TF Vision Example Project

This is a minimal example project to demonstrate how to use TF Model Garden's building blocks to implement a new vision project from scratch.

Below we use classification as an example. We will walk you through the process of creating a new projects leveraging existing components, such as tasks, data loaders, models, etc. You will get better understanding of these components by going through the process. You can also refer to the docstring of corresponding components to get more information.

Create Model

In example_model.py, we show how to create a new model. The `ExampleModel` is a subclass of `tf.keras.Model` that defines necessary parameters. Here, you need to have `input_specs` to specify the input shape and dimensions, and build layers within constructor:

```
class ExampleModel(tf.keras.Model):
  def __init__(
    self,
    num_classes: int,
    input_specs: tf.keras.layers.InputSpec = tf.keras.layers.InputSpec(
        shape=[None, None, None, 3]),
    **kwargs):
    # Build layers.
```

Given the `ExampleModel`, you can define a function that takes a model config as input and return an `ExampleModel` instance, similar as build_example_model. As a simple example, we define a single model. However, you can split the model implementation to individual components, such as backbones, decoders, heads, as what we do here. And then in `build_example_model` function, you can hook up these components together to obtain your full model.

Create Dataloader

A dataloader reads, decodes and parses the input data. We have created various dataloaders to handle standard input formats for classification, detection and segmentation. If you have non-standard or complex data, you may want to create your own dataloader. It contains a `Decoder` and a `Parser`.

- The Decoder decodes a TF Example record and returns a dictionary of decoded tensors:

```
class Decoder(decoder.Decoder):
      """A tf.Example decoder for classification task."""
      def __init__(self):
        """Initializes the decoder.

        The constructor defines the mapping between the field name and the
value
        from an input tf.Example. For example, we define two fields for image
bytes
        and labels. There is no limit on the number of fields to decode.
        """
```

```
        self._keys_to_features = {
            'image/encoded':
                tf.io.FixedLenFeature((), tf.string, default_value=''),
            'image/class/label':
                tf.io.FixedLenFeature((), tf.int64, default_value=-1)
        }
```

- The Parser parses the decoded tensors and performs pre-processing to the input data, such as image decoding, augmentation and resizing, etc. It should have `_parse_train_data` and `_parse_eval_data` functions, in which the processed images and labels are returned.

Create Config

Next you will define configs for your project. All configs are defined as `dataclass` objects, and can have default parameter values.

First, you will define your `ExampleDataConfig`. It inherits from `config_definitions.DataConfig` that already defines a few common fields, like `input_path`, `file_type`, `global_batch_size`, etc. You can add more fields in your own config as needed.

You can then define you model config `ExampleModel` that inherits from `hyperparams.Config`. Expose your own model parameters here.

You can then define your `Loss` and `Evaluation` configs.

Next, you will put all the above configs into an `ExampleTask` config. Here you list the configs for your data, model, loss, and evaluation, etc.

Finally, you can define a `tf_vision_example_experiment`, which creates a template for your experiments and fills with default parameters. These default parameter values can be overridden by a YAML file, like example_config_tpu.yaml. Also, make sure you give a unique name to your experiment template by the decorator:

```
@exp_factory.register_config_factory('tf_vision_example_experiment')
def tf_vision_example_experiment() -> cfg.ExperimentConfig:
  """Definition of a full example experiment."""
  # Create and return experiment template.
```

Create Task

A task is a class that encapsules the logic of loading data, building models, performing one-step training and validation, etc. It connects all components together and is called by the base Trainer.

You can create your own task by inheriting from base Task, or from one of the tasks we already defined, if most of the operations can be reused. An `ExampleTask` inheriting from ImageClassificationTask can be found here. We will go through each important components in the task in the following.

- `build_model`: you can instantiate a model you have defined above. It is also good practice to run forward pass with a dummy input to ensure layers within the model are properly initialized.
- `build_inputs`: here you can instantiate a Decoder object and a Parser object. They are used to create an `InputReader` that will generate a `tf.data.Dataset` object.

- `build_losses` : it takes groundtruth labels and model outputs as input, and computes the loss. It will be called in `train_step` and `validation_step` . You can also define different losses for training and validation, for example, `build_train_losses` and `build_validation_losses` . Just make sure they are called by the corresponding functions properly.
- `build_metrics` : here you can define your own metrics. It should return a list of `tf.keras.metrics.Metric` objects. You can create your own metric class by subclassing `tf.keras.metrics.Metric` .
- `train_step` and `validation_step` : they perform one-step training and validation. They take one batch of training/validation data, run forward pass, gather losses and update metrics. They assume the data format is consistency with that from the `Parser` output. `train_step` also contains backward pass to update model weights.

Import registry

To use your custom dataloaders, models, tasks, etc., you will need to register them properly. The recommended way is to have a single file with all relevant files imported, for example, registry_imports.py. You can see in this file we import all our custom components:

```
# pylint: disable=unused-import
from official.common import registry_imports
from official.vision.beta.projects.example import example_config
from official.vision.beta.projects.example import example_input
from official.vision.beta.projects.example import example_model
from official.vision.beta.projects.example import example_task
```

Training

You can create your own trainer by branching from our core trainer. Just make sure you import the registry like this:

```
from official.vision.beta.projects.example import registry_imports  # pylint:
disable=unused-import
```

You can run training locally for testing purpose:

```
# Assume you are under official/vision/projects.
python3 example/train.py \
  --experiment=tf_vision_example_experiment \
  --config_file=${PWD}/example/example_config_local.yaml \
  --mode=train \
  --model_dir=/tmp/tfvision_test/
```

It can also run on Google Cloud using Cloud TPU. Here is the instruction of using Cloud TPU and here is a more detailed tutorial of training a ResNet-RS model. Following the instructions to set up Cloud TPU and launch training by:

```
EXP_TYPE=tf_vision_example_experiment  # This should match the registered name of
your experiment template.
```

```
EXP_NAME=exp_001  # You can give any name to the experiment.
TPU_NAME=experiment01
# Now launch the experiment.
python3 example/train.py \
  --experiment=$EXP_TYPE \
  --mode=train \
  --tpu=$TPU_NAME \
  --model_dir=/tmp/tfvision_test/
  --
config_file=third_party/tensorflow_models/official/vision/examples/starter/example_con
```