```
# 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 [http://arxiv.org/abs/1709.03082] by A.F. Agarap"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

__version__ = "0.3.11"
__author__ = "Abien Fred Agarap"

import numpy as np
import os
import sys
```

```
import tensorflow as tf
import time



class GruSvm:
    """Implementation of the GRU+SVM model using TensorFlow"""

    def __init__(
        self,
        alpha,
        batch_size,
        cell_size,
        dropout_rate,
        num_classes,
        sequence_length,
        svm_c,
    ):
        """Initialize the GRU+SVM class

        Parameter
        ---------
        alpha : float
          The learning rate for the GRU+Softmax model.
        batch_size : int
          The number of batches to use for training/validation/testing.
        cell_size : int
          The size of cell state.
        dropout_rate : float
          The dropout rate to be used.
```

```
num_classes : int
  The number of classes in a dataset.
sequence_length : int
  The number of features in a dataset.
svm_c : float
  The SVM penalty parameter C.
"""
self.alpha = alpha
self.batch_size = batch_size
self.cell_size = cell_size
self.dropout_rate = dropout_rate
self.num_classes = num_classes
self.sequence_length = sequence_length
self.svm_c = svm_c


def __graph__():
   """Build the inference graph"""
   with tf.name_scope("input"):
      # [BATCH_SIZE, SEQUENCE_LENGTH]
      x_input = tf.placeholder(
         dtype=tf.uint8, shape=[None, self.sequence_length], name="x_input"
      )


      # [BATCH_SIZE, SEQUENCE_LENGTH, 10]
      x_onehot = tf.one_hot(
         indices=x_input,
         depth=10,
         on_value=1.0,
         off_value=0.0,
```

```
        name="x_onehot",
    )

    # [BATCH_SIZE]
    y_input = tf.placeholder(dtype=tf.uint8, shape=[None], name="y_input")

    # [BATCH_SIZE, N_CLASSES]
    y_onehot = tf.one_hot(
        indices=y_input,
        depth=self.num_classes,
        on_value=1.0,
        off_value=-1.0,
        name="y_onehot",
    )

state = tf.placeholder(
    dtype=tf.float32, shape=[None, self.cell_size], name="initial_state"
)

p_keep = tf.placeholder(dtype=tf.float32, name="p_keep")
learning_rate = tf.placeholder(dtype=tf.float32, name="learning_rate")

cell = tf.contrib.rnn.GRUCell(self.cell_size)
drop_cell = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=p_keep)

# outputs: [BATCH_SIZE, SEQUENCE_LENGTH, CELL_SIZE]
# states: [BATCH_SIZE, CELL_SIZE]
outputs, states = tf.nn.dynamic_rnn(
    drop_cell, x_onehot, initial_state=state, dtype=tf.float32
```

```
)

states = tf.identity(states, name="H")

with tf.name_scope("final_training_ops"):
    with tf.name_scope("weights"):
        weight = tf.get_variable(
            "weights",
            initializer=tf.random_normal(
                [self.cell_size, self.num_classes], stddev=0.01
            ),
        )
        self.variable_summaries(weight)
    with tf.name_scope("biases"):
        bias = tf.get_variable(
            "biases", initializer=tf.constant(0.1, shape=[self.num_classes])
        )
        self.variable_summaries(bias)
    hf = tf.transpose(outputs, [1, 0, 2])
    last = tf.gather(hf, int(hf.get_shape()[0]) - 1)
    with tf.name_scope("Wx_plus_b"):
        output = tf.matmul(last, weight) + bias
        tf.summary.histogram("pre-activations", output)

# L2-SVM
with tf.name_scope("svm"):
    regularization_loss = 0.5 * tf.reduce_sum(tf.square(weight))
    hinge_loss = tf.reduce_sum(
        tf.square(
```

```
            tf.maximum(
                tf.zeros([self.batch_size, self.num_classes]),
                1 - y_onehot * output,
            )
        )
    )
    with tf.name_scope("loss"):
        loss = regularization_loss + self.svm_c * hinge_loss
tf.summary.scalar("loss", loss)

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(
    loss
)

with tf.name_scope("accuracy"):
    predicted_class = tf.sign(output)
    predicted_class = tf.identity(predicted_class, name="prediction")
    with tf.name_scope("correct_prediction"):
        correct = tf.equal(
            tf.argmax(predicted_class, 1), tf.argmax(y_onehot, 1)
        )
    with tf.name_scope("accuracy"):
        accuracy = tf.reduce_mean(tf.cast(correct, "float"))
tf.summary.scalar("accuracy", accuracy)

# merge all the summaries collected from the TF graph
merged = tf.summary.merge_all()

# set class properties
```

```
        self.x_input = x_input
        self.y_input = y_input
        self.y_onehot = y_onehot
        self.p_keep = p_keep
        self.loss = loss
        self.optimizer = optimizer
        self.state = state
        self.states = states
        self.learning_rate = learning_rate
        self.predicted_class = predicted_class
        self.accuracy = accuracy
        self.merged = merged

    sys.stdout.write("\n<log> Building Graph...")
    __graph__()
    sys.stdout.write("</log>\n")

def train(
    self,
    checkpoint_path,
    log_path,
    model_name,
    epochs,
    train_data,
    train_size,
    validation_data,
    validation_size,
    result_path,
):
```

```
"""Trains the model

Parameter
---------
checkpoint_path : str
  The path where to save the trained model.
log_path : str
  The path where to save the TensorBoard summaries.
model_name : str
  The filename for the trained model.
epochs : int
  The number of passes through the whole dataset.
train_data : numpy.ndarray
  The NumPy array training dataset.
train_size : int
  The size of `train_data`.
validation_data : numpy.ndarray
  The NumPy array testing dataset.
validation_size : int
  The size of `validation_data`.
result_path : str
  The path where to save the actual and predicted classes array.
"""

if not os.path.exists(path=checkpoint_path):
    os.mkdir(path=checkpoint_path)

saver = tf.train.Saver(max_to_keep=1000)
```

```
# initialize H (current_state) with values of zeros
current_state = np.zeros([self.batch_size, self.cell_size])

# variables initializer
init_op = tf.group(
   tf.global_variables_initializer(), tf.local_variables_initializer()
)

# get the time tuple
timestamp = str(time.asctime())

train_writer = tf.summary.FileWriter(
   logdir=os.path.join(log_path, timestamp + "-training"),
   graph=tf.get_default_graph(),
)
validation_writer = tf.summary.FileWriter(
   logdir=os.path.join(log_path, timestamp + "-validation"),
   graph=tf.get_default_graph(),
)

with tf.Session() as sess:
   sess.run(init_op)

   checkpoint = tf.train.get_checkpoint_state(checkpoint_path)

   if checkpoint and checkpoint.model_checkpoint_path:
      saver = tf.train.import_meta_graph(
         checkpoint.model_checkpoint_path + ".meta"
      )
```

```
    saver.restore(sess, tf.train.latest_checkpoint(checkpoint_path))

try:
    for step in range(epochs * train_size // self.batch_size):

        # set the value for slicing
        # e.g. step = 0, batch_size = 256, train_size = 1898240
        # (0 * 256) % 1898240 = 0
        # [offset:(offset + batch_size)] = [0:256]
        offset = (step * self.batch_size) % train_size
        train_example_batch = train_data[0][
            offset : (offset + self.batch_size)
        ]
        train_label_batch = train_data[1][
            offset : (offset + self.batch_size)
        ]

        # dictionary for key-value pair input for training
        feed_dict = {
            self.x_input: train_example_batch,
            self.y_input: train_label_batch,
            self.state: current_state,
            self.learning_rate: self.alpha,
            self.p_keep: self.dropout_rate,
        }

        train_summary, _, predictions, actual, next_state = sess.run(
            [
                self.merged,
```

```
        self.optimizer,
        self.predicted_class,
        self.y_onehot,
        self.states,
    ],
    feed_dict=feed_dict,
)

# Display training loss and accuracy every 100 steps and at step 0
if step % 100 == 0:
    # get train loss and accuracy
    train_loss, train_accuracy = sess.run(
        [self.loss, self.accuracy], feed_dict=feed_dict
    )

    # display train loss and accuracy
    print(
        "step [{}] train -- loss : {}, accuracy : {}".format(
            step, train_loss, train_accuracy
        )
    )

    # write the train summary
    train_writer.add_summary(train_summary, step)

    # save the model at current step
    saver.save(
        sess=sess,
        save_path=os.path.join(checkpoint_path, model_name),
```

```
                global_step=step,
            )

        current_state = next_state

        self.save_labels(
            predictions=predictions,
            actual=actual,
            result_path=result_path,
            step=step,
            phase="training",
        )
    except KeyboardInterrupt:
        print("Training interrupted at {}".format(step))
        os._exit(1)
    finally:
        print("EOF -- Training done at step {}".format(step))

        for step in range(epochs * validation_size // self.batch_size):

            offset = (step * self.batch_size) % validation_size
            test_example_batch = validation_data[0][
                offset : (offset + self.batch_size)
            ]
            test_label_batch = validation_data[1][
                offset : (offset + self.batch_size)
            ]

            # dictionary for key-value pair input for validation
```

```
feed_dict = {
    self.x_input: test_example_batch,
    self.y_input: test_label_batch,
    self.state: np.zeros([self.batch_size, self.cell_size]),
    self.p_keep: 1.0,
}

(
    validation_summary,
    predictions,
    actual,
    validation_loss,
    validation_accuracy,
) = sess.run(
    [
        self.merged,
        self.predicted_class,
        self.y_onehot,
        self.loss,
        self.accuracy,
    ],
    feed_dict=feed_dict,
)

# Display validation loss and accuracy every 100 steps
if step % 100 == 0 and step > 0:

    # add the validation summary
    validation_writer.add_summary(validation_summary, step)
```

```
                # display validation loss and accuracy
                print(
                    "step [{}] validation -- loss : {}, accuracy : {}".format(
                        step, validation_loss, validation_accuracy
                    )
                )

            self.save_labels(
                predictions=predictions,
                actual=actual,
                result_path=result_path,
                step=step,
                phase="validation",
            )

        print("EOF -- Testing done at step {}".format(step))

@staticmethod
def predict(
    batch_size,
    cell_size,
    dropout_rate,
    num_classes,
    test_data,
    test_size,
    checkpoint_path,
    result_path,
):
```

"""Classifies the data whether there is an intrusion or none

Parameter

---

batch_size : int

  The number of batches to use for training/validation/testing.

cell_size : int

  The size of cell state.

dropout_rate : float

  The dropout rate to be used.

num_classes : int

  The number of classes in a dataset.

test_data : numpy.ndarray

  The NumPy array testing dataset.

test_size : int

  The size of `test_data`.

checkpoint_path : str

  The path where to save the trained model.

result_path : str

  The path where to save the actual and predicted classes array.

"""

create initial RNN state array, filled with zeros

initial_state = np.zeros([batch_size, cell_size])

cast the array to float32

initial_state = initial_state.astype(np.float32)

variables initializer

```
init_op = tf.group(
   tf.global_variables_initializer(), tf.local_variables_initializer()
)

with tf.Session() as sess:
   sess.run(init_op)

   # get the checkpoint file
   checkpoint = tf.train.get_checkpoint_state(checkpoint_path)

   if checkpoint and checkpoint.model_checkpoint_path:
      # if checkpoint file exists, load the saved meta graph
      saver = tf.train.import_meta_graph(
         checkpoint.model_checkpoint_path + ".meta"
      )
      # and restore previously saved variables
      saver.restore(sess, tf.train.latest_checkpoint(checkpoint_path))
      print(
         "Loaded model from {}".format(
            tf.train.latest_checkpoint(checkpoint_path)
         )
      )

   try:
      for step in range(test_size // batch_size):

         offset = (step * batch_size) % test_size
         test_features_batch = test_data[0][offset : (offset + batch_size)]
         test_labels_batch = test_data[1][offset : (offset + batch_size)]
```

```
# one-hot encode labels according to NUM_CLASSES
label_onehot = tf.one_hot(test_labels_batch, num_classes, 1.0, -1.0)
y_onehot = sess.run(label_onehot)

# dictionary for input values for the tensors
feed_dict = {
   "input/x_input:0": test_features_batch,
   "initial_state:0": initial_state,
   "p_keep:0": dropout_rate,
}

# get the tensor for classification
prediction_tensor = sess.graph.get_tensor_by_name(
   "accuracy/prediction:0"
)
predictions = sess.run(prediction_tensor, feed_dict=feed_dict)

# add key, value pair for labels
feed_dict["input/y_input:0"] = test_labels_batch

# get the tensor for calculating the classification accuracy
accuracy_tensor = sess.graph.get_tensor_by_name(
   "accuracy/accuracy/Mean:0"
)
accuracy = sess.run(accuracy_tensor, feed_dict=feed_dict)

if step % 100 == 0 and step > 0:
   print("step [{}] test -- accuracy : {}".format(step, accuracy))
```

```
            GruSvm.save_labels(
                predictions=predictions,
                actual=y_onehot,
                result_path=result_path,
                phase="testing",
                step=step,
            )

        except KeyboardInterrupt:
            print("KeyboardInterrupt at step {}".format(step))
        finally:
            print("Done classifying at step {}".format(step))

@staticmethod
def variable_summaries(var):
    with tf.name_scope("summaries"):
        mean = tf.reduce_mean(var)
        tf.summary.scalar("mean", mean)
        with tf.name_scope("stddev"):
            stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
        tf.summary.scalar("stddev", stddev)
        tf.summary.scalar("max", tf.reduce_max(var))
        tf.summary.scalar("min", tf.reduce_min(var))
        tf.summary.histogram("histogram", var)

@staticmethod
def save_labels(predictions, actual, result_path, step, phase):
    """Saves the actual and predicted labels to a NPY file
```

```
Parameter
---------
predictions : numpy.ndarray
  The NumPy array containing the predicted labels.
actual : numpy.ndarray
  The NumPy array containing the actual labels.
result_path : str
  The path where to save the concatenated actual and predicted labels.
step : int
  The time step for the NumPy arrays.
phase : str
  The phase for which the predictions is, i.e. training/validation/testing.
"""

# Concatenate the predicted and actual labels
labels = np.concatenate((predictions, actual), axis=1)

# Create the result_path directory if it does not exist
if not os.path.exists(path=result_path):
    os.mkdir(path=result_path)

# Save the labels array to NPY file
np.save(
    file=os.path.join(result_path, "{}-gru_svm-{}.npy".format(phase, step)),
    arr=labels,
)
```