1. Why would you want to use the Data API?

APIs are needed to bring applications together in order to perform a designed function built around sharing data and executing pre-defined processes. They work as the middle man, allowing developers to build new programmatic interactions between the various applications people and businesses use on a daily basis.

Perhaps you started off by Googling “APIs” and quickly found yourself drowning in acronyms, jargon, and unhelpful exposition. Do I need to take a REST or should I be using SOAP to clean off my data? What is a WSDL, and why do I need to put one in my SOAP? Is an RPC like an RPG and how do I level up? Confusion is a common experience for the growing body of people who are suddenly being asked to know more about tools that have historically been restricted to the realm of the “software engineer”.

This article will help you understand what API’s are without the pervasive (and unhelpful) jargon surrounding them. I will also include a code sample that you can run, right here in your browser, to access one of my favorite publicly available APIs — the Reddit API. Despite the code examples, I am writing this article with non-software engineers in mind. This article is for the sales analysts, executives, HR specialists, and others with no programming background, but who have a growing need to interact with and understand the growing world of Web APIs.

2.What are the benefits of splitting a large dataset into multiple files?

In this article we explore the array of benefits which one can derive by splitting an Ms Access Database

One of the most useful features provided by Access to its users is the feature that allows splitting the databases. Splitting a database involves creating different files, one in the backend, with relational data and another in the frontend, with interface objects. There are a lot of benefits that you can achieve by splitting your Access databases, a few of those benefits are described below.

1. Multiple Users can Access Data Simultaneously

When the data in a database is split, in frontend and backend, it can be easily supplied to multiple users sharing a network. If the backend is stored on file server and frontend on workstations, multiple users will be able to access data and make changes wherever necessary. If any of the user makes any kind of change in the databases, all authorized users will instantly get updates regarding the changes made.

2. Provides Better Protection

By splitting your databases, you can make them more secure, if all your database designs are stored in the backend, no one from the front end will be able to make any kind of changes in the tables. And those accessing the backend will not be able to view the interface objects. Thus by splitting databases, you can limit the access of users and protect your databases.

3. Allows for Future Planning

You can reduce the size of an ever growing database by splitting it. After splitting the database, you can also upsize split databases to larger relational database management software like SQL Server. This is possible due to the easily formed links between frontend and SQL tables. Thus by using splitting in databases, the organization can use SQL for front end and Access for backend.

4. Easy to Modify User Interface

Databases need to be modified with growing business requirements. You need to update them with new features, most of which are possible only at the frontend, usually in the form of modified reports or forms. In-case of split databases, changes in the frontend can be made easily, without any kind of disruption. All you need to do is to link the Access frontend to the backend and test your program. However, things are not always going to be this easy, but it is easier to test new interface objects, while using a split database.

Split Databases Can Get Corrupted Due to Access Conflicts

While Ms Access allows up to 256 concurrent users to access the database, resource conflicts do occur in multiple user scenarios. In many cases such conflicts leads to incidents of MDB corruption which can prove tricky to resolve. To deal with such scenarios you need to call-in a [fix mdb](https://www.datanumen.com/access-repair/) tool that can handle intricate issues of database corruption. While choosing such a tool, make it a point to check the capacity of the tool to handle different storage media types and recover thoroughly compromised database files.

3.During training, how can you tell that your input pipeline is the bottleneck? What can you do to fix it?

Data Preprocessing Bottleneck

A CPU bottleneck occurs when the GPU resource is under utilized as a result of one, or more of the CPUs, having reached maximum utilization. In this situation, the GPU will be partially idle while it waits for the CPU to pass in training data. This is an undesired state. Being that the GPU is, typically, the most expensive resource in the system, your goal should always be to maximize its utilization. Without getting into too many technical details, a CPU bottleneck generally occurs when the ratio between the “amount” of data pre-processing, which is performed on the CPU, and the “amount” of compute performed by the model on the GPU, is greater that the ratio between the overall CPU compute capacity and the overall GPU compute capacity. For example, if both your CPU cores and GPU are maximally utilized, and then you upgrade to a more powerful GPU, or downgrade to a system with fewer CPU cores, your training runtime performance will become CPU bound.

Naturally, your first instinct will be to simply switch over to a machine with a more appropriate CPU to GPU compute ratio. But, sadly, most of us don’t have that freedom. And while cloud services, such as [Amazon SageMaker](https://aws.amazon.com/sagemaker/), offer a variety of [training instance types](https://aws.amazon.com/sagemaker/pricing/), with different CPU-compute to GPU-compute ratios, you may find that none of them quite fit your specific needs.

Assuming that you are stuck with the system that you have, what steps can you take to address your performance bottleneck and speed up the training?

In the next sections we will propose four steps for addressing the preprocessing data bottleneck.

Identify any operations that can be moved to the data preparation phase

Optimize the data pre-processing pipeline

Perform some of the pre-processing steps on the GPU

Use the TensorFlow data service to offload some of the CPU compute to other machines

In order to facilitate our discussion, we will build a toy example based on Resnet50.

Sample Use Case

In the code block below, I have built a model using TensorFlow’s built in [Resnet50](https://www.tensorflow.org/api_docs/python/tf/keras/applications/ResNet50) application. I have added a relatively heavy data pre-processing pipeline which includes dilation, blur filtering, and a number of TensorFlow pre-processing layers. (See the [documentation](https://www.tensorflow.org/guide/keras/preprocessing_layers) for the advantages of using such layers.)

import tensorflow as tf  
import tensorflow\_addons as tfa  
from tensorflow.keras.applications.resnet50 import ResNet50  
from tensorflow.keras.layers.experimental import preprocessingdef get\_dataset(batch\_size):  
 # parse TFRecord  
 def parse\_image\_function(example\_proto):  
 image\_feature\_description =   
 {'image': tf.io.FixedLenFeature([], tf.string),  
 'label': tf.io.FixedLenFeature([], tf.int64)}  
 features = tf.io.parse\_single\_example(  
 example\_proto, image\_feature\_description)  
 image = tf.io.decode\_raw(features['image'], tf.uint8)  
 image.set\_shape([3 \* 32 \* 32])  
 image = tf.reshape(image, [32, 32, 3])  
 label = tf.cast(features['label'], tf.int32)  
 return image, label # dilation filter  
 def dilate(image, label):  
 dilateFilter = tf.zeros([3, 3, 3], tf.uint8)  
 image = tf.expand\_dims(image, 0)  
 image = tf.nn.dilation2d(  
 image, dilateFilter, strides=[1, 1, 1, 1],  
 dilations=[1, 1, 1, 1],  
 padding='SAME',   
 data\_format='NHWC')  
 image = tf.squeeze(image)  
 return image, label # blur filter  
 def blur(image, label):  
 image = tfa.image.gaussian\_filter2d(image=image,  
 filter\_shape=(11, 11), sigma=0.8)  
 return image, label # rescale filter  
 def rescale(image, label):  
 image = preprocessing.Rescaling(1.0 / 255)(image)  
 return image, label # augmentation filters  
 def augment(image, label):  
 data\_augmentation = tf.keras.Sequential(  
 [preprocessing.RandomFlip("horizontal"),  
 preprocessing.RandomRotation(0.1),  
 preprocessing.RandomZoom(0.1)])  
 image = data\_augmentation(image)  
 return image, label autotune = tf.data.experimental.AUTOTUNE  
 options = tf.data.Options()  
 options.experimental\_deterministic = False  
 records = tf.data.Dataset.list\_files('data/\*',   
 shuffle=True).with\_options(options)  
 # load from TFRecord files  
 ds = tf.data.TFRecordDataset(records,   
 num\_parallel\_reads=autotune).repeat()  
 ds = ds.map(parse\_image\_function, num\_parallel\_calls=autotune)  
 ds = ds.map(dilate, num\_parallel\_calls=autotune)  
 ds = ds.map(blur, num\_parallel\_calls=autotune)  
 ds = ds.batch(batch\_size)  
 ds = ds.map(rescale,num\_parallel\_calls=autotune)  
 ds = ds.map(augment, num\_parallel\_calls=autotune)  
 ds = ds.prefetch(autotune)  
 return dsif \_\_name\_\_ == "\_\_main\_\_":   
 model = ResNet50(weights=None,  
 input\_shape=(32, 32, 3),  
 classes=10)  
 model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),  
 optimizer=tf.optimizers.Adam())  
 dataset = get\_dataset(batch\_size = 1024)  
 model.fit(dataset, steps\_per\_epoch=100, epochs=10))

The raw data input is stored in [TFRecord](https://www.tensorflow.org/tutorials/load_data/tfrecord" \t "_blank) files, which I created from the [CIFAR-10](https://www.cs.toronto.edu/~kriz/cifar.html) dataset, (using [this](https://github.com/aws/amazon-sagemaker-examples/blob/master/sagemaker-debugger/tensorflow_profiling/demo/generate_cifar10_tfrecords.py) script).

I have created this example so as to artificially create a performance bottleneck. I would not, under any circumstances, recommend using it for actual training.

All tests were run on an [Amazon ec2](https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc) [p2.xlarge](https://aws.amazon.com/ec2/instance-types/p2/) instance type using an [Amazon Deep Learning AMI](https://aws.amazon.com/machine-learning/amis/).

Identifying the Bottleneck

There are a number of different tools and techniques for evaluating the runtime performance of a training session, and identifying and studying an input pipeline bottleneck.

4.Can you save any binary data to a TFRecord file, or only serialized protocol buffers?

In this post, I’m going to discuss Tensorflow Records. Tensorflow recommends to store and read data in tfRecords format. It internally uses Protocol Buffers to serialize/deserialize the data and store them in bytes, as it takes less space to hold an ample amount of data and to transfer them as well.

Protobufs work with the predefined schemas, unlike JSON and XML. tfRecords has such schemas already available in Protofiles, and also the compiled code for many supported languages. I, in the post, am going to import such compiled code in Python and use them on my data.

This post needs a basic understanding of Protobufs, as they are the building blocks of TFRecords. I have also written a post earlier about working with [Protobufs](https://medium.com/@gshbehera/serialization-deserialization-with-protobufs-223d401f621d). In this post, I start by going over the predefined protos for tfRecords, using them in Python with some dos and don’ts. Then I’ve two demos where I’ll be making tfRecords from existing datasets.

5.Why would you go through the hassle of converting all your data to the Example protobuf format? Why not use your own protobuf definition?

Protocol buffers provide a serialization format for packets of typed, structured data that are up to a few megabytes in size. The format is suitable for both ephemeral network traffic and long-term data storage. Protocol buffers can be extended with new information without invalidating existing data or requiring code to be updated.

Protocol buffers are the most commonly-used data format at Google. They are used extensively in inter-server communications as well as for archival storage of data on disk. Protocol buffer messages and services are described by engineer-authored .proto files. The following shows an example message:

message Person {  
  optional string name = 1;  
  optional int32 id = 2;  
  optional string email = 3;  
}

The proto compiler is invoked at build time on .proto files to generate code in various programming languages (covered in [Cross-language Compatibility](https://developers.google.com/protocol-buffers/docs/overview#cross-lang) later in this topic) to manipulate the corresponding protocol buffer. Each generated class contains simple accessors for each field and methods to serialize and parse the whole structure to and from raw bytes. The following shows you an example that uses those generated methods:

Person john = Person.newBuilder()  
    .setId(1234)  
    .setName("John Doe")  
    .setEmail("jdoe@example.com")  
    .build();  
output = new FileOutputStream(args[0]);  
john.writeTo(output);

Because protocol buffers are used extensively across all manner of services at Google and data within them may persist for some time, maintaining backwards compatibility is crucial. Protocol buffers allow for the seamless support of changes, including the addition of new fields and the deletion of existing fields, to any protocol buffer without breaking existing services. For more on this topic, see [Updating Proto Definitions Without Updating Code](https://developers.google.com/protocol-buffers/docs/overview#updating-defs), later in this topic.

What are the Benefits of Using Protocol Buffers?

Protocol buffers are ideal for any situation in which you need to serialize structured, record-like, typed data in a language-neutral, platform-neutral, extensible manner. They are most often used for defining communications protocols (together with gRPC) and for data storage.

Some of the advantages of using protocol buffers include:

Compact data storage

Fast parsing

Availability in many programming languages

Optimized functionality through automatically-generated classes

Cross-language Compatibility

The same messages can be read by code written in any supported programming language. You can have a Java program on one platform capture data from one software system, serialize it based on a .proto definition, and then extract specific values from that serialized data in a separate Python application running on another platform.

The following languages are supported directly in the protocol buffers compiler, protoc:

[C++](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated#invocation)

[C#](https://developers.google.com/protocol-buffers/docs/reference/csharp-generated#invocation)

[Java](https://developers.google.com/protocol-buffers/docs/reference/java-generated#invocation)

[Kotlin](https://developers.google.com/protocol-buffers/docs/reference/kotlin-generated#invocation)

[Objective-C](https://developers.google.com/protocol-buffers/docs/reference/objective-c-generated#invocation)

[PHP](https://developers.google.com/protocol-buffers/docs/reference/php-generated#invocation)

[Python](https://developers.google.com/protocol-buffers/docs/reference/python-generated#invocation)

[Ruby](https://developers.google.com/protocol-buffers/docs/reference/ruby-generated#invocation)

The following languages are supported by Google, but the projects' source code resides in GitHub repositories. The protoc compiler uses plugins for these languages:

[Dart](https://github.com/google/protobuf.dart)

[Go](https://github.com/protocolbuffers/protobuf-go)

Additional languages are not directly supported by Google, but rather by other GitHub projects. These languages are covered in [Third-Party Add-ons for Protocol Buffers](https://github.com/protocolbuffers/protobuf/blob/master/docs/third_party.md).

Cross-project Support

You can use protocol buffers across projects by defining message types in .proto files that reside outside of a specific project’s code base. If you're defining message types or enums that you anticipate will be widely used outside of your immediate team, you can put them in their own file with no dependencies.

A couple of examples of proto definitions widely-used within Google are [timestamp.proto](https://github.com/protocolbuffers/protobuf/blob/master/src/google/protobuf/timestamp.proto) and [status.proto](https://github.com/googleapis/googleapis/blob/master/google/rpc/status.proto).

Updating Proto Definitions Without Updating Code

It’s standard for software products to be backward compatible, but it is less common for them to be forward compatible. As long as you follow some [simple practices](https://developers.google.com/protocol-buffers/docs/proto#updating) when updating .proto definitions, old code will read new messages without issues, ignoring any newly added fields. To the old code, fields that were deleted will have their default value, and deleted repeated fields will be empty. For information on what “repeated” fields are, see [Protocol Buffers Definition Syntax](https://developers.google.com/protocol-buffers/docs/overview#syntax) later in this topic.

New code will also transparently read old messages. New fields will not be present in old messages; in these cases protocol buffers provide a reasonable default value.

When are Protocol Buffers not a Good Fit?

Protocol buffers do not fit all data. In particular:

Protocol buffers tend to assume that entire messages can be loaded into memory at once and are not larger than an object graph. For data that exceeds a few megabytes, consider a different solution; when working with larger data, you may effectively end up with several copies of the data due to serialized copies, which can cause surprising spikes in memory usage.

When protocol buffers are serialized, the same data can have many different binary serializations. You cannot compare two messages for equality without fully parsing them.

Messages are not compressed. While messages can be zipped or gzipped like any other file, special-purpose compression algorithms like the ones used by JPEG and PNG will produce much smaller files for data of the appropriate type.

Protocol buffer messages are less than maximally efficient in both size and speed for many scientific and engineering uses that involve large, multi-dimensional arrays of floating point numbers. For these applications, [FITS](https://en.wikipedia.org/wiki/FITS) and similar formats have less overhead.

Protocol buffers are not well supported in non-object-oriented languages popular in scientific computing, such as Fortran and IDL.

Protocol buffer messages don't inherently self-describe their data, but they have a fully reflective schema that you can use to implement self-description. That is, you cannot fully interpret one without access to its corresponding .proto file.

Protocol buffers are not a formal standard of any organization. This makes them unsuitable for use in environments with legal or other requirements to build on top of standards.

6.When using TFRecords, when would you want to activate compression? Why not do it systematically?

A hands-on guide to TFRecords

An introduction on working with image, audio, and text data

TensorFlow’s custom data format TFRecord is really useful. The files are supported natively by the blazing-fast tf.data API, support distributed datasets, and leverage parallel I/O. But they are somewhat overwhelming at first. This post serves as a hands-on introduction.

Overview

In the following, we’ll use artificial data to go over the concept behind TFRecord files. With this in mind, we can then go on to work with images; we will use both a small and a large dataset. Expanding our knowledge, we then work with audio data. The last large domain is the text domain, which we’ll cover as well. To combine all this, we create an artificial multi-data-type dataset and, you guessed it, write it to TFRecords as well.

TFRecord’s layout

When I started my deep learning research, I naively stored my data scattered over the disk. To make things worse, I polluted my directories with thousands of small files, in the order of a few KB. The cluster I was then working on was not amused. And it took quite some time to get all these files loaded.

This is where TFRecords (or large NumPy arrays, for that matter) come in handy: Instead of storing the data scattered around, forcing the disks to jump between blocks, we simply store the data in a sequential layout.

Visualization created by the author

The TFRecord file can be seen as a wrapper around all the single data samples. Every single data sample is called an Example and is essentially a dictionary storing the mapping between a key and our actual data.

Now, the seemingly complicated part is this: When you want to write your data to TFRecords, you first have to convert your data to a Feature.

So far, so good. But what is now the difference to storing your data in a compressed NumPy array or a pickle file? Two things: The TFRecord file is stored sequentially, enabling fast streaming due to low access times. And secondly, the TFRecord files are natively integrated into TensorFlows tf.data API, easily enabling batching, shuffling, caching, and the like.

As a bonus, if you ever have the chance and the computing resources to do multi-worker training, you can distribute the dataset across your machines.

On a code level, the feature creation happens with these convenient methods, which we will talk about later on:

To write data to TFRecord files, you first create a dictionary that says

I want to store this data point under this key

When reading from TFRecord files, you invert this process by creating a dictionary that says

I have this keys, fill this placeholder with the value stored at this key

Let us see how this looks in action.

Image data, small

A small cat. Photo by [Kote Puerto](https://unsplash.com/@kotecinho?utm_source=medium&utm_medium=referral" \t "_blank) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Images are a common domain in deep learning, with MNIST [1] and ImageNet [2] being two well-known datasets. There is a multitude of getting your images from the disk into the model: [writing a custom generator](https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly), using Keras’ [built-in tools](https://keras.io/api/preprocessing/image/#imagedatagenerator-class), or loading it from a NumPy array. To make loading and parsing image data-efficient, we can resort to TFRecords as the underlying file format.

The procedure is as follows: We first create some random images — that is, using NumPy to randomly fill a matrix of given image shape: width, height, and colour channels:

The output is as expected; we have 100 images of shape 250x250, with three channels each:

(100, 250, 250, 3)

We also create some artificial labels:

As a result, we have a label array of shape (100,1), storing one label per image. The first ten labels are printed out:

(100, 1)  
[[2] [4] [3] [3] [2] [4] [2] [3] [3] [0]]

To get these {image, label} pairs into the TFRecord file, we write a short method, taking an image and its label. Using our helper functions defined above, we create a dictionary to store the shape of our image in the keys height, width, and depth — we need this information to reconstruct our image later on. Next, we also store the actual image as raw\_image. For this, we first serialize the array (think building a long list) and then convert it to a bytes\_feature. Lastly, we store the label for our image.

All these key:value mappings make up the features for one Example, as described above:

Now that we have defined how we can create an Example from one pair of {image, label}, we need a function to write our complete dataset to a TFRecord file.

We begin by creating a TFRecordWriter, which is subsequently used to write the Examples to disk. For each image and corresponding label, we then use the function above to create such an object. Before writing it to disk, we have to serialize it. After we have consumed our data, we close our writer and print the number of files we have just parsed:

That is all that’s required to write images to a TFRecord file:

The output is as expected, as we have just parsed one hundred {image, label} pairs:

Wrote 100 elements to TFRecord

Having this file on our disk, we might also be interested in reading it later on. This is also possible and goes the other way:

Earlier, we defined a dictionary that we used to write our content to disk. We use a similar structure now, but this time to read the data. Previously, we said that the key width contains data of type int. Consequently, when we create our dictionary, we assign a placeholder of type int as well. And because we are dealing with features of fixed length (which we work with most of the time; sparse tensors are infrequently used), we say:

Give me the data that we have stored in the key ‘width’, and fill this placeholder with it

Similarly, we define the key:placeholder mappings for the other stored features. We then let the placeholders be filled by parsing our element with parse\_single\_example. Given that we are dealing with a dictionary, we can afterwards normally extract all values by accessing the corresponding keys.

In the last step, we have to parse our image back from a serialized form to the (height, width, channels) layout. Notice that we want the out\_type to be int16, which is required since we created the images with int16, too:

To create a dataset out of the parse elements, we simply leverage the tf.data API. We create a TFRecordDataset by pointing it to the TFRecord file on our disk and then apply our previous parsing function to every extracted Example. This returns a dataset:

We can explore the content of our dataset by taking a single data point:

The output is

(250, 250, 3)  
()

The first line is the shape of one image; the second line is the shape of a scalar element, which has no dimension.

This marks the end of parsing a small dataset. In the next section, we have a look at parsing a larger dataset, creating multiple TFRecord files on the way.

Image data, large

A large cat. Photo by [Timothy Meinberg](https://unsplash.com/@timothymeinberg?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

In the previous section, we wrote a fairly small dataset to a single TFRecord file. For larger datasets, we might consider sharding our data across multiple such files.

First, let’s create a random image dataset:

The corresponding labels are created in the next step:

Since we are dealing with a larger dataset now, we first have to determine how many shards we even need. To calculate this, we need both the number of total files and the number of elements that we want to store within a single shard. We also have to consider the case where we have, e.g., 64 images and 10 files per shard. This would lead to 6 shards (6x10) but misses the last 4 samples.

7.Data can be preprocessed directly when writing the data files, or within the tf.data pipeline, or in preprocessing layers within your model, or using TF Transform. Can you list a few pros and cons of each option?

Keras preprocessing

The Keras preprocessing layers API allows developers to build Keras-native input processing pipelines. These input processing pipelines can be used as independent preprocessing code in non-Keras workflows, combined directly with Keras models, and exported as part of a Keras SavedModel.

With Keras preprocessing layers, you can build and export models that are truly end-to-end: models that accept raw images or raw structured data as input; models that handle feature normalization or feature value indexing on their own.

Available preprocessing

Text preprocessing

[tf.keras.layers.TextVectorization](https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization): turns raw strings into an encoded representation that can be read by an Embedding layer or Dense layer.

Numerical features preprocessing

[tf.keras.layers.Normalization](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization): performs feature-wise normalize of input features.

[tf.keras.layers.Discretization](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Discretization): turns continuous numerical features into integer categorical features.

Categorical features preprocessing

[tf.keras.layers.CategoryEncoding](https://www.tensorflow.org/api_docs/python/tf/keras/layers/CategoryEncoding): turns integer categorical features into one-hot, multi-hot, or count dense representations.

[tf.keras.layers.Hashing](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Hashing): performs categorical feature hashing, also known as the "hashing trick".

[tf.keras.layers.StringLookup](https://www.tensorflow.org/api_docs/python/tf/keras/layers/StringLookup): turns string categorical values an encoded representation that can be read by an Embedding layer or Dense layer.

[tf.keras.layers.IntegerLookup](https://www.tensorflow.org/api_docs/python/tf/keras/layers/IntegerLookup): turns integer categorical values into an encoded representation that can be read by an Embedding layer or Dense layer.

Image preprocessing

These layers are for standardizing the inputs of an image model.

[tf.keras.layers.Resizing](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Resizing): resizes a batch of images to a target size.

[tf.keras.layers.Rescaling](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Rescaling): rescales and offsets the values of a batch of image (e.g. go from inputs in the [0, 255] range to inputs in the [0, 1] range.

[tf.keras.layers.CenterCrop](https://www.tensorflow.org/api_docs/python/tf/keras/layers/CenterCrop): returns a center crop of a batch of images.

Image data augmentation

These layers apply random augmentation transforms to a batch of images. They are only active during training.

[tf.keras.layers.RandomCrop](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomCrop)

[tf.keras.layers.RandomFlip](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomFlip)

[tf.keras.layers.RandomTranslation](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomTranslation)

[tf.keras.layers.RandomRotation](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomRotation)

[tf.keras.layers.RandomZoom](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomZoom)

[tf.keras.layers.RandomHeight](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomHeight)

[tf.keras.layers.RandomWidth](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomWidth)

[tf.keras.layers.RandomContrast](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomContrast)

The adapt() method

Some preprocessing layers have an internal state that can be computed based on a sample of the training data. The list of stateful preprocessing layers is:

TextVectorization: holds a mapping between string tokens and integer indices

StringLookup and IntegerLookup: hold a mapping between input values and integer indices.

Normalization: holds the mean and standard deviation of the features.

Discretization: holds information about value bucket boundaries.

Crucially, these layers are non-trainable. Their state is not set during training; it must be set before training, either by initializing them from a precomputed constant, or by "adapting" them on data.

You set the state of a preprocessing layer by exposing it to training data, via the adapt() method:

import numpy as np  
import tensorflow as tf  
from tensorflow.keras import layers  
  
data = np.array([[0.1, 0.2, 0.3], [0.8, 0.9, 1.0], [1.5, 1.6, 1.7],])  
layer = layers.Normalization()  
layer.adapt(data)  
normalized\_data = layer(data)  
  
print("Features mean: %.2f" % (normalized\_data.numpy().mean()))  
print("Features std: %.2f" % (normalized\_data.numpy().std()))

Features mean: -0.00

Features std: 1.00

The adapt() method takes either a Numpy array or a [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset) object. In the case of StringLookup and TextVectorization, you can also pass a list of strings:

data = []  
layer = layers.TextVectorization()  
layer.adapt(data)  
vectorized\_text = layer(data)  
print(vectorized\_text)

tf.Tensor(

[[37 12 25 5 9 20 21 0 0]

[51 34 27 33 29 18 0 0 0]

[49 52 30 31 19 46 10 0 0]

[ 7 5 50 43 28 7 47 17 0]

[24 35 39 40 3 6 32 16 0]

[ 4 2 15 14 22 23 0 0 0]

[36 48 6 38 42 3 45 0 0]

[ 4 2 13 41 53 8 44 26 11]], shape=(8, 9), dtype=int64)

In addition, adaptable layers always expose an option to directly set state via constructor arguments or weight assignment. If the intended state values are known at layer construction time, or are calculated outside of the adapt() call, they can be set without relying on the layer's internal computation. For instance, if external vocabulary files for the TextVectorization, StringLookup, or IntegerLookup layers already exist, those can be loaded directly into the lookup tables by passing a path to the vocabulary file in the layer's constructor arguments.

Here's an example where we instantiate a StringLookup layer with precomputed vocabulary:

vocab = ["a", "b", "c", "d"]  
data = tf.constant([["a", "c", "d"], ["d", "z", "b"]])  
layer = layers.StringLookup(vocabulary=vocab)  
vectorized\_data = layer(data)  
print(vectorized\_data)

tf.Tensor(

[[1 3 4]

[4 0 2]], shape=(2, 3), dtype=int64)

Preprocessing data before the model or inside the model

There are two ways you could be using preprocessing layers:

Option 1: Make them part of the model, like this:

inputs = keras.Input(shape=input\_shape)  
x = preprocessing\_layer(inputs)  
outputs = rest\_of\_the\_model(x)  
model = keras.Model(inputs, outputs)

With this option, preprocessing will happen on device, synchronously with the rest of the model execution, meaning that it will benefit from GPU acceleration. If you're training on GPU, this is the best option for the Normalization layer, and for all image preprocessing and data augmentation layers.

Option 2: apply it to your [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset), so as to obtain a dataset that yields batches of preprocessed data, like this:

dataset = dataset.map(lambda x, y: (preprocessing\_layer(x), y))

With this option, your preprocessing will happen on CPU, asynchronously, and will be buffered before going into the model. In addition, if you call dataset.prefetch(tf.data.AUTOTUNE) on your dataset, the preprocessing will happen efficiently in parallel with training:

dataset = dataset.map(lambda x, y: (preprocessing\_layer(x), y))  
dataset = dataset.prefetch(tf.data.AUTOTUNE)  
model.fit(dataset, ...)

This is the best option for TextVectorization, and all structured data preprocessing layers. It can also be a good option if you're training on CPU and you use image preprocessing layers.

When running on TPU, you should always place preprocessing layers in the [tf.data](https://www.tensorflow.org/api_docs/python/tf/data) pipeline (with the exception of Normalization and Rescaling, which run fine on TPU and are commonly used as the first layer is an image model).

Benefits of doing preprocessing inside the model at inference time

Even if you go with option 2, you may later want to export an inference-only end-to-end model that will include the preprocessing layers. The key benefit to doing this is that it makes your model portable and it helps reduce the [training/serving skew](https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew).

When all data preprocessing is part of the model, other people can load and use your model without having to be aware of how each feature is expected to be encoded & normalized. Your inference model will be able to process raw images or raw structured data, and will not require users of the model to be aware of the details of e.g. the tokenization scheme used for text, the indexing scheme used for categorical features, whether image pixel values are normalized to [-1, +1] or to [0, 1], etc. This is especially powerful if you're exporting your model to another runtime, such as TensorFlow.js: you won't have to reimplement your preprocessing pipeline in JavaScript.