```
"""An implementation of L2-SVM model"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

__version__ = "0.3.7"
__author__ = "Abien Fred Agarap"

import numpy as np
import os
import sys
```

```
import tensorflow as tf
import time


class Svm:
    """Implementation of L2-Support Vector Machine using TensorFlow"""

    def __init__(self, alpha, batch_size, svm_c, num_classes, num_features):
        """Initialize the SVM class

        Parameter
        ---------
        alpha : float
          The learning rate for the SVM model.
        batch_size : int
          Number of batches to use for training and testing.
        svm_c : float
          The SVM penalty parameter.
        num_classes : int
          Number of classes in a dataset.
        num_features : int
          Number of features in a dataset.
        """
        self.alpha = alpha
        self.batch_size = batch_size
        self.svm_c = svm_c
        self.num_classes = num_classes
        self.num_features = num_features
```

```
def __graph__():
    """Building the inference graph"""

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

    with tf.name_scope("input"):
        # [BATCH_SIZE, SEQUENCE_LENGTH]
        x_input = tf.placeholder(
            dtype=tf.float32, shape=[None, self.num_features], name="x_input"
        )

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

        y_onehot = tf.one_hot(
            indices=y_input,
            depth=self.num_classes,
            on_value=1,
            off_value=-1,
            name="y_onehot",
        )

    with tf.name_scope("training_ops"):
        with tf.name_scope("weights"):
            weight = tf.get_variable(
                name="weights",
                initializer=tf.random_normal(
                    [self.num_features, self.num_classes], stddev=0.01
                ),
```

```
        )
        self.variable_summaries(weight)
    with tf.name_scope("biases"):
        bias = tf.get_variable(
            name="biases",
            initializer=tf.constant(0.1, shape=[self.num_classes]),
        )
        self.variable_summaries(bias)
    with tf.name_scope("Wx_plus_b"):
        y_hat = tf.matmul(x_input, weight) + bias
        tf.summary.histogram("pre-activations", y_hat)

# L2-SVM
with tf.name_scope("svm"):
    regularization = 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 - tf.cast(y_onehot, tf.float32) * y_hat,
            )
        )
    )
    with tf.name_scope("loss"):
        loss = regularization + 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(y_hat)
        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 in the inference graph
    merged = tf.summary.merge_all()

    self.x_input = x_input
    self.y_input = y_input
    self.y_onehot = y_onehot
    self.loss = loss
    self.optimizer = optimizer
    self.predicted_class = predicted_class
    self.learning_rate = learning_rate
    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,
    result_path,
    train_data,
    train_size,
    validation_data,
    validation_size,
):
    """Trains the SVM model

    Parameter
    ---------
    checkpoint_path : str
      The directory where to save the trained model.
    log_path : str
      The directory where to save the TensorBoard logs.
    model_name : str
      The filename of the trained model.
    epochs : int
      The number of passes through the entire dataset.
    result_path : str
      The path where to save the NPY files consisting of the actual and predicted labels.
    train_data : numpy.ndarray
      The numpy.ndarray to be used as the training dataset.
```

```
train_size : int
  The number of data in `train_data`.
validation_data : numpy.ndarray
  The numpy.ndarray to be used as the validation dataset.
validation_size : int
  The number of data in `validation_data`.
"""

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

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

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

# get the time tuple, and parse to str
timestamp = str(time.asctime())

# event file to contain TF graph summaries during training
train_writer = tf.summary.FileWriter(
    log_path + timestamp + "-training", graph=tf.get_default_graph()
)

# event file to contain TF graph summaries during validation
validation_writer = tf.summary.FileWriter(
    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)

    # check if a trained model exists
    if checkpoint and checkpoint.model_checkpoint_path:
        # load the graph from the trained model
        saver = tf.train.import_meta_graph(
            checkpoint.model_checkpoint_path + ".meta"
        )
        # restore the variables
        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, to fetch batches of data
            offset = (step * self.batch_size) % train_size
            train_feature_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_feature_batch,
    self.y_input: train_label_batch,
    self.learning_rate: self.alpha,
}

train_summary, _, predictions, actual = sess.run(
    [
        self.merged,
        self.optimizer,
        self.predicted_class,
        self.y_onehot,
    ],
    feed_dict=feed_dict,
)

# display training accuracy and loss every 100 steps and at step 0
if step % 100 == 0:

    # get the train loss and train accuracy
    train_accuracy, train_loss = sess.run(
        [self.accuracy, self.loss], feed_dict=feed_dict
    )

    # display the train loss and train 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 the current time step
        saver.save(sess, checkpoint_path + model_name, global_step=step)

    self.save_labels(
        predictions=predictions,
        actual=actual,
        result_path=result_path,
        phase="training",
        step=step,
    )
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
        validation_feature_batch = validation_data[0][
            offset : (offset + self.batch_size)
        ]
        validation_label_batch = validation_data[1][
```

```
    offset : (offset + self.batch_size)
]

# dictionary for key-value pair input for validation
feed_dict = {
    self.x_input: validation_feature_batch,
    self.y_input: validation_label_batch,
}

(
    test_summary,
    predictions,
    actual,
    test_loss,
    test_accuracy,
) = sess.run(
    [
        self.merged,
        self.predicted_class,
        self.y_onehot,
        self.loss,
        self.accuracy,
    ],
    feed_dict=feed_dict,
)

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

```
                # write the validation summary
                validation_writer.add_summary(test_summary, step)

                print(
                    "step [{}] validation -- loss : {}, accuracy : {}".format(
                        step, test_loss, test_accuracy
                    )
                )

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

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

@staticmethod
def predict(
    batch_size, 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.
```

```
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.
"""

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

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

    checkpoint = tf.train.get_checkpoint_state(checkpoint_path)

    # check if trained model exists
    if checkpoint and checkpoint.model_checkpoint_path:
        # load the trained model
        saver = tf.train.import_meta_graph(
            checkpoint.model_checkpoint_path + ".meta"
        )
        # restore the 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_example_batch = test_data[0][offset : (offset + batch_size)]
        test_label_batch = test_data[1][offset : (offset + batch_size)]

        # dictionary for input values for the tensors
        feed_dict = {"input/x_input:0": test_example_batch}

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

        label_onehot = tf.one_hot(test_label_batch, num_classes, 1.0, -1.0)
        y_onehot = sess.run(label_onehot)

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

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

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

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

```
    arr=labels,
)
```