iris_custom_full_para_object2.py

```
import tensorflow as tf
from datetime import datetime
import path,os
from typing import Dict,List,Tuple,Optional,Any

from types import SimpleNamespace


#  ######################数据ETL############################
class ParameterSetter:

    def set_parameter(self, key, value):
        if hasattr(self, key):
            setattr(self, key, value)
        else:
            raise KeyError("没有该属性，请检查")
        self._update_dependency(key)

    def _update_dependency(self, key, value=None):
        pass


class MetaETL(SimpleNamespace, ParameterSetter):
    def __init__(self):
        super().__init__()
        """该数据遵循先设置更新后使用原则，并保证其依赖得到更新，能确保每次引用的最新的"""
        self.train_url = "http://download.tensorflow.org/data/iris_training.csv"
        self.test_url = "http://download.tensorflow.org/data/iris_test.csv"
        self.train_path = tf.keras.utils.get_file(self.train_url.split('/')[-1], self.train_url)
        self.test_path = tf.keras.utils.get_file(self.test_url.split('/')[-1], self.test_url)
        self.target = 'species'
        self.feature_names = ['sepallength', 'sepalwidth', 'petallength', 'petalwidth']
        self.categorical_feature_names = []
        self.numeric_feature_names = self.feature_names
        self.csv_column_names = self.feature_names + [self.target]
        self.species = ['setosa', 'versicolor', 'virginica']
        self.csv_column_defaults = [[0.0], [0.0], [0.0], [0.0], [0]]
        self.feature_defaults = [[0.0], [0.0], [0.0], [0.0]]

    def _update_dependency(self, key, value=None):
        if key in ['feature_names', 'target']:
            self.csv_column_names = self.feature_names + [self.target]
            self.numeric_feature_names = self.feature_names

        if key in ['train_url', 'test_url']:
            self.train_path = tf.keras.utils.get_file(self.train_url.split('/')[-1], self.train_url)
            self.test_path = tf.keras.utils.get_file(self.test_url.split('/')[-1], self.test_url)

    def gen_parse_csv_row(self):
        def parse_csv_row(csv_row)->Tuple[Dict[str, tf.Tensor], tf.Tensor]:
            columns = tf.decode_csv(csv_row, record_defaults=self.csv_column_defaults)
            features = dict(zip(self.csv_column_names, columns))
            target = features.pop(self.target)
            return features, target
        return parse_csv_row

    def process_features(self, features):
        pass

    def gen_csv_input_fn(self):
        def csv_input_fn(file_name_pattern, mode=tf.estimator.ModeKeys.EVAL, skip_header_lines=0,
                         num_epochs=None, batch_size=200)->tf.data.Dataset:
            shuffle = True if mode == tf.estimator.ModeKeys.TRAIN else False
            file_names = tf.matching_files(file_name_pattern)
            dataset = tf.data.TextLineDataset(file_names)
            dataset = dataset.skip(skip_header_lines)
            if shuffle:
                dataset = dataset.shuffle(buffer_size=int(2 * batch_size + 1))
            dataset = dataset.map(self.gen_parse_csv_row())
            dataset = dataset.batch(int(batch_size))
            dataset = dataset.repeat(int(num_epochs))
            return dataset
        return csv_input_fn

    def get_feature_columns(self)->Dict[str, Any]:
        numeric_columns = {feature_name: tf.feature_column.numeric_column(feature_name)
                           for feature_name in self.numeric_feature_names}
```

```
        feature_columns = {}
        if numeric_columns is not None:
            feature_columns.update(numeric_columns)
        return feature_columns

    def gen_csv_serving_input_fn(self):
        def csv_serving_input_fn():
            rows_string_tensor = tf.placeholder(dtype=tf.string, shape=[None], name='csv_rows')
            receiver_tensor = {'csv_rows': rows_string_tensor}
            row_columns = tf.expand_dims(rows_string_tensor, -1)
            columns = tf.decode_csv(row_columns, record_defaults=self.feature_defaults)
            features = dict(zip(self.feature_names, columns))
            return tf.estimator.export.ServingInputReceiver(features, receiver_tensor)
        return csv_serving_input_fn


# 得深入去了解Estimator要什么
def my_model(features, labels, mode, params):
    net = tf.feature_column.input_layer(features, params.feature_columns)
    for units in params.hidden_units:
        net = tf.layers.dense(net, units=units, activation=tf.nn.relu)
    logits = tf.layers.dense(net, params.n_classes, activation=None)
    predicted_classes = tf.argmax(logits, 1)
    # 在预测模式下要计算操作节点，主要是包括类别
    if mode == tf.estimator.ModeKeys.PREDICT:
        predictions = {'class_ids': predicted_classes[:, tf.newaxis], 'probabilities': tf.nn.softmax(logits), 'logits': logits}
        export_outputs = {'probabilities': tf.estimator.export.PredictOutput(tf.nn.softmax(logits)),
                          'class_ids': tf.estimator.export.PredictOutput(predicted_classes[:, tf.newaxis]),
                          tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY:
                              tf.estimator.export.PredictOutput(predictions)}
        return tf.estimator.EstimatorSpec(mode, predictions=predictions, export_outputs=export_outputs)

    # 计算评价量，损失函数和其它评价函数都属于评价准则，只是用途有所区别
    loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)      # 损失
    # Compute evaluation metrics.
    accuracy = tf.metrics.accuracy(labels=labels, predictions=predicted_classes, name='acc_op')
    average_loss = tf.reduce_mean(loss)
    metrics = {'accuracy': accuracy, 'average_loss': tf.metrics.mean(average_loss)}

    tf.summary.scalar('accuracy', accuracy[1])
    # 验证模式下
    if mode == tf.estimator.ModeKeys.EVAL:
        return tf.estimator.EstimatorSpec(mode, loss=loss, eval_metric_ops=metrics)
    # 训练模式下，要运行优化节点，计算损失.
    assert mode == tf.estimator.ModeKeys.TRAIN
    optimizer = tf.train.AdagradOptimizer(learning_rate=0.1)
    train_op = optimizer.minimize(loss, global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)


# 传递较大空间
class MetaMD(SimpleNamespace, ParameterSetter):
    def __init__(self, meta_etl):
        super().__init__()
        self.meta_etl = meta_etl
        self.train_size = 1200
        self.num_epochs = 100
        self.batch_size = 20
        #eval_after_sec = 1
        self.model_name = 'my_custom_iris'

    @property
    def max_steps(self):
        return (self.train_size / self.batch_size) * self.num_epochs

    @property
    def model_dir(self):
        return path.Path('trained_models/{}'.format(self.model_name)).makedirs_p()

    @property
    def export_dir(self):
        return path.Path(self.model_dir + "/export/estimate").makedirs_p()

    @property
    def hparams(self):
        return tf.contrib.training.HParams(
            feature_columns=list(self.meta_etl.get_feature_columns().values()),
            hidden_units=[10, 10],
            n_classes=3,
            learning_rate=0.01,
            max_steps=self.max_steps)
```

```
    @property
    def run_config(self):
        return tf.estimator.RunConfig(tf_random_seed=19830610, model_dir=self.model_dir)

    print("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")


def create_estimator(model, run_config, hparams):
    estimator = tf.estimator.Estimator(model_fn=model, params=hparams, config=run_config)
    print("Estimator Type: {}".format(type(estimator)))
    return estimator


# 这算中间状态量没必要公开设置口
def config_experiment(meta_etl, meta_md):
    cfg = SimpleNamespace()
    csv_input_fn = meta_etl.gen_csv_input_fn()
    csv_serving_input_fn = meta_etl.gen_csv_serving_input_fn()
    cfg.train_input_tr_fn = lambda: csv_input_fn(meta_etl.train_path, mode=tf.estimator.ModeKeys.TRAIN,
                                                 skip_header_lines=1, batch_size=meta_md.batch_size,
                                                 num_epochs=meta_md.num_epochs)
    cfg.eval_input_tr_fn = lambda: csv_input_fn(meta_etl.test_path, mode=tf.estimator.ModeKeys.EVAL, skip_header_lines=1,
                                       batch_size=meta_md.batch_size, num_epochs=meta_md.num_epochs)

    cfg.train_spec = tf.estimator.TrainSpec(input_fn=cfg.train_input_tr_fn, max_steps=meta_md.hparams.max_steps, hooks=None)
    cfg.eval_spec = tf.estimator.EvalSpec(input_fn=cfg.eval_input_tr_fn, steps=None, #throttle_secs=MetaMD.EVAL_AFTER_SEC
        exporters=[tf.estimator.LatestExporter(name="estimate", serving_input_receiver_fn=csv_serving_input_fn,as_text=True)])

    cfg.train_input_eval_fn = lambda: csv_input_fn(meta_etl.train_path, mode=tf.estimator.ModeKeys.EVAL, skip_header_lines=1,
                                                   batch_size=meta_md.batch_size, num_epochs=meta_md.num_epochs)
    cfg.eval_input_eval_fn = lambda: csv_input_fn(meta_etl.test_path, mode=tf.estimator.ModeKeys.EVAL, skip_header_lines=1,
                                                  batch_size=meta_md.batch_size, num_epochs=meta_md.num_epochs)
    return cfg


def run_experiment(EXP_CFG, estimator):
    time_start = datetime.utcnow()
    print("Experiment started at {}".format(time_start.strftime("%H:%M:%S")))
    print(".....................................")
    tf.estimator.train_and_evaluate(estimator=estimator, train_spec=EXP_CFG.train_spec, eval_spec=EXP_CFG.eval_spec)
    time_end = datetime.utcnow()
    print(".....................................")
    print("Experiment finished at {}".format(time_end.strftime("%H:%M:%S")))
    print("")
    time_elapsed = time_end - time_start
    print("Experiment elapsed time: {} seconds".format(time_elapsed.total_seconds()))
    ##################################################################################################################
    import math

    train_results = estimator.evaluate(input_fn=EXP_CFG.train_input_eval_fn, steps=1)
    #train_rmse = round(math.sqrt(train_results["average_loss"]), 5)
    print()
    print("##################################################################################################################")
    #print("# Train RMSE: {} - {}".format(train_rmse, train_results))
    print("##################################################################################################################")

    test_results = estimator.evaluate(input_fn=EXP_CFG.eval_input_eval_fn, steps=1)
    #test_rmse = round(math.sqrt(test_results["average_loss"]), 5)
    print()
    print("##################################################################################################################")
    #print("# Test RMSE: {} - {}".format(test_rmse, test_results))
    print("##################################################################################################################")

    predictions = estimator.predict(input_fn=EXP_CFG.eval_input_eval_fn)

    for it in range(10):
        it = next(predictions)
        print(it)


def export_model(estimator,csv_serving_input_fn, model_dir, sub_dir=''):
    export_dir = path.Path(model_dir + sub_dir).makedirs_p()
    estimator.export_savedmodel(export_dir_base=export_dir, serving_input_receiver_fn=csv_serving_input_fn, as_text=True)
    print(export_dir)
    return export_dir


def predict_input(export_dir):
    saved_model_dir = export_dir + "/" + os.listdir(path=export_dir)[-1]
    print(saved_model_dir)
```

```
    predictor_fn = tf.contrib.predictor.from_saved_model(export_dir=saved_model_dir, signature_def_key='class_ids')
    output = predictor_fn({'csv_rows': ["0.5,1,2,4"]})
    print(output)
    predictor_fn = tf.contrib.predictor.from_saved_model(export_dir=saved_model_dir)
    output = predictor_fn({'csv_rows': ["0.5,1,2,4"]})
    print(output)


def run_pipe(meta_etl, meta_md):

    estimator = create_estimator(my_model, meta_md.run_config, meta_md.hparams)
    ExperimentConfig = config_experiment(meta_etl, meta_md)
    run_experiment(ExperimentConfig, estimator)
    export_dir = export_model(estimator, meta_etl.gen_csv_serving_input_fn(), meta_md.model_dir, sub_dir='/my_export')
    predict_input(export_dir)
    print("================================================================================")


def search_parameter(params_path, params_manager, app):
    import pandas, time

    def config_it(it, parameter_manager):
        print(it)
        for fullname, value in it.items():
            group, name = fullname.split("__")
            print(group, name,value)
            parameter_manager[group].set_parameter(name, value)
    paras = pandas.read_csv(params_path)
    length = paras.shape[0]
    for idx in range(length):
        item = paras.iloc[idx].dropna().to_dict()
        item = {fullname.strip(): value for fullname, value in item.items()}
        config_it(item, params_manager)
        app()
        print("++++++++++++++++++++++++++++++++_______________________________++++++++++++++++++++++=")
        print("++++++++++++++++++++++++++++++++_______________________________++++++++++++++++++++++=")
        print("++++++++++++++++++++++++++++++++_______________________________++++++++++++++++++++++=")
        time.sleep(5)
        print('\n\n\n')


if __name__ == "__main__":

    tf.logging.set_verbosity(tf.logging.INFO)
    meta_etl = MetaETL()
    meta_md = MetaMD(meta_etl)
    param_path = r'doc/parameters.csv'
    param_manager = dict(MetaETL=meta_etl, MetaMD=meta_md)
    search_parameter(param_path, param_manager, lambda: run_pipe(meta_etl, meta_md))
```