Contents

1 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 snoop
# import deeplake
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
import efficient_spiking_networks.srnn_layers.spike_rnn as sr
from GSC.data import (  # pylint: disable=C0301
    MelSpectrogram,
    Normalize,
    Pad,
    Rescale,
    SpeechCommandsDataset,
)
from GSC.utils import generate_random_silence_files

pp = pprint.PrettyPrinter(indent=4, width=41, compact=True)

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

sys.path.append("..")

# from tqdm import tqdm_notebook

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

```
NUMBER_OF_WORKERS = 4 if device.type == "cpu" else 8
logger.info(f"The Dataloader will spawn {NUMBER_OF_WORKERS} worker processes.")

# Directories
TRAIN_DATA_ROOT = "./DATA/train"
TEST_DATA_ROOT = "./DATA/test"

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

# Isolate 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)}]"
)

# ls 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=}"))

SR = 16000
SIZE = 16000

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)

# Turn wav files into Melspectrograms

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

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])

# 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

BATCH_SIZE = 32

# Collect the training, testing and validation data

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

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"
)
test_dataloader = DataLoader(
    test_dataset,
```

```
        batch_size=BATCH_SIZE,
        shuffle=True,
        num_workers=NUMBER_OF_WORKERS,
        collate_fn=collate_fn,
    )

    #  breakpoint()

    # train_ds = deeplake.load("hub://activeloop/speech-commands-train")
    # train_dataloader = train_ds.pytorch(num_workers=0, batch_size=32, shuffle=True)  # noqa:E501 pylint:
    disable=C0301
    # test_ds = deeplake.load("hub://activeloop/speech-commands-test")
    # test_dataloader = test_ds.pytorch(num_workers=0, batch_size=32, shuffle=True)

    thr_func = sn.ActFunADP.apply
    IS_BIAS = True

    # Define the overall RNN network
    class RecurrentSpikingNetwork(nn.Module):

        """
        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),
        ]

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

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

```
model.to(device)

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

LEARNING_RATE = 3e-3  # 1.2e-2

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.Adamax(
#     [
#         {"params": base_params},
#         {"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},
#         #   {"params": model.dense_2.tau_adp, "lr": LEARNING_RATE * 10},
#         {"params": model.rnn_1.tau_adp, "lr": LEARNING_RATE * 2},
#     ],
#     lr=LEARNING_RATE,eps=1e-5
# )

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)

EPOCHS = 30

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}")

# finis

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

2 data.py

```python
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0
# flake8: noqa
# pylint: skip-file
# type: ignore
# REUSE-IgnoreStart
import os

import librosa
import numpy as np
import scipy.io.wavfile as wav
import torch
from torch.utils.data import Dataset
from utils import split_wav, txt2list

class SpeechCommandsDataset(Dataset):
    def __init__(
        self, data_root, label_dct, mode, transform=None, max_nb_per_class=None
    ):

        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" 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):
            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("ignored command: %s" % 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
```

```
                ]
                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(
                [label_weights[label] for label in self.labels]
            )

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

    def __getitem__(self, idx):

        filename = self.filenames[idx]
        item = wav.read(filename)[1].astype(float)
        m = np.max(np.abs(item))
        if m > 0:
            item /= m
        if self.transform is not None:
```

```
            item = self.transform(item)

        label = self.labels[idx]

        return item, label

class Pad:
    def __init__(self, size):

        self.size = size

    def __call__(self, wav):
        wav_size = wav.shape[0]
        pad_size = (self.size - wav_size) // 2
        padded_wav = np.pad(
            wav,
            ((pad_size, self.size - wav_size - pad_size),),
            "constant",
            constant_values=(0, 0),
        )
        return padded_wav

class RandomNoise:
    def __init__(self, noise_files, size, coef):

        self.size = size
        self.noise_files = noise_files
        self.coef = coef
```

```
    def __call__(self, wav):

        if np.random.random() < 0.8:

            noise_wav = get_random_noise(self.noise_files, self.size)
            noise_power = (noise_wav**2).mean()
            sig_power = (wav**2).mean()

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

        else:

            noisy_wav = wav

        return noisy_wav

class RandomShift:
    def __init__(self, min_shift, max_shift):

        self.min_shift = min_shift
        self.max_shift = max_shift

    def __call__(self, wav):

        shift = np.random.randint(self.min_shift, self.max_shift + 1)
        shifted_wav = np.roll(wav, shift)

        if shift > 0:
```

```
            shifted_wav[:shift] = 0
        elif shift < 0:
            shifted_wav[shift:] = 0

        return shifted_wav

class MelSpectrogram:
    def __init__(
        self,
        sr,
        n_fft,
        hop_length,
        n_mels,
        fmin,
        fmax,
        delta_order=None,
        stack=True,
    ):

        self.sr = sr
        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, wav):
```

```
        S = librosa.feature.melspectrogram(
            y=wav,
            sr=self.sr,
            n_fft=self.n_fft,
            hop_length=self.hop_length,
            n_mels=self.n_mels,
            fmax=self.fmax,
            fmin=self.fmin,
        )

        M = np.max(np.abs(S))
        if M > 0:
            feat = np.log1p(S / M)
        else:
            feat = S

        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)

        elif 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)

        else:
            return np.expand_dims(feat.T, 0)
```

```
class Rescale:
    def __call__(self, input):

        std = np.std(input, axis=1, keepdims=True)
        std[std == 0] = 1

        return input / std

class Normalize:
    def __call__(self, input):

        input_ = (input > 0.1) * input
        std = np.std(input_, axis=1, keepdims=True)
        std[std == 0] = 1

        return input / std

class WhiteNoise:
    def __init__(self, size, coef_max):

        self.size = size
        self.coef_max = coef_max

    def __call__(self, wav):

        noise_wav = np.random.normal(size=self.size)
        noise_power = (noise_wav**2).mean()
        sig_power = (wav**2).mean()
```

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

        noisy_wav = wav + coef * noise_wav * np.sqrt(sig_power / noise_power)

        return noisy_wav

# REUSE-IgnoreEnd
```

3 optim.py

```
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0
# flake8: noqa
# pylint: skip-file
# type: ignore
# REUSE-IgnoreStart
import math

import torch
from torch.optim.optimizer import Optimizer, required

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

class RAdam(Optimizer):
    def __init__(
        self,
        params,
        lr=1e-3,
        betas=(0.9, 0.999),
        eps=1e-8,
        weight_decay=0,
        degenerated_to_sgd=True,
    ):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
```

```
        if not 0.0 <= eps:
            raise ValueError("Invalid epsilon value: {}".format(eps))
        if not 0.0 <= betas[0] < 1.0:
            raise ValueError(
                "Invalid beta parameter at index 0: {}".format(betas[0])
            )
        if not 0.0 <= betas[1] < 1.0:
            raise ValueError(
                "Invalid beta parameter at index 1: {}".format(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 = dict(
            lr=lr,
            betas=betas,
            eps=eps,
            weight_decay=weight_decay,
            buffer=[[None, None, None] for _ in range(10)],
```

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

    def __setstate__(self, state):
        super(RAdam, self).__setstate__(state)

    def step(self, closure=None):

        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:

            for p in group["params"]:
                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(p_data_fp32)
    state["exp_avg_sq"] = torch.zeros_like(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 = buffered[1], buffered[2]
else:
    buffered[0] = state["step"]
    beta2_t = beta2 ** state["step"]
    N_sma_max = 2 / (1 - beta2) - 1
    N_sma = N_sma_max - 2 * state["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

# REUSE-IgnoreEnd
```

4 utils.pys

```
# SPDX-FileCopyrightText: 2021 Centrum Wiskunde en Informatica
#
# SPDX-License-Identifier: MPL-2.0
# flake8: noqa
# pylint: skip-file
# type: ignore
# REUSE-IgnoreStart
import matplotlib.pyplot as plt
import numpy as np
import scipy.io.wavfile as wav
import torch
from matplotlib.gridspec import GridSpec

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:
        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])

def get_random_noise(noise_files, size):

    noise_idx = np.random.choice(len(noise_files))
    fs, noise_wav = wav.read(noise_files[noise_idx])

    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(
    nb_files, noise_files, size, prefix, sr=16000
):

    for i in range(nb_files):

        silence_wav = get_random_noise(noise_files, size)
        wav.write(prefix + "_" + str(i) + ".wav", sr, silence_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

    # REUSE-IgnoreEnd
```

5 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:
```

6 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_JO = 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:
```

7 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
            )
```

```python
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:
```

8 spike_neuron.py

```
# 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"
```

```python
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
    )
```

```python
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:
```

9 decorators.py

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

""" Custom function decorators """

__all__ = ["initializer"]

import inspect
from functools import wraps

from .exceptions import InvalidContextError

def initializer(fun):
    """This decorator takes a class constructor signature
    and makes corresponding class member variables."""

    if fun.__name__ != "__init__":
        raise InvalidContextError(
            "Only applicable context is decorating a class constructor."
        )

    specs = inspect.getfullargspec(fun)

    @wraps(fun)
    def wrapper(self, *args, **kargs):
```

```
            for name, arg in list(zip(specs.args[1:], args)) + list(kargs.items()):
                setattr(self, name, arg)
            if specs.defaults is not None:
                for i in range(len(specs.defaults)):
                    index = -(i + 1)
                    if not hasattr(self, specs.args[index]):
                        setattr(self, specs.args[index], specs.defaults[index])
            fun(self, *args, **kargs)

        return wrapper

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

10 exceptions.py

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

""" Custom exceptions """

__all__ = ["InvalidContextError"]

class InvalidContextError(Exception):
    """Raise this exception when you want to signal an invalid context."""

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