
Tensor2Robot

This repository contains distributed machine learning and reinforcement learning infrastructure.

It is used internally at Alphabet, and open-sourced with the intention of making research at Robotics @ Google more reproducible for the broader robotics and computer vision communities.

Projects and Publications Using Tensor2Robot

- QT-Opt
- Grasp2Vec
- Watch, Try, Learn
- BC-Z

Features

Tensor2Robot (T2R) is a library for training, evaluation, and inference of large-scale deep neural networks, tailored specifically for neural networks relating to robotic perception and control. It is based on the TensorFlow deep learning framework.

A common task in robotics research involves adding a new sensor modality or new label tensor to a neural network graph. This involves 1) changing what data is saved, 2) changing data pipeline code to read in new modalities at training time 3) adding a new `tf.placeholder` to handle the new input modality at test time. The main feature of Tensor2Robot is the automatic generation of TensorFlow code for steps 2 and 3. Tensor2Robot can automatically generate placeholders for a model to match its inputs, or alternatively exports a `SavedModel` that can be used with a `TfExportedSavedModelPolicy` so that the original graph does not have to be re-constructed.

Another common task encountered in ML involves cropping / transforming input modalities, such as jpeg-decoding and applying random image distortions at training time. The `Preprocessor` class declares its own input features and labels, and is expected to output shapes compatible with the input features and labels of the model. Example preprocessors can be found in `preprocessors`.

Design Decisions

- Scalability: This codebase is designed for training large-scale, real-world robotic perception models with algorithms that do not require a tight perception-action-learning loop (supervised learning, off-policy reinforcement learning of large computer vision models). An example setup might involve multiple GPUs pulling data asynchronously from a replay buffer and training with

an off-policy RL algorithm, while data collection agents periodically update their checkpoints and push experiences to the same replay buffer. This can also be run on a single workstation for smaller-scale experiments.

- T2R is *NOT* a general-purpose reinforcement learning library. Due to the sizes of models we work with (e.g. grasping from vision) Inference is assumed to be within 1-10Hz and without real-time guarantees, and training is assumed to be distributed by default. If you are doing reinforcement learning with small networks (e.g. two-layer perceptron) with on-policy RL (e.g. PPO), or require hard real-time guarantees, this is probably not the right codebase to use. We recommend using TF-Agents or Dopamine for those use cases.
- Minimize boilerplate: Tensor2Robot Models auto-generate their own data input pipelines and provide sensible defaults for optimizers, common architectures (actors, critics), and train/eval scaffolding. Models automatically work with both GPUs and TPUs (via [TPUEstimator](#)), parsing bmp/gif/jpeg/png-encoded images.
- gin-configurable: Gin-Config is used to configure models, policies, and other experiment hyper-parameters.

Quickstart

Requirements: Python 3.

```
1 git clone https://github.com/google/tensor2robot
2 # Optional: Create a virtualenv
3 python3 -m venv ~/venv
4 source ~/venv/bin/activate
5 pip install -r tensor2robot/requirements.txt
6 python -m tensor2robot.research.pose_env.pose_env_test
7
8 # Install protoc and compile the protobufs.
9 pip install protobuf
10 cd tensor2robot/proto
11 protoc -I=. --python_out=`pwd` tensor2robot/t2r.proto
12 python -m tensor2robot.research.pose_env.pose_env_models_test
```

T2RModel

To use Tensor2Robot, a user defines a [T2RModel](#) object that define their input requirements by specifications - one for their features (`feature_spec`) and one for their labels (`label_spec`):

These specifications define all required and optional tensors in order to call the `model_fn`. An input pipeline parameterized with the model's input pipeline will ensure that all required specifications are

fulfilled. **Note:** we always omit the batch dimension and only specify the shape of a single element.

At training time, the T2RModel provides `model_train_fn` or `model_eval_fn` as the `model_fn` argument `tf.estimator.Estimator` class. Both `model_train_fn` and `model_eval_fn` are defined with respect to the features, labels, and outputs of `inference_network_fn`, which presumably implements the shared portions of the train/eval graphs.

```
1 class MyModel(T2RModel):
2     def get_feature_specification(self, mode):
3         spec = tensorspec_utils.TensorSpecStruct()
4         spec['state'] = ExtendedTensorSpec(
5             shape=(8,128), dtype=tf.float32, name='s')
6     def get_label_specification(self, mode):
7         spec = tensorspec_utils.TensorSpecStruct()
8         spec['action'] = ExtendedTensorSpec(shape=(8), dtype=tf.float32,
9             name='a')
9     def inference_network_fn(self,
10         features: tensorspec_utils.TensorSpecStruct,
11         labels: Optional[tensorspec_utils.
12             TensorSpecStruct],
13         mode: tf.estimator.ModeKeys,
14         config: RunConfigType = None,
15         params: ParamsType = None) -> DictOrSpec:
16         inference_outputs = {}
17         inference_outputs['predictions'] = layers.fully_connected(features.
18             state, 8)
19         return inference_outputs
20     def model_train_fn(self,
21         features: tensorspec_utils.TensorSpecStruct,
22         labels: tensorspec_utils.TensorSpecStruct,
23         inference_outputs: DictOrSpec,
24         mode: tf.estimator.ModeKeys,
25         config: RunConfigType = None,
26         params: ParamsType = None) -> ModelTrainOutputType
27         :
28         """See base class."""
29         del features, config
30         loss = tf.losses.mean_squared_error(
31             labels.action, inference_outputs['predictions'])
32         return loss
```

Note how the **key** on the **left hand side** has a value for **name** that is different from the one on the **right hand side** within the `ExtendedTensorSpec`. The key on the left is used within the `model_fn` to access the loaded tensor whereas the name is used when creating the `parse_tf_example_fn` or `numpy_feed_dict`. We ensure that the name is unique within the whole spec, unless the specs match, otherwise we cannot guarantee the mapping functionality.

Benefits of Inheriting a T2RModel

- Self-contained input specifications for features and labels.
- Auto-generated `tf.data.Dataset` pipelines for `tf.train.Examples` and `tf.train.SequenceExamples`.
- For policy inference, T2RModels can generate placeholders or export `SavedModels` that are hermetic and can be used with `ExportSavedModelPolicy`.
- Automatic construction of `model_fn` for Estimator for training and evaluation graphs that share a single `inference_network_fn`.
- It is possible to compose multiple models' `inference_network_fn` and `model_train_fn` together under a single model. This abstraction allows us to implement generic Meta-Learning models (e.g. MAML) that call their sub-model's `model_train_fn`.
- Automatic support for distributed training on GPUs and TPUs.

Policies and Placeholders

For performance reasons, policy inference is done by a vanilla `session.run()` or a `predict_fn` call on the output of a model, instead of `Estimator.predict`. `tensorspec_utils.make_placeholders` automatically creates placeholders from a spec structure which can be used in combination with a matching hierarchy of numpy inputs to create a `feed_dict`.

```
1 # batch_size = -1 -> No batch size will be prepended to the spec.
2 # batch_size = None -> We will prepend None, have a variable batch_size
3 # batch_size > 0 -> We will have a fixed batch_size.
4 placeholders = tensorspec_utils.make_placeholders(hierarchical_spec,
5                                                    batch_size=None)
6 feed_dict = inference_model.MakeFeedDict(placeholders, numpy_inputs)
7 # This can be passed to a sess.run function to evaluate the model.
```

If you use `TfExportedSavedModelPolicy`, note that your T2RModel should not query the static batch shape (`x.shape[0]`) in the graph. This is because placeholder generation creates inputs with unknown batch shape `None`, causing static shape retrieval to fail. Instead, use `tf.shape(x)[0]` to access batch shapes dynamically.

Working with Tensor Specifications

Specifications can be a hierarchical data structure of either

- dictionaries (dict),

-
- tuples (tuple),
 - lists (list), or
 - TensorSpecStruct (preferred).

The leaf elements have to be of type TensorSpec or ExtendedTensorSpec (preferred). In the following, we will present some examples using ExtendedTensorSec and TensorSpecStruct to illustrate the different usecases.

We use tensorspec_utils.TensorSpecStruct() for specifying specs since this data structure is mutable, provides attribute (dot) access and item iteration. Further, we can use pytype to ensure compile time type checking. This data structure is both hierarchical and flat: The dictionary interface using .items() is flat representing hierarchical data with paths, as shown later on. However, we maintain a hierarchical interface using attribute access, also shown later on. Therefore, we can use this data structure in order to create and alter hierarchical specifications which work with both TPUEstimator and Estimator since both apis operate on the dictionary view.

Hierarchical example

Creating a hierarchical spec from spec using tensorspec_utils.copy_tensorspec.

```
1 simple_spec = tensorspec_utils.TensorSpecStruct()
2 simple_spec['state'] = ExtendedTensorSpec(
3     shape=(8,128), dtype=tf.float32, name='s')
4 simple_spec['action'] = ExtendedTensorSpec(shape=(8), dtype=tf.float32,
5     name='a')
6
7 hierarchical_spec = tensorspec_utils.TensorSpecStruct()
8 hierarchical_spec.train = tensorspec_utils.copy_tensorspec(simple_spec,
9     prefix='train')
```

Note, we use attribute access to define the train hierarchy. This will copy all our specs from simple_spec internally to

```
1 # 'train/{}' -> 'train/state' == simple_spec.state and
2 # 'train/action' == simple_spec.action
```

We forbid the following pattern:

```
1 hierarchical_spec.train = tensorspec_utils.TensorSpecStruct()
```

and encourage the user to use this pattern instead.

```
1 train = tensorspec_utils.TensorSpecStruct()
2 train.stuff = ExtendedTensorSpec(...)
3 hierarchical_spec.train = train
```

```

4 # or
5 hierarchical_spec['train/stuff'] = ExtendedTensorSpec(...)

```

```

1 # All of the following statements are True.
2 hierarchical_spec.train.state == simple_spec.state
3 hierarchical_spec.train.action == simple_spec.action
4 hierarchical_spec.keys() == ['train/state', 'train/action']
5 hierarchical_spec.train.keys() == ['state', 'action']

```

Now we want to use the same spec another time for our input.

```

1 hierarchical_spec.val = tensorspec_utils.copy_tensorspec( simple_spec,
2   prefix='val')
3
4 # All of the following statements are True.
5 hierarchical_spec.keys() == ['train/state', 'train/action', 'val/state'
6   'val/action']
7 hierarchical_spec.train.keys() == ['state', 'action']
8 hierarchical_spec.train.state.name == 'train/s'
9 hierarchical_spec.val.keys() == ['state', 'action']
10 hierarchical_spec.val.state.name == 'val/s'

```

Manually extending/creating a hierarchical spec from an existing simple spec is also possible. TensorSpec is an immutable data structure therefore the recommend way to alter a spec is:

```

1 hierarchical.train.state = ExtendedTensorSpec.from_spec(
2   hierarchical.train.state, ...PARAMETERS TO OVERWRITE)

```

A different way of changing a hierarchical spec would be:

```

1 for key, value in simple_spec.items():
2   hierarchical['test'/' + key] = ExtendedTensorSpec.from_spec(
3     value, name='something_random'/' + value.name)
4 # hierarchical_spec.keys() == ['train/state', 'train/action', 'val/
5   'state,
6   'val/'action, 'test/'state, 'test/'action]
7 # hierarchical_spec.train.keys() == ['state', 'action']
8 # hierarchical_spec.val.keys() == ['state', 'action']
9 # hierarchical_spec.test.keys() == ['state', 'action']
10 # hierarchical_spec.test.state.name == 'something_random/'s

```

Sequential Inputs

Tensor2Robot can parse both `tf.train.Example` and `tf.train.SequenceExample` protos (useful for training recurrent models like LSTMs). To declare a model whose data is parsed from SequenceExamples, set `is_sequence=True`.

```
1 spec['state'] = ExtendedTensorSpec(  
2     shape=(8,128), dtype=tf.float32, name='s', is_sequence=True)
```

This will result in a parsed tensor of shape (b, ?, 8, 128) where b is the batch size and the second dimension is the unknown sequence length (only known at run-time). Note that if `is_sequence=True` for any `ExtendedTensorSpec` in the `TensorSpecStruct`, the proto will be assumed to be a `SequenceExample` (and non-sequential Tensors will be assumed to reside in `example.context`).

Flattening hierarchical specification structures

Any valid spec structure can be flattened into a `tensorspec_utils.TensorSpecStruct`. In the following we show different hierarchical data structures and the effect of `flatten_spec_structure`.

```
1 flat_hierarchy = tensorspec_utils.flatten_spec_structure(hierarchical)
```

Note, `tensorspec_utils.TensorSpecStruct` will have flat dictionary access. We can print-/access/change all elements of our spec by iterating over the items.

```
1 for key, value in flat_hierarchy.items():  
2     print('path: {}, spec: {}'.format(key, value))  
3 # This will print:  
4 # path: train/state, spec: ExtendedTensorSpec(  
5 #     shape=(8, 128), dtype=tf.float32, name='train/s')  
6 # path: train/action, spec: ExtendedTensorSpec(  
7 #     shape=(8), dtype=tf.float32, name='train/a')  
8 # path: val/state, spec: ExtendedTensorSpec(  
9 #     shape=(8, 128), dtype=tf.float32, name='val/s')  
10 # path: val/action, spec: ExtendedTensorSpec(  
11 #     shape=(8), dtype=tf.float32, name='val/a')  
12 # path: test/state, spec: ExtendedTensorSpec(  
13 #     shape=(8, 128), dtype=tf.float32, name='something_random/s')  
14 # path: test/action, spec: ExtendedTensorSpec(  
15 #     shape=(8), dtype=tf.float32, name='something_random/a')  
16 # This data structure still maintains the hierarchical user interface.  
17 train = flat_hierarchy.train  
18 for key, value in flat_hierarchy.items():  
19     print('path: {}, spec: {}'.format(key, value))  
20 # This will print:  
21 # path: state, spec: ExtendedTensorSpec(  
22 #     shape=(8, 128), dtype=tf.float32, name='train/s')  
23 # path: action, spec: ExtendedTensorSpec(  
24 #     shape=(8), dtype=tf.float32, name='train/a')
```

Note, the path has changed, but the name is still from the hierarchy. This is an important distinction. The model could access the data in a different manner but the same “name” is used to access

`tf.Examples` of `feed_dicts` in order to feed the tensors.

An alternative hierarchical spec using `namedtuples`:

```
1 Hierarchy = namedtuple('Hierarchy', ['train', 'val'])
2 Sample = namedtuple('Sample', ['state', 'action'])
3 hierarchy = Hierarchy(
4     train=Sample(
5         state=ExtendedTensorSpec(shape=(8, 128), dtype=tf.float32, name='
        train/s'),
6         action=ExtendedTensorSpec(shape=(8), dtype=tf.float32, name='train/
        a'),
7     ),
8     eval=Sample(
9         state=ExtendedTensorSpec(shape=(8, 128), dtype=tf.float32, name='
        val/s'),
10        action=ExtendedTensorSpec(shape=(8), dtype=tf.float32, name='val/a'
        ),
11    )
12 )
13 flat_hierarchy = tensorspec_utils.flatten_spec_structure(hierarchy)
14
15 for key, value in flat_hierarchy.items():
16     print('path: {}, spec: {}'.format(key, value))
17 # This will print:
18 # path: train/state, spec: ExtendedTensorSpec(
19 #     shape=(8, 128), dtype=tf.float32, name='train/s')
20 # path: train/action, spec: ExtendedTensorSpec(
21 #     shape=(8), dtype=tf.float32, name='train/a')
22 # path: val/state, spec: ExtendedTensorSpec(
23 #     shape=(8, 128), dtype=tf.float32, name='val/s')
24 # path: val/action, spec: ExtendedTensorSpec(
25 #     shape=(8), dtype=tf.float32, name='val/a')
```

Note, `hierarchy` (namedtuple) is immutable whereas `flat_hierarchy` is a mutable instance of `TensorSpecStruct`.

Validate and flatten or pack

`tensorspec_utils.validate_and_flatten` and `tensorspec_utils.validate_and_pack` allow to verify that an existing, e.g. loaded spec data structure filled with tensors fulfills our expected spec structure and is flattened or packed into a hierarchical structure.

Disclaimer

This is not an official Google product. External support not guaranteed. The API may change subject to Alphabet needs. File a GitHub issue if you have questions.