from collections import defaultdict

import os

from toxicity\_ml\_pipeline.settings.default\_settings\_tox import REMOTE\_LOGDIR

from toxicity\_ml\_pipeline.settings.default\_settings\_abs import LABEL\_NAMES

from toxicity\_ml\_pipeline.utils.absv\_utils import parse\_labeled\_data

from toxicity\_ml\_pipeline.utils.helpers import compute\_precision\_fixed\_recall, execute\_command

from sklearn.metrics import average\_precision\_score, roc\_auc\_score

import tensorflow as tf

import wandb

class NothingCallback(tf.keras.callbacks.Callback):

def on\_epoch\_begin(self, epoch, logs=None):

print("ici, ", epoch)

def on\_epoch\_end(self, epoch, logs=None):

print("fin ", epoch)

def on\_train\_batch\_end(self, batch, logs=None):

print("fin de batch ", batch)

class ControlledStoppingCheckpointCallback(tf.keras.callbacks.ModelCheckpoint):

def \_\_init\_\_(self, stopping\_epoch, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.stopping\_epoch = stopping\_epoch

def on\_epoch\_end(self, epoch, logs=None):

super().on\_epoch\_end(epoch, logs)

if epoch == self.stopping\_epoch:

self.model.stop\_training = True

class SyncingTensorBoard(tf.keras.callbacks.TensorBoard):

def \_\_init\_\_(self, remote\_logdir=None, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.remote\_logdir = remote\_logdir if remote\_logdir is not None else REMOTE\_LOGDIR

def on\_epoch\_end(self, epoch, logs=None):

super().on\_epoch\_end(epoch, logs=logs)

self.synchronize()

def synchronize(self):

base\_dir = os.path.dirname(self.log\_dir)

cmd = f"gsutil -m rsync -r {base\_dir} {self.remote\_logdir}"

execute\_command(cmd)

class GradientLoggingTensorBoard(SyncingTensorBoard):

def \_\_init\_\_(self, loader, val\_data, freq, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

val\_dataset = loader.get\_balanced\_dataset(

training\_data=val\_data, size\_limit=50, return\_as\_batch=False

)

data\_args = list(val\_dataset.batch(32).take(1))[0]

self.x\_batch, self.y\_batch = data\_args[0], data\_args[1]

self.freq = freq

self.counter = 0

def \_log\_gradients(self):

writer = self.\_train\_writer

with writer.as\_default():

with tf.GradientTape() as tape:

y\_pred = self.model(self.x\_batch)

loss = self.model.compiled\_loss(y\_true=self.y\_batch, y\_pred=y\_pred)

gradient\_norm = tf.linalg.global\_norm(tape.gradient(loss, self.model.trainable\_weights))

tf.summary.scalar("gradient\_norm", data=gradient\_norm, step=self.counter)

writer.flush()

def on\_train\_batch\_end(self, batch, logs=None):

super().on\_batch\_end(batch, logs=logs)

self.counter += 1

if batch % self.freq == 0:

self.\_log\_gradients()

class AdditionalResultLogger(tf.keras.callbacks.Callback):

def \_\_init\_\_(

self,

data,

set\_,

fixed\_recall=0.85,

from\_logits=False,

dataset\_transform\_func=None,

batch\_size=64,

dual\_head=None,

\*args,

\*\*kwargs,

):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.set\_ = set\_

if data is None:

return None

self.single\_head = True

try:

self.labels = data.int\_label.values

except AttributeError:

self.labels = data.to\_dataframe()[LABEL\_NAMES].values.astype('int')

self.data = data.to\_tf\_dataset().map(parse\_labeled\_data).batch(batch\_size)

self.label\_names = LABEL\_NAMES

else:

self.label\_names = ['']

if dual\_head:

self.label\_names = [f'{e}\_label' for e in dual\_head]

self.labels = {f'{e}\_output': data[f'{e}\_label'].values for e in dual\_head}

self.single\_head = False

if dataset\_transform\_func is None:

self.data = data.text.values

else:

self.data = dataset\_transform\_func(data, mb\_size=batch\_size, shuffle=False)

finally:

if len(self.label\_names) == 1:

self.metric\_kw = {}

else:

self.metric\_kw = {'average': None}

self.counter = 0

self.best\_metrics = defaultdict(float)

self.from\_logits = from\_logits

print(f"Loaded callback for {set\_}, from\_logits: {from\_logits}, labels {self.label\_names}")

if 1 < fixed\_recall <= 100:

fixed\_recall = fixed\_recall / 100

elif not (0 < fixed\_recall <= 100):

raise ValueError("Threshold should be between 0 and 1, or 0 and 100")

self.fixed\_recall = fixed\_recall

self.batch\_size = batch\_size

def compute\_precision\_fixed\_recall(self, labels, preds):

result, \_ = compute\_precision\_fixed\_recall(labels=labels, preds=preds,

fixed\_recall=self.fixed\_recall)

return result

def on\_epoch\_end(self, epoch, logs=None):

self.additional\_evaluations(step=epoch, eval\_time="epoch")

def on\_train\_batch\_end(self, batch, logs=None):

self.counter += 1

if self.counter % 2000 == 0:

self.additional\_evaluations(step=self.counter, eval\_time="batch")

def \_binary\_evaluations(self, preds, label\_name=None, class\_index=None):

mask = None

curr\_labels = self.labels

if label\_name is not None:

curr\_labels = self.labels[label\_name]

if class\_index is not None:

curr\_labels = (curr\_labels == class\_index).astype(int)

if -1 in curr\_labels:

mask = curr\_labels != -1

curr\_labels = curr\_labels[mask]

preds = preds[mask]

return {

f"precision\_recall{self.fixed\_recall}": self.compute\_precision\_fixed\_recall(

labels=curr\_labels, preds=preds

),

"pr\_auc": average\_precision\_score(y\_true=curr\_labels, y\_score=preds),

"roc\_auc": roc\_auc\_score(y\_true=curr\_labels, y\_score=preds),

}

def \_multiclass\_evaluations(self, preds):

pr\_auc\_l = average\_precision\_score(y\_true=self.labels, y\_score=preds, \*\*self.metric\_kw)

roc\_auc\_l = roc\_auc\_score(y\_true=self.labels, y\_score=preds, \*\*self.metric\_kw)

metrics = {}

for i, label in enumerate(self.label\_names):

metrics[f'pr\_auc\_{label}'] = pr\_auc\_l[i]

metrics[f'roc\_auc\_{label}'] = roc\_auc\_l[i]

return metrics

def additional\_evaluations(self, step, eval\_time):

print("Evaluating ", self.set\_, eval\_time, step)

preds = self.model.predict(x=self.data, batch\_size=self.batch\_size)

if self.from\_logits:

preds = tf.keras.activations.sigmoid(preds.logits).numpy()

if self.single\_head:

if len(self.label\_names) == 1:

metrics = self.\_binary\_evaluations(preds)

else:

metrics = self.\_multiclass\_evaluations(preds)

else:

if preds[0].shape[1] == 1:

binary\_preds = preds[0]

multic\_preds = preds[1]

else:

binary\_preds = preds[1]

multic\_preds = preds[0]

binary\_metrics = self.\_binary\_evaluations(binary\_preds, label\_name='target\_output')

metrics = {f'{k}\_target': v for k, v in binary\_metrics.items()}

num\_classes = multic\_preds.shape[1]

for class\_ in range(num\_classes):

binary\_metrics = self.\_binary\_evaluations(multic\_preds[:, class\_], label\_name='content\_output', class\_index=class\_)

metrics.update({f'{k}\_content\_{class\_}': v for k, v in binary\_metrics.items()})

for k, v in metrics.items():

self.best\_metrics[f"max\_{k}"] = max(v, self.best\_metrics[f"max\_{k}"])

self.log\_metrics(metrics, step=step, eval\_time=eval\_time)

def log\_metrics(self, metrics\_d, step, eval\_time):

commit = False if self.set\_ == "validation" else True

to\_report = {self.set\_: {\*\*metrics\_d, \*\*self.best\_metrics}}

if eval\_time == "epoch":

to\_report["epoch"] = step

wandb.log(to\_report, commit=commit)