```
# A Neural Network Architecture Combining Gated Recurrent Unit (GRU) and
# Support Vector Machine (SVM) for Intrusion Detection in Network Traffic Data
# Copyright (C) 2017  Abien Fred Agarap
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU Affero General Public License as published
# by the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU Affero General Public License for more details.
#
# You should have received a copy of the GNU Affero General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
# ==============================================================================

"""Implementation of the GRU+SVM model for Intrusion Detection"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

__version__ = "0.1.1"
__author__ = "Abien Fred Agarap"

import argparse
from utils import data
from models.gru_svm.gru_svm import GruSvm
```

```
# hyper-parameters for the model
BATCH_SIZE = 256
CELL_SIZE = 256
DROPOUT_P_KEEP = 0.85
HM_EPOCHS = 10
LEARNING_RATE = 1e-5
N_CLASSES = 2
SEQUENCE_LENGTH = 21
SVM_C = 0.5


def parse_args():
    parser = argparse.ArgumentParser(description="GRU+SVM for Intrusion Detection")
    group = parser.add_argument_group("Arguments")
    group.add_argument(
        "-o",
        "--operation",
        required=True,
        type=str,
        help='the operation to perform: "train" or "test"',
    )
    group.add_argument(
        "-t",
        "--train_dataset",
        required=False,
        type=str,
        help="the NumPy array training dataset (*.npy) to be used",
    )
```

```
group.add_argument(
    "-v",
    "--validation_dataset",
    required=True,
    type=str,
    help="the NumPy array validation dataset (*.npy) to be used",
)
group.add_argument(
    "-c",
    "--checkpoint_path",
    required=True,
    type=str,
    help="path where to save the trained model",
)
group.add_argument(
    "-l",
    "--log_path",
    required=False,
    type=str,
    help="path where to save the TensorBoard logs",
)
group.add_argument(
    "-m",
    "--model_name",
    required=False,
    type=str,
    help="filename for the trained model",
)
group.add_argument(
        "-r",
        "--result_path",
        required=True,
        type=str,
        help="path where to save the actual and predicted labels",
    )
    arguments = parser.parse_args()
    return arguments


def main(argv):

    if argv.operation == "train":
        # get the train data
        # features: train_data[0], labels: train_data[1]
        train_features, train_labels = data.load_data(dataset=argv.train_dataset)

        # get the validation data
        # features: validation_data[0], labels: validation_data[1]
        validation_features, validation_labels = data.load_data(
            dataset=argv.validation_dataset
        )

        # get the size of the dataset for slicing
        train_size = train_features.shape[0]
        validation_size = validation_features.shape[0]

        # slice the dataset to be exact as per the batch size
        # e.g. train_size = 1898322, batch_size = 256
# [:1898322-(1898322%256)] = [:1898240]
# 1898322 // 256 = 7415; 7415 * 256 = 1898240
train_features = train_features[: train_size - (train_size % BATCH_SIZE)]
train_labels = train_labels[: train_size - (train_size % BATCH_SIZE)]

# modify the size of the dataset to be passed on model.train()
train_size = train_features.shape[0]

# slice the dataset to be exact as per the batch size
validation_features = validation_features[
    : validation_size - (validation_size % BATCH_SIZE)
]
validation_labels = validation_labels[
    : validation_size - (validation_size % BATCH_SIZE)
]

# modify the size of the dataset to be passed on model.train()
validation_size = validation_features.shape[0]

# instantiate the model
model = GruSvm(
    alpha=LEARNING_RATE,
    batch_size=BATCH_SIZE,
    cell_size=CELL_SIZE,
    dropout_rate=DROPOUT_P_KEEP,
    num_classes=N_CLASSES,
    sequence_length=SEQUENCE_LENGTH,
    svm_c=SVM_C,
)
```

```
        # train the model
        model.train(
            checkpoint_path=argv.checkpoint_path,
            log_path=argv.log_path,
            model_name=argv.model_name,
            epochs=HM_EPOCHS,
            train_data=[train_features, train_labels],
            train_size=train_size,
            validation_data=[validation_features, validation_labels],
            validation_size=validation_size,
            result_path=argv.result_path,
        )
    elif argv.operation == "test":
        test_features, test_labels = data.load_data(dataset=argv.validation_dataset)

        test_size = test_features.shape[0]

        test_features = test_features[: test_size - (test_size % BATCH_SIZE)]
        test_labels = test_labels[: test_size - (test_size % BATCH_SIZE)]

        test_size = test_features.shape[0]

        GruSvm.predict(
            batch_size=BATCH_SIZE,
            cell_size=CELL_SIZE,
            dropout_rate=DROPOUT_P_KEEP,
            num_classes=N_CLASSES,
            test_data=[test_features, test_labels],
            test_size=test_size,
            checkpoint_path=argv.checkpoint_path,
            result_path=argv.result_path,
        )


if __name__ == "__main__":
    args = parse_args()

    main(argv=args)
```