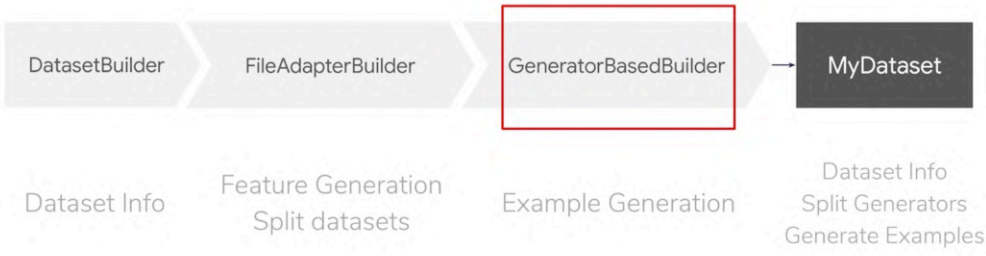
Under the hood

Under the hood

Next, datasets also need to provide information on how the dataset can be split into sets such as training, validation, and testing, along with describing the features that we'll use. You do all of this in one of the subclasses of dataset builder called file adapter builder

Under the hood

Finally, generator base builder simplifies all of this as its subclasses, implement all of the things that we've discussed just now as well as generating examples in the dataset from source data. You're going to be using this a lot.



Getting started

```
# Clone the tensorflow-datasets repository
git clone https://github.com/tensorflow/datasets.git

tensorflow_datasets/scripts/create_new_dataset.py \
  --dataset my_dataset \
  --type image  # text, audio, translation,...
```

Before actually building your dataset, you'll first need to get your hands on a template. Fortunately, there are open source scripts to help with this. To get started, you should clone the tfds repository at github.com/tensorflow/datasets.git.

Getting started

```
# Clone the tensorflow-datasets repository
git clone https://github.com/tensorflow/datasets.git

tensorflow_datasets/scripts/create_new_dataset.py \
  --dataset my_dataset \
  --type image  # text, audio, translation,...
```

You will then run the create new dataset script while you're working inside the repository. This will come in quite handy because it generates all of the required Python files. You'll next then look into what needs to be done with this default template to tailor it to your datasets needs.

```
class MyDataset(tfds.core.GeneratorBasedBuilder):
  VERSION = tfds.core.Version('0.1.0')
  def _info(self):
    return tfds.core.DatasetInfo(builder=self, description=..., features=...,
                                 supervised_keys=..., urls=..., citation=...)

  def _split_generators(self, dl_manager):

  def _generate_examples(self):
    yield 'key', {}
```

Most dataset subclass tfds.core.generatorbasedbuilder, which is a subclass of dataset builder that simplifies defining a dataset. This works really well for datasets that can be generated on a single machine. Its subclasses implement three methods, info, split generators, and generate examples.

The point of the info function is to build the dataset info object to describe the dataset.

```python
class MyDataset(tfds.core.GeneratorBasedBuilder):
  VERSION = tfds.core.Version('0.1.0')
  def _info(self):
    return tfds.core.DatasetInfo(builder=self, description=..., features=...,
                                 supervised_keys=..., urls=..., citation=...)

  def _split_generators(self, dl_manager):

  def _generate_examples(self):
    yield 'key', {}
```

Split generator sets up your datasets by downloading the source data and preparing it by defining the dataset splits.


```python
class MyDataset(tfds.core.GeneratorBasedBuilder):
  VERSION = tfds.core.Version('0.1.0')
  def _info(self):
    return tfds.core.DatasetInfo(builder=self, description=..., features=...,
                                 supervised_keys=..., urls=..., citation=...)

  def _split_generators(self, dl_manager):

  def _generate_examples(self):
    yield 'key', {}
```

Split generator sets up your datasets by downloading the source data and preparing it by defining the dataset splits.


```python
def _info(self):
  return tfds.core.DatasetInfo(
    builder=self,description=("INSERT DESCRIPTION HERE"),
    features=tfds.features.FeaturesDict({
          "description": tfds.features.Text(),
          "image": tfds.features.Image(),
          "label": tfds.features.ClassLabel(num_classes=5),
    }),

     supervised_keys=("image", "label"),
     urls=["https://dataset-homepage.org"],
     citation=r"""@article{my-awesome-dataset-2020,
                  ...
                  author = {Smith, John},"}""")
```

The info method is used to describe all the details of a dataset. Description, features used, and they're supervised keys, if nay. You also put citations here. But most importantly, you need to specify the place where you can find this data set in a URL.

When it comes to features, though, each one needs to be described accurately. For this, feature connectors come into play by documenting each element, providing shape and type checks, and abstracting away serialization to and from disk. There are many feature types already defined, but you can also add a new one.

```
def _info(self):
  return tfds.core.DatasetInfo(
      builder=self,
      description="A large set of images of horses and humans.",
      features=tfds.features.FeaturesDict({
          "image": tfds.features.Image(shape=_IMAGE_SHAPE),
          "label": tfds.features.ClassLabel(
              names=["horses", "humans"]),
      }),
      supervised_keys=("image", "label"),
      urls=["http://laurencemoroney.com/horses-or-humans-dataset"],
      citation=_CITATION
      )
```

So while that was all placeholders, here's an example of a real info function from the horses-or-humans-dataset that was created for the TensorFlow and practice specialization.

Most datasets need to download data from the web. All downloads and extractions must go through the download manager object in tfds. This will be responsible for managing the download and retrieval of files as well as caching. All the downloaded files will be cached under download directory.

Download manager currently supports extracting zip, GS, and tar files. So for example, one can both download and extract URLs with download and extract, or they can even be called separately by first calling the download method and then wrapping it inside the extract.

However, for source data that cannot automatically be downloaded, for example, if the user needs to authorize via login, you will need to manually download the Source data and place it in the manual directory. This can then be accessed with the DL manager object, and by default, it's set to point to squiggle slash tensorflow datasets slash manual slash my dataset.

```
def _split_generators(self, dl_manager):
  # Equivalent to dl_manager.extract(dl_manager.download(urls))
  dl_paths = dl_manager.download_and_extract({
      'foo': 'https://example.com/foo.zip',
      'bar': 'https://example.com/bar.zip',
  })
  dl_paths['foo'], dl_paths['bar']
```

For source data that cannot be automatically downloaded, manually download the source data and place it in the manual_dir.

This can be accessed with the dl_manager object, which is set to ~/tensorflow_datasets/manual/my_dataset by default

```
def _split_generators(self, dl_manager):
  """Returns SplitGenerators."""
  # TODO(my_dataset): Downloads the data and defines the splits
  # dl_manager is a tfds.download.DownloadManager that can be used to
  # download and extract URLs
  return [
      tfds.core.SplitGenerator(
          name=tfds.Split.TRAIN,
          # These kwargs will be passed to _generate_examples
          gen_kwargs={},
      ),
  ]
```

If the dataset comes with predefined splits, for example, MNIST has train and test splits, you can keep those splits in the dataset builder. If this is your data and you can define your splits, then we suggest going for a data separation of about 80% for training, 10% for validation, and 10% for testing. You're always free to choose other sub splits, of course, through tfds dot split dot subsplit.

You can choose a custom subsplit through using tfds.Split.subsplit.

So this method ends with returning a split generator describing how a split should be generated. The generator's keyword arguments take in parameters that will be used by generate examples

```
def _split_generators(self, dl_manager):
  return [tfds.core.SplitGenerator(
              name=tfds.Split.TRAIN,
              gen_kwargs={
                  "dir_path": os.path.join(extracted_path, "train"),
                  "labels": os.path.join(extracted_path, "train_labels.csv")}),
          tfds.core.SplitGenerator(
              name=tfds.Split.TEST,
              gen_kwargs={
                  "dir_path": os.path.join(extracted_path, "test"),
                  "labels": os.path.join(extracted_path, "test_labels.csv")})
        ]
```

Here's an example of what it looks like in the MNIST dataset. You can see here that the split generator returns a list of two split generators. The first is the training data set along with its labels. And the second is the test data set along with its labels.

```
    def _split_generators(self, dl_manager):
        train_path, test_path = dl_manager.download([_TRAIN_URL, _TEST_URL])

        return [
            tfds.core.SplitGenerator(
                name=tfds.Split.TRAIN,
                num_shards=10,
                gen_kwargs={
                    "archive": dl_manager.iter_archive(train_path)
                }),
            tfds.core.SplitGenerator(
                name=tfds.Split.TEST,
                num_shards=10,
                gen_kwargs={
                    "archive": dl_manager.iter_archive(test_path)
                }),
    ]
```

Another example here, this time used for the horses-or-humans-dataset, where, again, we return a training a test split as two split generators.

In this case, the args show the archive where the data was downloaded to.

```
def _generate_examples(self):
  """Yields examples."""
  # TODO(my_dataset): Yields (key, example) tuples from the dataset
  yield 'key', {}
```

Finally, the third method that needs to be implemented is generate examples. This will use the information passed on by the split generator to generate the examples for each split from the source data. This method will typically read source dataset artifacts, for example, a CSV file, and it will yield tuples comprising of keys representing the names of feachers and the features themselves as values. All of these features should comply with the feature specification in dataset info.

```python
def _generate_examples(self, archive):
  for fname, fobj in archive:
    res = _NAME_RE.match(fname)
    if not res:  # if anything other than .png; skip
      continue
    label = res.group(1).lower()
    record = {
        "image": fobj,
        "label": label,
    }
    yield fname, record
```

Here's an example of generate_examples in action, where it simply gets the images in the archive and returns a record for the file name with the image being a representation of the image object and a label derived from the file name using a regular expression. You can see the source file for more details.

Now, if you were wondering how example generation works in the builder, it will be something like this. For the train split with the genkwargs defined above, generate_examples will be called as follows by specifying the directory where the images are located and also the path to the label's CSV file.

```python
builder._generate_examples(
    images_dir_path="{extracted_path}/train",
    labels="{extracted_path}/train_labels.csv",
)
```

File access

- Use `tf.io.gfile` or `tf.python_io` to support Cloud Storage systems
- **Avoid** using **Python built-ins**
  - e.g., `open`, `os.rename`, `gzip`

Some things to consider a file access including extra dependencies as well as dealing with corrupted or inconsistent data. For all file system access, use tf.io.gfile or other tensorflow file APIs, for example, tf.python.io. This is done to support Cloud storage systems. Try to avoid using python built-ins for file operations such as open, os.rename and gzip, etcetera.

Extra dependencies

Some datasets require additional python dependencies during data generation. For example, the SVHn dataset uses scipy to load some data.

To keep the tensorflow datasets package small and allow users to only install the additional dependencies that they need, use tfds.core.lazy_imports.

To use lazy import, you need to add an entry for your dataset into dataset extras in the setup.pi file for your package.

This makes it so that users can only need to call PIP install tensorflow datasets to install the extra dependencies.

You could even import it yourself in the dataset.pi file by manually placing the lazy imports wherever necessary.

- Use `tfds.core.lazy_imports`
- Add a lazy import with an entry in DATASET_EXTRAS
- Install with

```
pip install tensorflow-datasets[<dataset-name>]
```

For more details on setup.py, Please look at the setup.py file on GitHub.

https://github.com/tensorflow/datasets/blob/master/setup.py

In particular, look at the DATASET_EXTRAS section, where you would declare any extras that you might need in your dataset. For example, cats v. dogs needs the matplotlib library.

Problems in data

- Corrupted data
  - e.g., invalid image formats
- Inconsistent data
  - **Solution**: Mark dataset as unstable by adding

    A class constant - UNSTABLE in DatasetBuilder

Once you're done collecting data, you might not always have a perfect and complete data set as you're bound to come across some corrupt data. Now, case for this might be you're collecting images where some invalid image formats are encountered. Now these examples should be skipped or ignored. But you should do try to leave a note in the data set description of how many samples were dropped and why.

Some datasets provide a set of URLs for individual records or features. So for example, URLs to various images around the web and these may or may not exist anymore. These datasets become very difficult to version properly because the source data can be unstable, URLs do come and go. If the dataset is inherently unstable, that is if multiple runs may not yield the same data over time, you should mark the dataset as unstable. You can do this in practice by adding a class constant to the data set builder defining it

When to use configurations

Heavy	Light
• Specifies how data needs to be written to the disk • Different `DatasetInfo` setups • When changing access for download data • Use `tfds.core.BuilderConfigs` to configure data generation	• Deals with runtime preprocessing • tf.data input pipelines • Perform additional transformations

Some datasets may have variance that should be exposed or options for how the data is pre-processed. Based on how you want your data set to behave, you can have two configurations, and these are heavy and light.

The heavy configuration deals with how you want your data to be written to disk. For example, this may come in handy if you have a data set with versions having dataset infos differing from one another. Similarly, this would be the case if you want to change the access for data that's being downloaded.

Another situation where you may come across the need for testing a heavy configuration is with text data. This is when you want multiple builder configs to configure data generation concerning texting coders and vocabulary that can affect token IDs that are written to disk.

Loading a dataset with a custom configuration

```
# See the built-in configs
configs = tfds.text.IMDBReviews.builder_configs

>>> print(configs.keys())
dict_keys(['plain_text', 'bytes', 'subwords8k', 'subwords32k'])

# Address a built-in config with tfds.builder
imdb = tfds.builder("imdb_reviews/bytes")
# or when constructing the builder directly
imdb = tfds.text.IMDBReviews(config="bytes")
```

Publishing your own dataset

Add an import for registration

All sub-classes of tfds.core.DatasetBuilder are automatically registered when a module is imported so that they can be accessed through tfds.builder and tfds.load. If you're contributing the dataset to tensorflow datasets at the module imports to one of the subdirectories in it.py file, the directories may be image, audio, video, text, translate or structured.

```
# In the 'image' subdirectory of tensorflow/datasets
from tensorflow_datasets.image.cifar import Cifar10
from tensorflow_datasets.image.cifar import Cifar100
...
from tensorflow_datasets.image.my_image_dataset import MyImageDataset

# In the 'text' subdirectory of tensorflow/datasets
from tensorflow_datasets.text.cnn_dailymail import CnnDailymail
...
from tensorflow_datasets.image.my_text_dataset import MyTextDataset
```

Download and prepare

- Create file

  `tensorflow_datasets/url_checksums/my_new_dataset.txt`

- Run `download_and_prepare` locally to ensure that data generation works

```
# default data_dir is ~/tensorflow_datasets
python -m tensorflow_datasets.scripts.download_and_prepare \
  --register_checksums \
  --datasets=my_new_dataset
```

Next, you'll proceed with downloading and preparing the dataset locally to ensure that the data generation works as it should. For this, you first need to create a text file that has the same name as the dataset file in the tensorflow-dataset//url_checksums folder. Then you'll manually run the download and prepare script to register your dataset with the checksums. On the first download, the download manager will automatically add the sizes and checksums for all downloaded URLs to that file. This ensures that on subsequent data generation, the downloaded files are as expected. If you're contributing the dataset to tensorflow datasets, add a checksum's file for your dataset. Note that the dash dash register checksums flag must only be used while in development, a JSON file will be generated and you have to include this file in your pull request.

Double-check citations

It's essential that dataset info.citation includes proper citations for the dataset. It's hard and vital work contributing a dataset to the community and we want to make it easy for dataset users to cite the work. You can use the Bib Tex Online Editor to create a custom bib text entry.

```
@ARTICLE {,
    author  = "John",
    title   = "Classification of ...",
    journal = "International Journal of ...",
    year    = "2019"
}
```

https://truben.no/latex/bibtex/

Test data

- Put test data under your dataset's directory
- Make sure there are no duplicates in splits
- No copyrighted material

The test data should be put in testing/test data/fake examples/ under your datasets directory, and it should mimic the source dataset artifacts as downloaded and extracted. It can be created manually or automatically with a script that you'll see next.

Also make sure to use different data in your test data splits as the test will fail if your dataset splits overlap.

Finally, ensure that the test data does not contain any copyrighted material.

```python
from tensorflow_datasets import my_dataset
import tensorflow_datasets.testing as tfds_test

class MyDatasetTest(tfds_test.DatasetBuilderTestCase):
  DATASET_CLASS = my_dataset.MyDataset
  SPLITS = {  # Expected number of examples on each split from fake example.
      "train": 3,
      "test": 3,
  }
  # If dataset `download_and_extract`s more than one resource:
  DL_EXTRACT_RESULT = {
      "name1": "path/to/file1",  # Relative to fake_examples/my_dataset dir.
      "name2": "file2",
  }
  if __name__ == "__main__":
    tfds_test.test_main()
```

Tfds.testing.DatasetBuilderTestCase, is a base test case to exercise a dataset fully. It uses fake examples as test data that mimic the structure of the source dataset.

Here's an example of how you can write a test case before you publish a dataset. This will ensure that all of your data is downloaded and extracted correctly. First of all, you need to pass the reference to the datasets class to dataset class variable. Next, you'll specify the number of samples you wish to test in each available split. Finally, you end the test case by providing the dictionary of paths to your test data files.

Final touches

- Make sure coding style is compliant with
  - PEP8
  - Google's Python Style Guide
- Add release notes
- Send for review

In the end, before publishing your dataset, here's a checklist that you need to keep in mind. First, follow the PEP8 Python Style Guide but there's one exception that you need to know, which is concerning spaces. Tensorflow expects you to use two spaces instead of four. Also, please conform to Google's Python coding style. Next, add the release notes with all of the updated changes, if any. Lastly, you can send the pull request to tensorflow datasets and wait for the team to review it.