().__init__()
        self.mem = None
        self.spike = None
        self.b = None  # pylint: disable=C0103
        # input_size = [c,h]
        self.input_size = input_size
        self.input_dim = input_size[0]
        self.output_dim = output_dim
        self.is_adaptive = is_adaptive
        self.dilation = dilation
        self.device = device

        if pooling_type is not None:
            if pooling_type == "max":
                self.pooling = nn.MaxPool1d(
                    kernel_size=pool_size, stride=pool_strides, padding=1
                )
```

```
            elif pooling_type == "avg":
                self.pooling = nn.AvgPool1d(
                    kernel_size=pool_size, stride=pool_strides, padding=1
                )
        else:
            self.pooling = None

        self.conv = nn.Conv1d(
            self.input_dim,
            self.output_dim,
            kernel_size=kernel_size,
            stride=strides,
            padding=(
                np.ceil(((kernel_size - 1) * self.dilation) / 2).astype(int),
            ),
            dilation=(self.dilation,),
        )

        self.output_size = self.compute_output_size()

        self.tau_m = nn.Parameter(torch.Tensor(self.output_size))
        self.tau_adp = nn.Parameter(torch.Tensor(self.output_size))

        nn.init.normal_(self.tau_m, tau_m, tau_m_inital_std)
        nn.init.normal_(self.tau_adp, tau_adp_inital, tau_adp_inital_std)

    def parameters(self):
        """parameters member function docstring"""
        return [self.dense.weight, self.dense.bias, self.tau_m, self.tau_adp]
```

```
def set_neuron_state(self, batch_size):
    """se_neuron_state member function docstring"""
    self.mem = (
        torch.zeros(batch_size, self.output_size[0], self.output_size[1])
        * B_J0
    ).to(self.device)

    self.spike = torch.zeros(
        batch_size, self.output_size[0], self.output_size[1]
    ).to(self.device)

    self.b = (
        torch.ones(batch_size, self.output_size[0], self.output_size[1])
        * B_J0
    ).to(self.device)

def forward(self, input_spike):
    """forward member function docstring"""
    d_input = self.conv(input_spike.float())
    if self.pooling is not None:
        d_input = self.pooling(d_input)
    (
        self.mem,
        self.spike,
        theta,  # pylint: disable=W0612
        self.b,
    ) = sn.mem_update_adp(
        d_input,
        self.mem,
```

```
                self.spike,
                self.tau_adp,
                self.b,
                self.tau_m,
                device=self.device,
                isAdapt=self.is_adaptive,
            )

            return self.mem, self.spike

    def compute_output_size(self):
        """compute_output member function docstring"""
        x_emp = torch.randn([1, self.input_size[0], self.input_size[1]])
        out = self.conv(x_emp)
        if self.pooling is not None:
            out = self.pooling(out)
        # print(self.name+'\'s size: ', out.shape[1:])
        return out.shape[1:]

class SpikeCov2D(nn.Module):  # pylint: disable=R0902
    """Spike_Cov2D docstring"""

    def __init__(  # pylint: disable=R0913
        self,
        input_size,
        output_dim,
        kernel_size=5,
        strides=1,
        pooling_type=None,
```

```
        pool_size=2,
        pool_strides=2,
        tau_m=20,
        tau_adp_inital=100,
        tau_initializer="normal",  # pylint: disable=W0613
        tau_m_inital_std=5,
        tau_adp_inital_std=5,
        is_adaptive=1,
        device="cpu",
    ):
        """Class constructor member function docstring"""
        super().__init__()
        self.mem = None
        self.spike = None
        self.b = None  # pylint: disable=C0103

        # input_size = [c,w,h]
        self.input_size = input_size
        self.input_dim = input_size[0]
        self.output_dim = output_dim
        self.is_adaptive = is_adaptive
        self.device = device

        if pooling_type is not None:
            if pooling_type == "max":
                self.pooling = nn.MaxPool2d(
                    kernel_size=pool_size, stride=pool_strides, padding=1
                )
            elif pooling_type == "avg":
```

```
                self.pooling = nn.AvgPool2d(
                    kernel_size=pool_size, stride=pool_strides, padding=1
                )
        else:
            self.pooling = None

        self.conv = nn.Conv2d(  # Look at the original!!!!
            self.input_dim, self.output_dim, kernel_size, strides
        )

        self.output_size = self.compute_output_size()

        self.tau_m = nn.Parameter(torch.Tensor(self.output_size))
        self.tau_adp = nn.Parameter(torch.Tensor(self.output_size))

        nn.init.normal_(self.tau_m, tau_m, tau_m_inital_std)
        nn.init.normal_(self.tau_adp, tau_adp_inital, tau_adp_inital_std)

    def parameters(self):
        """parameters member function docstring"""
        return [self.dense.weight, self.dense.bias, self.tau_m, self.tau_adp]

    def set_neuron_state(self, batch_size):
        """set_neuron_state member function docstring"""
        self.mem = torch.rand(batch_size, self.output_size).to(self.device)
        self.spike = torch.zeros(batch_size, self.output_size).to(self.device)
        self.b = (torch.ones(batch_size, self.output_size) * B_J0).to(
            self.device
        )
```

```
def forward(self, input_spike):
    """forward member function docstring"""
    d_input = self.conv(input_spike.float())
    if self.pooling is not None:
        d_input = self.pool(d_input)
    (
        self.mem,
        self.spike,
        theta,  # pylint: disable=W0612
        self.b,
    ) = sn.mem_update_adp(
        d_input,
        self.mem,
        self.spike,
        self.tau_adp,
        self.b,
        self.tau_m,
        device=self.device,
        isAdapt=self.is_adaptive,
    )

    return self.mem, self.spike

def compute_output_size(self):
    """compute_output_size member function docstring"""
    x_emp = torch.randn(
        [1, self.input_size[0], self.input_size[1], self.input_size[2]]
    )
    out = self.conv(x_emp)
```

```
        if self.pooling is not None:
            out = self.pooling(out)
        # print(self.name+'\'s size: ', out.shape[1:])
        return out.shape[1:]

# Local Variables:
# compile-command: "pyflakes spike_cnn.py; pylint-3 -d E0401 -f parseable spike_cnn.py" # NOQA, pylint: disable=C0301
# End:
```

8 spike_dense.py

```
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0

"""  module docstring """

__all__ = ["SpikeDENSE", "SpikeBIDENSE", "ReadoutIntegrator"]

import numpy as np
import torch
from torch import nn
from torch.autograd import Variable

from . import spike_neuron as sn

B_J0: float = sn.B_J0_VALUE

def multi_normal_initilization(
    param, means=[10, 200], stds=[5, 20]
):  # pylint: disable=W0102
    """multi_normal_initialization function

    The tensor returned is composed of multiple, equal length
    partitions each drawn from a normal distribution described
    by a mean and std. The shape of the returned tensor is the same
    at the original input tensor."""
```

```
        shape_list = param.shape
        if len(shape_list) == 1:
            num_total = shape_list[0]
        elif len(shape_list) == 2:
            num_total = shape_list[0] * shape_list[1]

        num_per_group = int(num_total / len(means))
        # if num_total%len(means) != 0:
        num_last_group = num_total % len(means)
        a = []  # pylint: disable=C0103
        for i in range(len(means)):  # pylint: disable=C0200
            a = (  # pylint: disable=C0103
                a
                + np.random.normal(means[i], stds[i], size=num_per_group).tolist()
            )

            if i == len(means) - 1:
                a = (  # pylint: disable=C0103
                    a
                    + np.random.normal(
                        means[i], stds[i], size=num_per_group + num_last_group
                    ).tolist()
                )
        p = np.array(a).reshape(shape_list)  # pylint: disable=C0103
        with torch.no_grad():
            param.copy_(torch.from_numpy(p).float())
        return param

    class SpikeDENSE(nn.Module):
```

```
    """Spike_Dense class docstring"""

    def __init__(  # pylint: disable=R0913,W0231
        self,
        input_dim,
        output_dim,
        tau_m=20,
        tau_adp_inital=200,
        tau_initializer="normal",  # pylint: disable=W0613
        tau_m_inital_std=5,
        tau_adp_inital_std=5,
        is_adaptive=1,
        device="cpu",
        bias=True,
    ):
        """Class constructor member function docstring"""
        super().__init__()
        self.mem = None
        self.spike = None
        self.b = None  # pylint: disable=C0103
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.is_adaptive = is_adaptive
        self.device = device

        self.dense = nn.Linear(input_dim, output_dim, bias=bias)

        # Parameters are Tensor subclasses, that have a very special
        # property when used with Module s - when they're assigned as
```

```
        # Module attributes they are automatically added to the list
        # of its parameters, and will appear e.g. in parameters() iterator.
        self.tau_m = nn.Parameter(torch.Tensor(self.output_dim))
        self.tau_adp = nn.Parameter(torch.Tensor(self.output_dim))

        if tau_initializer == "normal":
            # Initialize self.tau_m and self.tau_adp from a single
            # normal distributions.
            nn.init.normal_(self.tau_m, tau_m, tau_m_inital_std)
            nn.init.normal_(self.tau_adp, tau_adp_inital, tau_adp_inital_std)
        elif tau_initializer == "multi_normal":
            # Initialize self.tau_m and self.tau_adp from from
            # multiple normal distributions. tau_m and tar_adp_initial
            # must be lists of means and tar_m_initial_std and
            # tar_adp_initial_std must be lists of standard
            # deviations.
            self.tau_m = multi_normal_initilization(
                self.tau_m, tau_m, tau_m_inital_std
            )
            self.tau_adp = multi_normal_initilization(
                self.tau_adp, tau_adp_inital, tau_adp_inital_std
            )

    def parameters(self):
        """Return a list of parameters being trained."""
        # The latter two are module parameters; the first two aren't
        # Where is dense.weight defined or assigned?
        return [self.dense.weight, self.dense.bias, self.tau_m, self.tau_adp]
```

```
def set_neuron_state(self, batch_size):
    """Initialize mem, spike and b tensors.

    The Variable API has been deprecated: Variables are no
    longer necessary to use autograd with tensors. Autograd
    automatically supports Tensors with requires_grad set to
    True.
    """
    # self.mem = (torch.rand(batch_size, self.output_dim) * self.b_j0).to(
    #     self.device
    # )
    self.mem = Variable(
        torch.zeros(batch_size, self.output_dim) * B_J0
    ).to(self.device)

    self.spike = Variable(torch.zeros(batch_size, self.output_dim)).to(
        self.device
    )

    self.b = Variable(torch.ones(batch_size, self.output_dim) * B_J0).to(
        self.device
    )

def forward(self, input_spike):
    """SpikeDENSE forward pass"""

    d_input = self.dense(input_spike.float())
    (
        self.mem,
```

```
                self.spike,
                theta,  # pylint: disable=W0612
                self.b,
            ) = sn.mem_update_adp(
                d_input,
                self.mem,
                self.spike,
                self.tau_adp,
                self.b,
                self.tau_m,
                device=self.device,
                isAdapt=self.is_adaptive,
            )

            return self.mem, self.spike

class SpikeBIDENSE(nn.Module):  # pylint: disable=R0902
    """Spike_Bidense class docstring"""

    def __init__(  # pylint: disable=R0913
        self,
        input_dim1,
        input_dim2,
        output_dim,
        tau_m=20,
        tau_adp_inital=100,
        tau_initializer="normal",  # pylint: disable=W0613
        tau_m_inital_std=5,
        tau_adp_inital_std=5,
```

```
        is_adaptive=1,
        device="cpu",
    ):
        """Class constructor member function docstring"""
        super().__init__()
        self.mem = None
        self.spike = None
        self.b = None  # pylint: disable=C0103
        self.input_dim1 = input_dim1
        self.input_dim2 = input_dim2
        self.output_dim = output_dim
        self.is_adaptive = is_adaptive
        self.device = device

        self.dense = nn.Bilinear(input_dim1, input_dim2, output_dim)
        self.tau_m = nn.Parameter(torch.Tensor(self.output_dim))
        self.tau_adp = nn.Parameter(torch.Tensor(self.output_dim))

        if tau_initializer == "normal":
            nn.init.normal_(self.tau_m, tau_m, tau_m_inital_std)
            nn.init.normal_(self.tau_adp, tau_adp_inital, tau_adp_inital_std)
        elif tau_initializer == "multi_normal":
            self.tau_m = multi_normal_initilization(
                self.tau_m, tau_m, tau_m_inital_std
            )
            self.tau_adp = multi_normal_initilization(
                self.tau_adp, tau_adp_inital, tau_adp_inital_std
            )
```

```
    def parameters(self):
        """parameter member function docstring"""
        return [self.dense.weight, self.dense.bias, self.tau_m, self.tau_adp]

    def set_neuron_state(self, batch_size):
        """set_neuron_state member function docstring"""
        self.mem = (torch.rand(batch_size, self.output_dim) * B_J0).to(
            self.device
        )
        self.spike = torch.zeros(batch_size, self.output_dim).to(self.device)
        self.b = (torch.ones(batch_size, self.output_dim) * B_J0).to(
            self.device
        )

    def forward(self, input_spike1, input_spike2):
        """forward member function docstring"""
        d_input = self.dense(input_spike1.float(), input_spike2.float())
        (
            self.mem,
            self.spike,
            theta,  # pylint: disable=W0612
            self.b,
        ) = sn.mem_update_adp(
            d_input,
            self.mem,
            self.spike,
            self.tau_adp,
            self.b,
            self.tau_m,
```

```
            device=self.device,
            isAdapt=self.is_adaptive,
        )

        return self.mem, self.spike

class ReadoutIntegrator(nn.Module):
    """Redout_Integrator class docstring"""

    def __init__(  # pylint: disable=R0913
        self,
        input_dim,
        output_dim,
        tau_m=20,
        tau_initializer="normal",  # pylint: disable=W0613
        tau_m_inital_std=5,
        device="cpu",
        bias=True,
    ):
        """Class constructor member function"""
        super().__init__()
        self.mem = None

        # UNUSED?!
        self.spike = None
        self.b = None  # pylint: disable=C0103

        self.input_dim = input_dim
        self.output_dim = output_dim
```

```
        self.device = device

        self.dense = nn.Linear(input_dim, output_dim, bias=bias)
        self.tau_m = nn.Parameter(torch.Tensor(self.output_dim))

        nn.init.normal_(self.tau_m, tau_m, tau_m_inital_std)

    def parameters(self):
        """parameters member function docstring"""
        return [self.dense.weight, self.dense.bias, self.tau_m]

    def set_neuron_state(self, batch_size):
        """set_neuron_state member function docstring"""
        # self.mem = torch.rand(batch_size,self.output_dim).to(self.device)
        self.mem = (torch.zeros(batch_size, self.output_dim)).to(self.device)

    def forward(self, input_spike):
        """forward member function docstring"""
        d_input = self.dense(input_spike.float())
        self.mem = sn.output_Neuron(
            d_input, self.mem, self.tau_m, device=self.device
        )
        return self.mem

# Local Variables:
# compile-command: "pyflakes spike_dense.py; pylint-3 -d E0401 -f parseable spike_dense.py" # NOQA,
pylint: disable=C0301
# End:
```

9 spike_neuron.py

```python
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0

"""
This module contains one class and three functions that together
aree used to calculate the membrane potential of the various spiking
neurons defined in this package. In particular, the functions
mem_update_adp and output_Neuron are called in the forward member
function of the SpikeDENSE, SpikeBIDENSE, SpikeRNN, SpikeCov1D and
SpikeCov2D layer classes and the readout_integration classes
respectively.
"""

import math

# import numpy as np
import torch
from loguru import logger

# from torch import nn
from torch.nn import functional as F

# all = ["output_Neuron, mem_update_adp"]

SURROGATE_TYPE: str = "MG"
```

```
GAMMA: float = 0.5
LENS: float = 0.5
R_M: float = 1
BETA_VALUE: float = 0.184
B_JO_VALUE: float = 1.6
SCALE: float = 6.0
HIGHT: float = 0.15

# act_fun_adp = ActFunADP.apply

class NoSurrogateTypeException(Exception):
    pass

def gaussian(
    x: torch.Tensor,  # pylint: disable=C0103
    mu: float = 0.0,  # pylint: disable=C0103
    sigma: float = 0.5,
) -> torch.Tensor:
    """Gussian

    Used in the backward method of a custom autograd function class
    ActFunADP to approximate the gradiant in a surrogate function
    for back propogation.
    """
    return (
        torch.exp(-((x - mu) ** 2) / (2 * sigma**2))
        / torch.sqrt(2 * torch.tensor(math.pi))
        / sigma
    )
```

```
def mem_update_adp(  # pylint: disable=R0913
    inputs,
    mem,
    spike,
    tau_adp,
    b,  # pylint: disable=C0103
    tau_m,
    dt=1,  # pylint: disable=C0103
    isAdapt=1,  # pylint: disable=C0103
    device=None,
):  # pylint: disable=C0103
    """Update the membrane potential.

    Called in the forward member function of the SpikeDENSE,
    SpikeBIDENSE, SpikeRNN, SpikeCov1D and SpikeCov2D layer
    classes.
    """

    alpha = torch.exp(-1.0 * dt / tau_m).to(device)
    ro = torch.exp(-1.0 * dt / tau_adp).to(device)  # pylint: disable=C0103

    beta = BETA_VALUE if isAdapt else 0.0
    if isAdapt:
        beta = BETA_VALUE
    else:
        beta = 0.0

    b = ro * b + (1 - ro) * spike  # Hard reset equation 1.8 page 12.
    B = B_JO_VALUE + beta * b  # pylint: disable=C0103
```

```
    mem = mem * alpha + (1 - alpha) * R_M * inputs - B * spike * dt
    inputs_ = mem - B

    # Non spiking output
    spike = F.relu(inputs_)

    # For details about calling the 'apply' member function,
    # See: https://pytorch.org/docs/stable/autograd.html#function
    # Spiking output
    spike = ActFunADP.apply(inputs_)

    return mem, spike, B, b

def output_Neuron(
    inputs, mem, tau_m, dt=1, device=None
):  # pylint: disable=C0103
    """Output the membrane potential of a LIF neuron without spike

    The only appears of this function is in the forward member
    function of the ReadoutIntegrator layer class.
    """

    alpha = torch.exp(-1.0 * dt / tau_m).to(device)
    mem = mem * alpha + (1 - alpha) * inputs
    return mem

class ActFunADP(torch.autograd.Function):
    """ActFunADP
```

```
    Custom autograd function redefining how forward and backward
    passes are performed. This class is 'applied' in the
    mem_update_adp function to calculate the new spike value.

    For details about calling the 'apply' member function, See:
    https://pytorch.org/docs/stable/autograd.html#function
    """

    @staticmethod
    def forward(ctx, i):  # ? What is the type and dimension of i?
        """Redefine the default autograd forward pass function.
        inp = membrane potential- threshold

        Returns a tensor whose values are either 0 or 1 dependent
        upon their value in the input tensor i.

        """
        ctx.save_for_backward(i)

        return i.gt(0).float()  # is firing ???

    @staticmethod
    def backward(ctx, grad_output):
        """Defines a formula for differentiating during back propogation.

        Since the spike function is nondifferentiable, we
        approximate the back propogation gradients with one of
        several surrogate functions.
        """
```

```
        (result,) = ctx.saved_tensors
        # grad_input = grad_output.clone()
        # temp = abs(result) < lens
        if SURROGATE_TYPE == "G":
            # temp = gaussian(result, mu=0.0, sigma=LENS)
            temp = (
                torch.exp(-(result**2) / (2 * LENS**2))
                / torch.sqrt(2 * torch.tensor(math.pi))
                / LENS
            )
        elif SURROGATE_TYPE == "MG":
            temp = (
                gaussian(result, mu=0.0, sigma=LENS) * (1.0 + HIGHT)
                - gaussian(result, mu=LENS, sigma=SCALE * LENS) * HIGHT
                - gaussian(result, mu=-LENS, sigma=SCALE * LENS) * HIGHT
            )
        elif SURROGATE_TYPE == "linear":
            temp = F.relu(1 - result.abs())
        elif SURROGATE_TYPE == "slayer":
            temp = torch.exp(-5 * result.abs())
        else:
            logger.critical(
                "No Surrogate type chosen, so temp tensor is undefined."
            )
            raise NoSurrogateTypeException("No Surrogate type chosen.")
        return grad_output * temp.float() * GAMMA

# Local Variables:
```

```
# compile-command: "pyflakes spike_neuron.py; pylint-3 -d E0401 -f parseable spike_neuron.py" # NOQA,
pylint: disable=C0301
# End:
```

10 decorators.py

11 exceptions.py

12 gencat.py

13 gendfind.py

14 genfind.py

15 gengrep.py

16 genopen.py