import tensorflow as tf from tfinterface.base import Model from tfinterface.metrics import r2_score, sigmoid_score, softmax_score from tfinterface.utils import huber_loss import cytoolz as cz import itertools as it from tfinterface.decorators import return_self, with_graph_as_default, copy_self from .supervised_inputs import SupervisedInputs from abc import abstractmethod, ABCMeta

class GeneralSupervisedModel(Model): """

Inteface

- `inputs : SupervisedInputs` -
- `predictions : Tensor` -
- `loss : Tensor` -

- `update : Tensor` - """

  metaclass = ABCMeta

  def **init**(self, name, optimizer=tf.train.AdamOptimizer, learning_rate=0.001, **kwargs):

  ```
      super(GeneralSupervisedModel, self).__init__(name, **kwargs)

      self._optimizer = optimizer
      self._learning_rate_arg = learning_rate
  ```

  @return_self def build_tensors(self, *args,* *kwargs):

  ```
      super(GeneralSupervisedModel, self).build_tensors(*args, **kwargs)
  ```

```
    self.learning_rate = self.get_learning_rate(*args, **kwargs)
    self.predictions = self.get_predictions(*args, **kwargs)
    self.loss = self.get_loss(*args, **kwargs)
    self.score_tensor = self.get_score_tensor(*args, **kwargs)
    self.update = self.get_update(*args, **kwargs)
    self.summaries = self.get_all_summaries(*args, **kwargs)



def get_learning_rate(self, *args, **kwargs):
    if hasattr(self.inputs, "learning_rate"):
        return self.inputs.learning_rate
    else:
        return self._learning_rate_arg

@abstractmethod
def get_predictions(self, *args, **kwargs):
```

```
    pass

@abstractmethod
def get_loss(self, *args, **kwargs):
    pass

@abstractmethod
def get_score_tensor(self, *args, **kwargs):
    pass

def get_update(self, *args, **kwargs):
    return self._optimizer(self.learning_rate).minimize(
        self.loss,
        global_step = self.inputs.global_step
    )

def get_all_summaries(self, *args, **kwargs):
    standard = self.get_standard_summaries()
    summaries = self.get_summaries(*args, **kwargs)


    return tf.summary.merge(standard + summaries)


def get_standard_summaries(self):
    return [
        tf.summary.scalar("loss_summary", self.loss),
        tf.summary.scalar("score_summary", self.score_tensor)
    ]

def get_summaries(self, *args, **kwargs):
    return []

def predict(self, **kwargs):
    predict_feed = self.inputs.predict_feed(**kwargs)
    return self.sess.run(self.predictions, feed_dict=predict_feed)


def score(self, **kwargs):
    predict_feed = self.inputs.predict_feed(**kwargs)
    score = self.sess.run(self.score_tensor, feed_dict=predict_feed)

    return score


@with_graph_as_default
@return_self
def fit(self, epochs=2000, data_generator=None, log_summaries=False, log_interval=20, print_te
    if log_summaries and not hasattr(self, "writer"):
        self.writer = tf.summary.FileWriter(self.logs_path, graph=self.graph, **writer_kwargs)

    if not hasattr(self, "summaries"):
        self.summaries = tf.no_op()

    if data_generator is None:
        #generator of empty dicts
        data_generator = it.repeat({})

    data_generator = data_generator |> cz.take$(epochs)
    for step, batch_feed_data in enumerate(data_generator):

        fit_feed = self.inputs.fit_feed(**batch_feed_data)
        _, summaries = self.sess.run([self.update, self.summaries], feed_dict=fit_feed)

        if log_summaries and step % log_interval == 0 and summaries is not None:
            self.writer.add_summary(
                summaries,
                global_step = self.sess.run(self.inputs.global_step)
            )


        if print_test_info and step % log_interval == 0:
            loss, score = self.sess.run([self.loss, self.score_tensor], feed_dict=fit_feed)
            print("loss {}, score {}, at {}".format(loss, score, step))


        ################
        # on_train
        ################
        kwargs = dict(
            step = step,
            loss = loss,
            score = score
        )

        for command in on_train:
            when = command.get("when", lambda **kwargs: True)
            do = command.get("do")

            if when(**kwargs):
                do(**kwargs)
```

class SupervisedModel(GeneralSupervisedModel): """docstring for SupervisedModel."""

```
@return_self
def build_tensors(self, *args, **kwargs):
    self.labels = self.get_labels(*args, **kwargs)
    super(SupervisedModel, self).build_tensors(*args, **kwargs)

def get_labels(self, inputs, *args, **kwargs):
    return inputs.labels
```

class SoftmaxClassifier(SupervisedModel): """docstring for SoftmaxClassifier."""

```
@abstractmethod
def get_logits(self):
    pass

def get_predictions(self, *args, **kwargs):
    self.logits = self.get_logits(*args, **kwargs)
    return tf.nn.softmax(self.logits)


def get_loss(self, *args, **kwargs):
    return (
        tf.nn.softmax_cross_entropy_with_logits(logits=self.logits, labels=self.labels)
        |> tf.reduce_mean
    )


def get_score_tensor(self, *args, **kwargs):
    return softmax_score(self.predictions, self.labels)
```

class SigmoidClassifier(SupervisedModel): """docstring for SoftmaxClassifier."""

```
@abstractmethod
def get_logits(self):
    pass

def get_predictions(self, *args, **kwargs):
    self.logits = self.get_logits(*args, **kwargs)

    return tf.nn.sigmoid(self.logits)


def get_loss(self, *args, **kwargs):
    return (
        tf.nn.sigmoid_cross_entropy_with_logits(logits=self.logits, labels=self.labels)
        |> tf.reduce_mean
    )


def get_score_tensor(self, *args, **kwargs):
    return sigmoid_score(self.predictions, self.labels)
```

class RegressionModel(SupervisedModel): """docstring for SoftmaxClassifier."""

```
def __init__(self, *args, **kwargs):
    loss = kwargs.pop("loss", "mse")

    if loss == "mse":
        loss = lambda error: tf.nn.l2_loss(error) * 2

    elif loss == "huber":
        loss = huber_loss

    self._loss_fn = loss

    super(RegressionModel, self).__init__(*args, **kwargs)



def get_loss(self, *args, **kwargs):
    error = self.predictions - self.labels
    return self._loss_fn(error) |> tf.reduce_mean


def get_score_tensor(self, *args, **kwargs):
    return r2_score(self.predictions, self.labels)
```