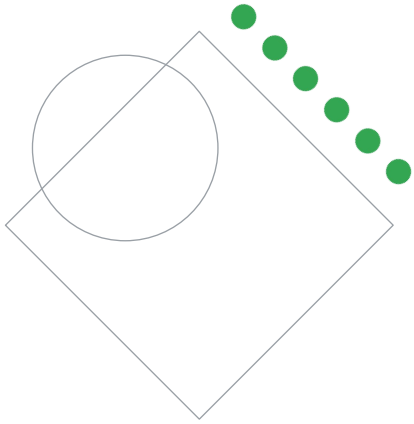
Design and Build an Input Data Pipeline



Welcome to the **Design and Build an Input Data Pipeline** module.

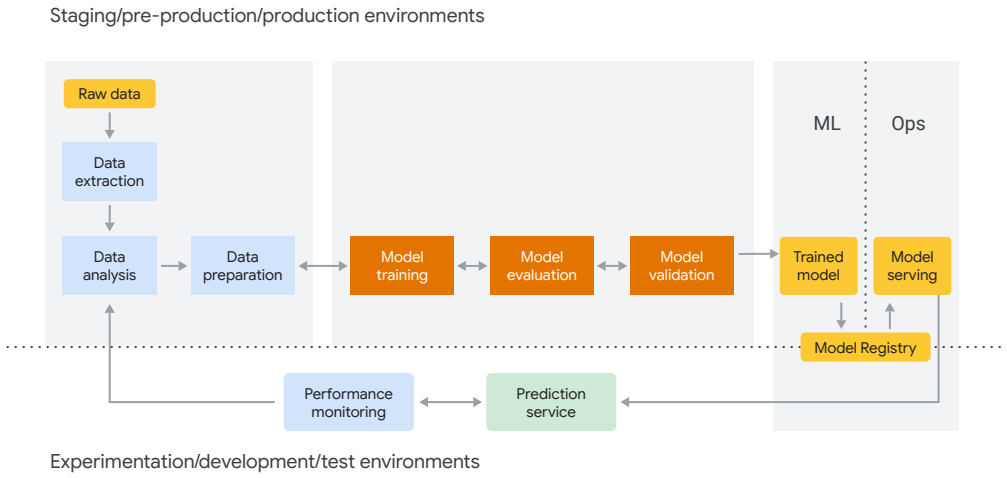
In this module, you learn to ...

01	Train on Large Datasets with tf.data
02	Work with in-memory files
03	Get the data ready for training
04	Describe embeddings
05	Understand scaling data with tf.Keras preprocessing layers

In this module, you'll learn to:

- Train on Large Datasets with tf.data
- Work with in-memory files
- Get the data ready for training
- Describe embeddings
- Understand scaling data with tf.Keras preprocessing layers

An ML recap

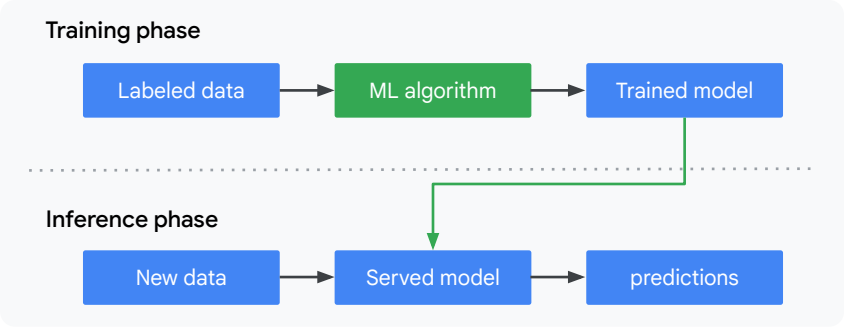


Let's start with a recap. In any ML project, after you define the business use case and establish the success criteria, the process of delivering an ML model to production involves the following steps.

These steps can be completed manually or can be completed by an automated pipeline:

1. Data extraction
2. Data analysis
3. Data preparation
4. Model training
5. Model evaluation
6. Model validation
7. Model serving, and,
8. Model monitoring

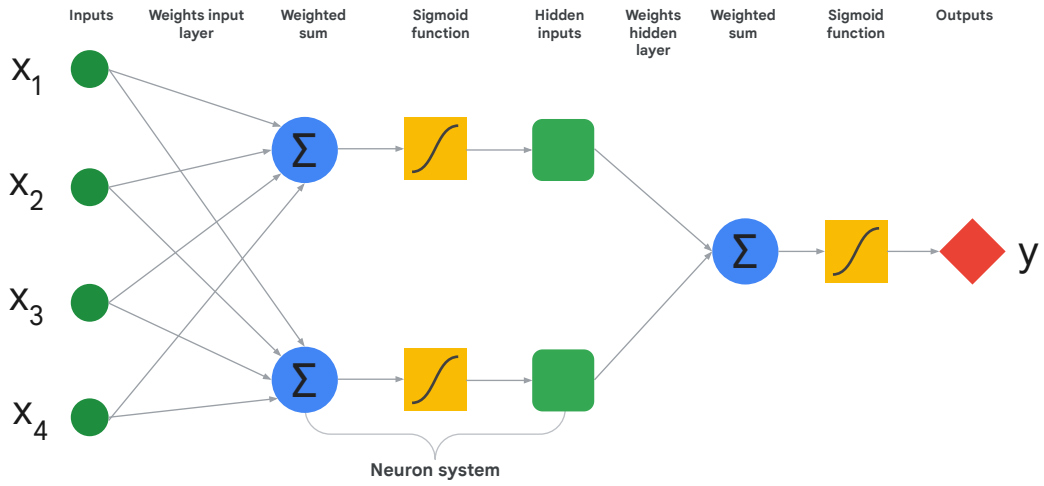
An ML recap



We saw that there are two phases in machine learning: a training phase and an inference phase.

We learned that an ML problem can be thought of as being all about data.

An ML recap

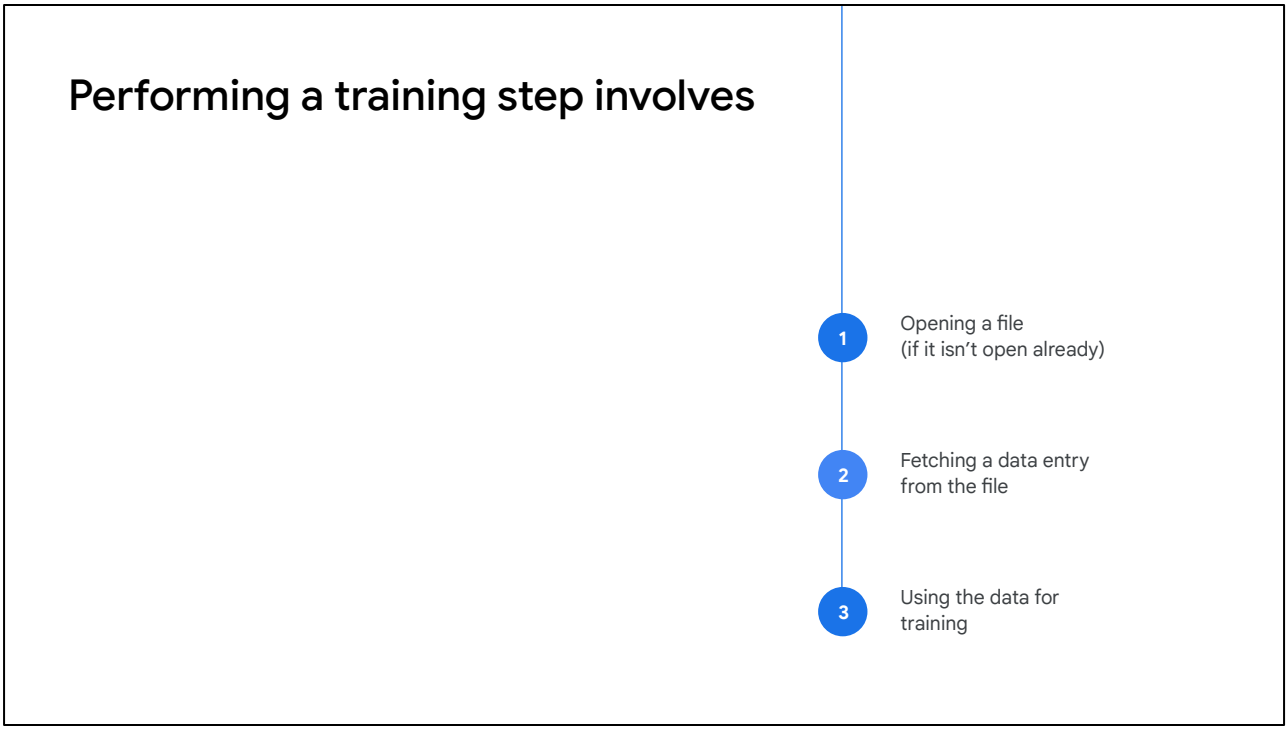


From a practical perspective, many machine learning models must represent the data (or features) as real-numbered vectors because the feature values must be multiplied by the model weights. In some cases, the data is raw and must be transformed to feature vectors.

Features, the columns of your dataframe, are key in assisting machine learning models to learn.

Better features result in faster training and more accurate predictions.

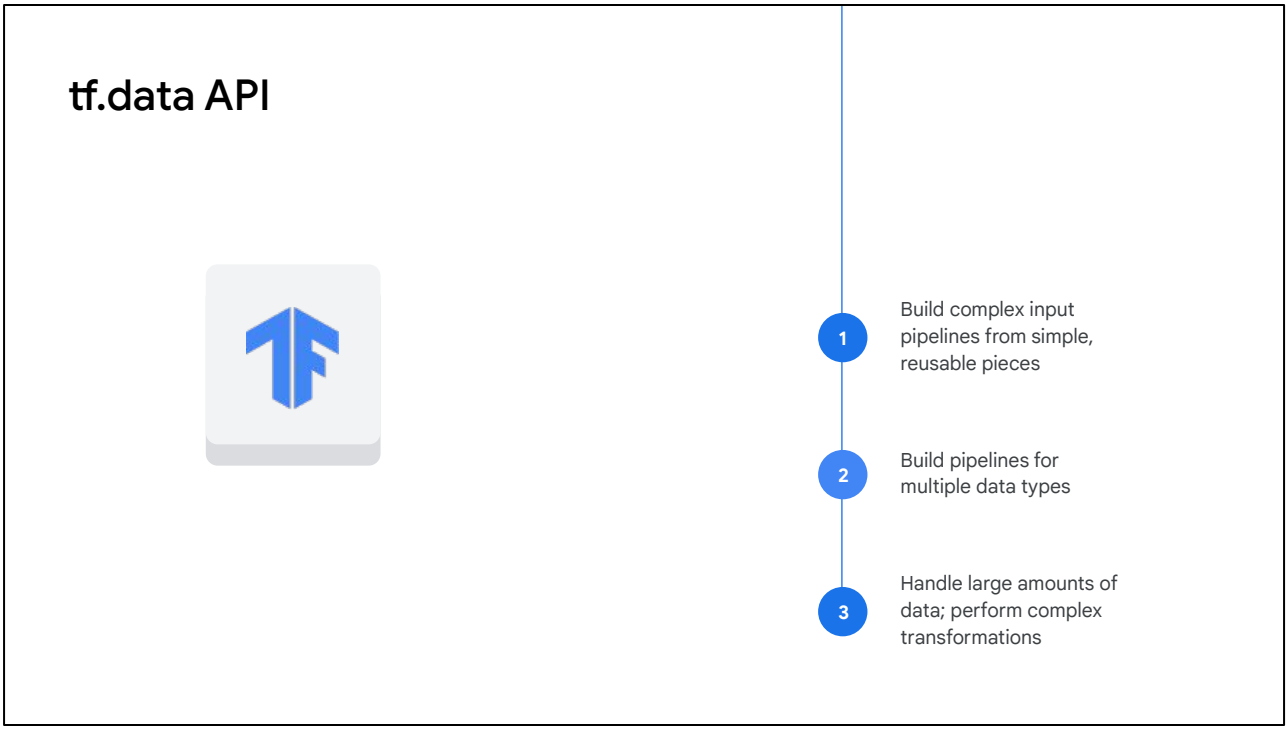
As the diagram shows, feature columns are input into the model—not as raw data, but as feature columns.



Having efficient data pipelines is of paramount importance for any machine learning model, because performing a training step involves:

1. Opening a file if it has not been opened
2. Fetching a data entry from the file and
3. Using the data for training

After you complete steps one and two, how do you use the data for training?

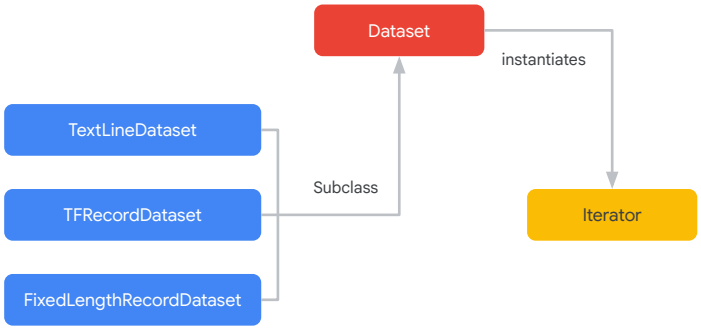


TensorFlow's Dataset module, tf.data, is one way to help build efficient data pipelines—and data pipelines are really just a series of data processing steps.

1. The tf.data API enables you to build complex input pipelines from simple, reusable pieces. For example, the pipeline for an image model might aggregate data from files in a distributed file system, apply random perturbations to each image, and merge randomly selected images into a batch for training.
2. The pipeline for a text model might involve extracting symbols from raw text data, converting them to embedding identifiers with a lookup table, and batching together sequences of different lengths.
3. The tf.data API makes it possible to handle large amounts of data, read from different data formats, and perform complex transformations.

We'll use the tf.data API quite a bit in this lesson!

Multiple ways to feed TensorFlow models with data



There are multiple ways to feed TensorFlow models with data, and you will see those in the next videos.

So, let's get started!

A tf.data.Dataset allows you to

- Create data pipelines from
  - in-memory dictionary and lists of tensors
  - out-of-memory sharded data files
- Preprocess data in parallel (and cache result of costly operations)
  `dataset = dataset.map(preproc_fun).cache()`
- Configure the way the data is fed into a model with a number of chaining methods
  `dataset = dataset.shuffle(1000).repeat(epochs).batch(batch_size, drop_remainder=True)`

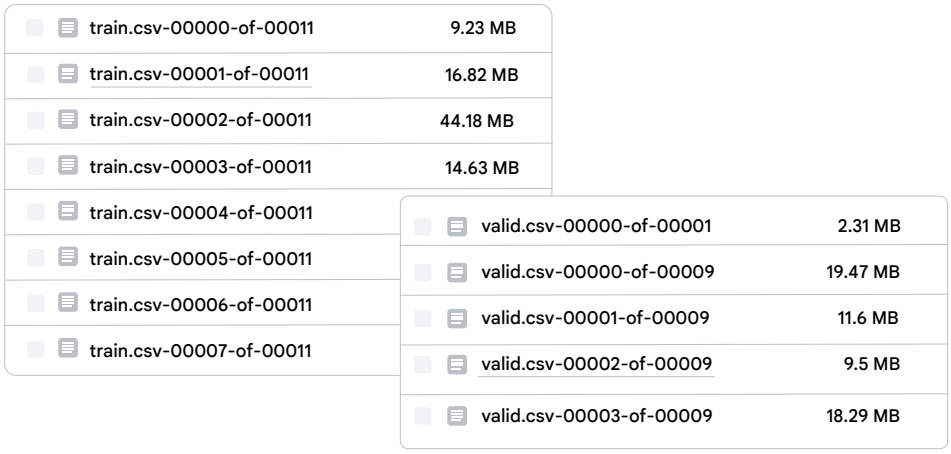
in a easy and very compact way

It's time to look at some specifics. The tf.data API introduces a tf.data.Dataset abstraction that represents a sequence of elements, in which each element consists of one or more components. For example, in an image pipeline, an element might be a single training example, with a pair of tensor components representing the image and its label.

There are two distinct ways to create a dataset:

- A data source constructs a Dataset from data stored in memory or in one or more files.
- A data transformation constructs a dataset from one or more tf.data.Dataset objects.

What about out-of-memory sharded datasets?

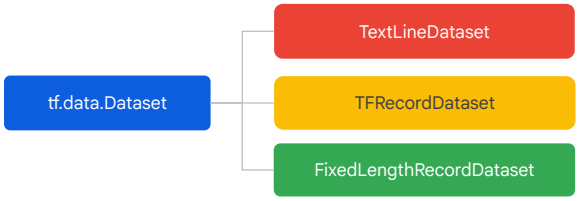


So what about data that won't fit into memory?

Large datasets tend to be sharded into multiple files which can be loaded progressively.

Remember that you train on mini-batches of data. You do not need to have the entire dataset in memory. One mini-batch is all you need for one training step.

Datasets can be created from different file formats



The Dataset API will help you create input functions for your model that will load data *progressively*.

There are specialized Dataset classes that can read data from text files like CSVs, TensorFlow records or fixed length record files.

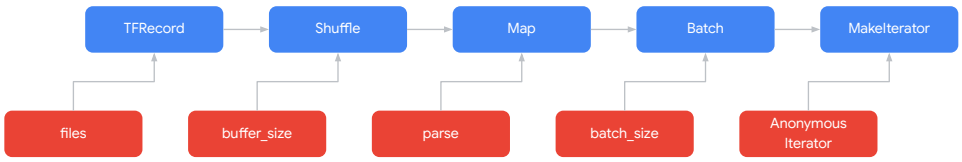
Datasets can be created from different file formats:

- Use **TextLineDataset** to instantiate a Dataset object which is comprised of lines from one or more text files.
- **TFRecordDataset** comprises records from one or more TFRecord files.
- And **FixedLengthRecordDataset** is a dataset object from fixed-length records from one or more binary files.

For anything else, you can use the generic Dataset class and add your own decoding code.

TFRecordDataset example

```
dataset = tf.data.TFRecordDataset(files)
dataset = dataset.shuffle(buffer_size=X)
dataset = dataset.map(lambda record: parse(record))
dataset = dataset.batch(batch_size=Y)

for element in dataset: #  iter() is called
  ...
```



Let's walk through an example of TFRecordDataset.

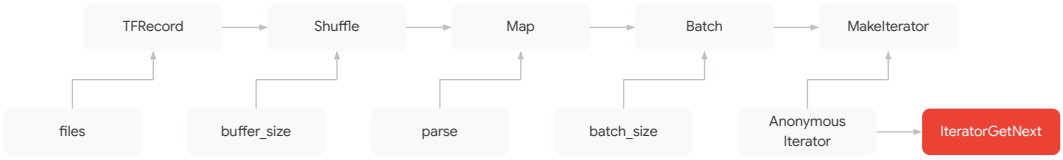
- At the beginning, the **TFRecord** op is created and executed, producing a variant tensor representing a dataset which is stored in the corresponding Python object.
- Next, the **Shuffle op** is executed, using the output of the TFRecord op as its input, connecting the two stages of the input pipeline.
- Next, the **user-defined function** is traced and passed as attribute to the Map operation, along with the Shuffle dataset variant input.
- Finally, the **Batch op** is created and executed, creating the final stage of the input pipeline.

When the for loop mechanism is used for enumerating the elements of dataset, the `**iter**` method is invoked on the dataset, which triggers creation and execution of two ops.

First an anonymous iterator op is created and executed, which results in creation of an iterator resource. Subsequently, this resource along with the Batch dataset variant is passed into the MakeIterator op, initializing the state of the iterator resource with the dataset.

TFRecordDataset example

```
dataset = tf.data.TFRecordDataset(files)
dataset = dataset.shuffle(buffer_size=X)
dataset = dataset.map(lambda record: parse(record))
dataset = dataset.batch(batch_size=Y)

for element in dataset:
  ... #  next() is called
```

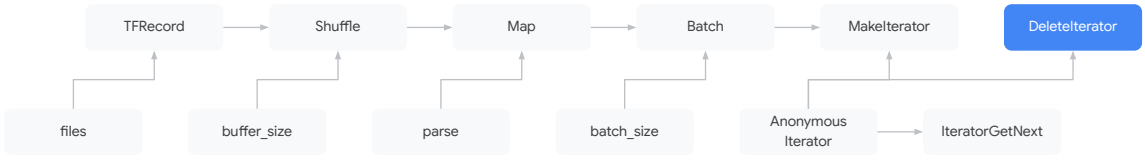


When the `next` method is called, it triggers creation and execution of the IteratorGetNext op, passing in the iterator resource as the input.

Note that the iterator op is created only once but executed as many times as there are elements in the input pipeline.

TFRecordDataset example

```
dataset = tf.data.TFRecordDataset(files)
dataset = dataset.shuffle(buffer_size=X)
dataset = dataset.map(lambda record: parse(record))
dataset = dataset.batch(batch_size=Y)

for element in dataset:
  ... #  iterator goes out of scope
```



Finally, when the Python iterator object goes out of scope, the DeleteIterator op is executed to make sure the iterator resource is properly disposed of.

To state the obvious, properly disposing of the iterator resource is essential as it is not uncommon for the iterator resource to allocate 100MBs to 1GBs of memory because of internal buffering.

Creating a dataset from in-memory tensors

```
def create_dataset(X, Y, epochs, batch_size):

    dataset = tf.data.Dataset.from_tensor_slices((X, Y))

    dataset = dataset.repeat(epochs).batch(batch_size, drop_remainder=True)

    return dataset
```

X = [x_0, x_1, ..., x_n] Y = [y_0, y_1, ..., y_n]

The dataset is made of slices of (X, Y) along the 1st axis

When data used to train a model sits in memory, we can create an input pipeline by constructing a Dataset using tf.data.Dataset.from_tensors() or tf.data.Dataset.from_tensor_slices().

Use from_tensors() or from_tensor_slices()

```
t = tf.constant([[4, 2], [5, 3]])
ds = tf.data.Dataset.from_tensors(t)   # [[4, 2], [5, 3]]


t = tf.constant([[4, 2], [5, 3]])
ds = tf.data.Dataset.from_tensor_slices(t)   # [4, 2], [5, 3]
```

from_tensors() combines the input and returns a dataset with a single element while from_tensor_slices() creates a dataset with a separate element for each row of the input tensor.

Read one CSV file using TextLineDataset

```
def parse_row(records):
   cols = tf.decode_csv(records, record_defaults=[[0], ['house'], [0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2]
   return features, label

def create_dataset(csv_file_path):
   dataset = tf.data.TextLineDataset(csv_file_path)
   dataset = dataset.map(parse_row)
   dataset = dataset.shuffle(1000).repeat(15).batch(128)
   return dataset
```

dataset = "[line1, line2, etc.]"

dataset = "[parse_row(line1), parse_row(line2), etc.]"

property type

sq_footage		PRICE in K$
1001,	house,	501
2001,	house,	1001
3001,	house,	1501
1001,	apt,	701
2001,	apt,	1301
3001,	apt,	1901
1101,	house,	526
2101,	house,	1026

Here is an example where you use TextLineDataset to load data from a CSV file. This is a Dataset comprising lines from one or more text files.

The TextLineDataset instantiation expects a file name and it has optional arguments such as, for example, the type of compression of the files or the number of parallel reads.

The map function is responsible for parsing each row of the CSV file. It returns a dictionary from the file content. Once that is done, shuffling, batching and prefetching are steps that can be applied to the dataset to allow for the data to be fed into the training loop iteratively.

Please note that it is recommended that we only shuffle the training data. So, for the shuffle operation, you may want to add a condition before applying the operation to the dataset.

Read a set of sharded CSV files using TextLineDataset

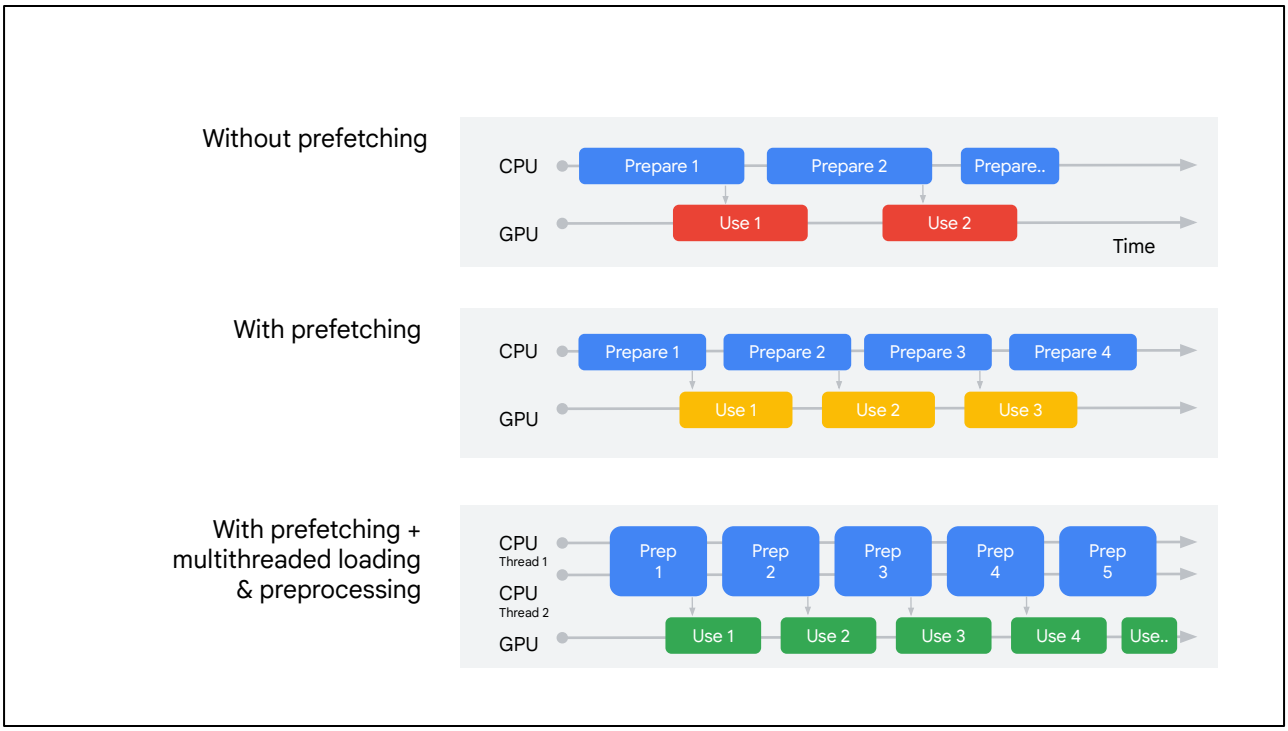
```
def parse_row(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label


def create_dataset(path):
    dataset = tf.data.Dataset.list_files(path)                    \
                             .flat_map(tf.data.TextLineDataset) \
                             .map(parse_row)

    dataset = dataset.shuffle(1000) \
                     .repeat(15)   \
                     .batch(128)
    return dataset
```

Finally, we have to address our initial concern: loading large datasets from a set of sharded files. An extra line of code will do.

We first scan the disk and load a dataset of file names using the Dataset.list_files functions. It supports a glob-like syntax with stars to match filenames with a common pattern. Then we use TextLineDataset to load the files and turn each file name into a dataset of text lines. We flat_map all of them together into a single dataset. And then, for each line of text we use map to apply the CSV parsing algorithm and obtain a dataset of features and labels.

Why are there two mapping functions: map and flat_map?

One of them is simply for one to one transformations and the other one for one-to-many transformations. Parsing a line of text is a one to one transformation so we apply it with map. When loading a file with TextLineDataset, one file name becomes a collection of text lines so that's a one to many transformation and is applied with flat_map to to flatten all the resulting text line datasets into a one.



Dataset allows for data to be prefetched.

Let's say we have a cluster with a GPU on it. Without prefetching, the CPU will be preparing the first batch while the GPU is waiting. Once that is done, the GPU can then run the computations on that batch. Once that is finished, the CPU will start preparing the next batch and so forth.

We can easily see that this is not very efficient.

Prefetching allows for subsequent batches to be prepared as soon as their previous batches have been sent to computation.

By combining prefetching with multithreading loading and preprocessing, we can achieve very good performance by making sure that the GPU (or GPUs) are constantly busy.

Now you know how to use Datasets to generate proper input functions for your models and get them training on large out of memory datasets. But Datasets also offer a rich API for working on and transforming your data. Take advantage of it!

The real benefit of Dataset is that you can do more than just ingest data

```
dataset = tf.data.TextLineDataset(filename)\
                 .skip(num_header_lines)\
                 .map(add_key)\
                 .map(decode_csv)\
                 .map(lambda feats, labels: preproc(feats), labels)
                 .filter(is_valid)\
                 .cache()
```

As we think about modeling a real problem with machine learning, we first need to think about what input signals we can use to train the model.

In this next section, let's use a common example. How about real estate? Can you predict the price of a property?

As you think about that problem, you must first choose your "features", that is, the data you will be basing your predictions on.

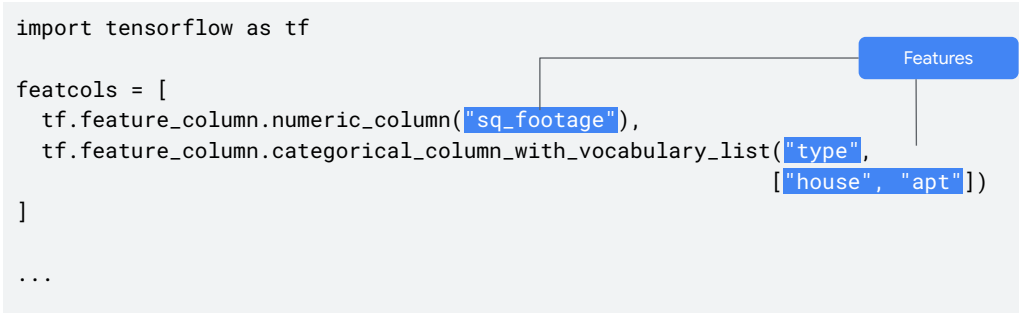
Why not try and build a model that predicts the price based on the area of a house or apartment.

Your features will be:

1. the square footage
2. the category: "house" or "apartment".

So far, the square footage is numeric. Numbers can be directly fed into a neural network for training, but we are going to come back to that later. The type of the property, though, is not numeric. This piece of information maybe be represented in a database by a string ("house" or "apartment"), and strings need to be transformed into numbers before being fed to a neural network.

Feature columns tell the model what inputs to expect



Here is how you implement this in code. You use the feature_column API to define the features.

First, a numeric column for the square footage. Then a categorical column for the property type. Two possible categories in this simple model: “house” or “apartment”. You probably noticed that the categorical column is called categorical_column_with_vocabulary_list. Use this when your inputs are in string or integer format, and you have an in-memory vocabulary mapping each value to an integer ID. By default, out-of-vocabulary values are ignored.

Other variations of it are:

- categorical_column_with_vocabulary_file (used when the inputs are in string or integer format, and there is a vocabulary file that maps each value to an integer ID.);
- categorical_column_with_identity (used when the inputs are integers in the range [0, num_buckets), and you want to use the input value itself as the categorical ID.)
- And finally categorical_column_with_hash_bucket (used when features are sparse and in string or integer format, and you want to distribute your inputs into a finite number of buckets by hashing.)

In this example, after the raw input is modified by feature_column transformations, you can then instantiate a LinearRegressor to train on these features. A Regressor is a model that outputs a number, in our example, the predicted sales price of the property.

Under the hood: Feature columns take care of packing the inputs into the input vector of the model

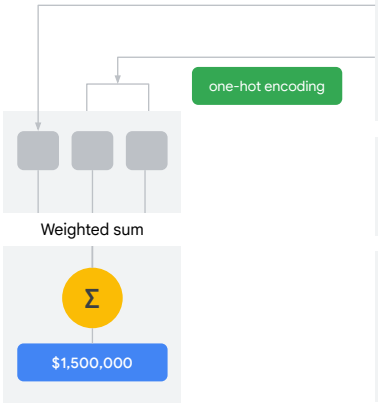
```
tf.feature_column.numeric_column("sq_footage")

tf.feature_column.categorical_column_with_
vocabulary_list("type", ["house", "apt"])
```

But why do you need feature columns in the content of model building?

Do you remember how they get used?

Let’s break it down.

Under the hood: Feature columns take care of packing the inputs into the input vector of the model




```
tf.feature_column.numeric_column("sq_footage")

tf.feature_column.categorical_column_with_
vocabulary_list("type", ["house", "apt"])
```

```
“ house” -> 1, 0
“ apt”    -> 0, 1
Many more...
```

```
tf.feature_column.bucketized_column(..)
tf.feature_column.embedding_column(..)
tf.feature_column.crossed_column(..)
tf.feature_column.categorical_column_with_hash_bucket(..)
...
```

- A linear regressor is a model that works on a vector of data. It computes a weighted sum of all input data elements and it can be trained to adjust the weights for your problem. Here predicting the sales price. But how can you pack your data into the single input vector that LinearRegressor expects? The answer is: in various ways, depending on what data you are packing and that is where the feature columns API comes in handy. It implements various standard ways of packing data into vectors elements.

- Here, values in your numeric column are just numbers. They get copied as they are into a single element of the input vector. On the other hand, your categorical column gets one-hot encoded.
  You have two categories so a house will be 1,0 while an apartment will become 0, 1.
  A third category would be encoded as 0, 0, 1 and so on. Now the LinearRegressor knows how to take the features you care about, pack them into its input vector and apply whatever a LinearRegressor does.

- Besides the categorical ones we've seen, there are many other mode feature column types to choose from: columns for continuous values you want to bucketize, word embeddings, column crosses, and so on. The transformations they apply are clearly described in the TensorFlow documentation so that you always know what is going on. We will look at some of them.

fc.bucketized_column splits a numeric feature into categories based on numeric ranges

```
NBUCKETS = 16
latbuckets = np.linspace(start=38.0, stop=42.0, num=NBUCKETS).tolist()
lonbuckets = np.linspace(start=-76.0, stop=-72.0, num=NBUCKETS).tolist()

fc_bucketized_plat = fc.bucketized_column(
    source_column=fc.numeric_column("pickup_longitude"),
    boundaries=lonbuckets)

fc_bucketized_plon = fc.bucketized_column(
    source_column=fc.numeric_column("pickup_latitude"),
    boundaries=latbuckets)
...
```

set up numeric ranges

create bucketized columns for pickup latitude and pickup longitude

Bucketized column helps with discretizing continuous feature values.

In this example, if we were to consider the latitude and longitude of the house or apartment we are training/predicting on, we would not want to feed the raw latitude/longitude values.

Instead, we would create buckets that could group the range of values for latitude and longitude.

Representing feature columns as sparse vectors

These are all different ways to create a categorical column.

If you know the keys beforehand:

```
tf.feature_column.categorical_column_with_vocabulary_list('zipcode',
    vocabulary_list = ['12345', '45678', '78900', '98723', '23451']),
```

If your data is already indexed; i.e., has integers in [0-N):

```
tf.feature_column.categorical_column_with_identity('schoolsRatings',
    num_buckets = 2)
```

If you don't have a vocabulary of all possible values:

```
tf.feature_column.categorical_column_with_hash_bucket('nearStoreID',
    hash_bucket_size = 500)
```

If you are thinking this seems familiar, and just like vocabulary building for categorical columns, you are absolutely correct.

Categorical columns are represented by TensorFlow as sparse tensors. So, categorical columns are an example of something that is sparse.

TensorFlow can do math operations on sparse tensors without having to convert them into dense.

This saves memory, and optimizes compute.

fc.embedding_column represents data as a lower-dimensional, dense vector

```
fc_ploc = fc.embedding_column(categorical_column=fc_crossed_ploc,
                              dimension=3)
```

lower dimensional, dense vector in which each cell contains a number, not just 0 or 1

As the number of categories of a feature grow large, it becomes infeasible to train a neural network using one-hot encodings.

Recall that we can use an embedding column to overcome this limitation. Instead of representing the data as a one-hot vector of many dimensions, an embedding column represents that data as a lower-dimensional, dense vector in which each cell can contain ANY number, not just 0 or 1.

We'll get back to our real estate example shortly but first let's take a quick detour into the world of embeddings.

fc.embedding_column represents data as a lower-dimensional, dense vector

```
fc_ploc = fc.embedding_column(categorical_column=fc_crossed_ploc,
                              dimension=3)
```

lower dimensional, dense vector in which each cell contains a number, not just 0 or 1

As the number of categories of a feature grow large, it becomes infeasible to train a neural network using one-hot encodings.

Recall that we can use an embedding column to overcome this limitation. Instead of representing the data as a one-hot vector of many dimensions, an embedding column represents that data as a lower-dimensional, dense vector in which each cell can contain ANY number, not just 0 or 1.

We'll get back to our real estate example shortly but first let's take a quick detour into the world of embeddings.

How can we visually cluster 10,000 variations of handwritten digits to look for similarities?

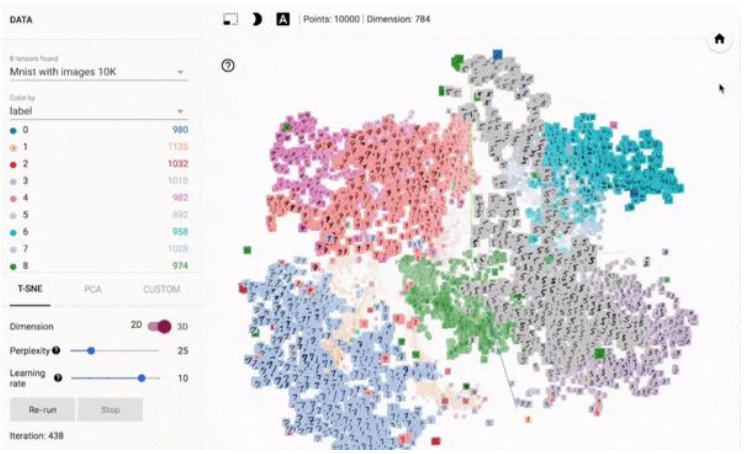
Embeddings!

Generally, Neural network embeddings have 3 primary purposes:

1. Finding nearest neighbors in the embedding space. These can be used to make recommendations based on user interests or cluster categories.
2. As input to a machine learning model for a supervised task.
3. For visualization of concepts and relations between categories.

Let’s take a look at an example using the popular handwritten digits dataset MNIST.

Embeddings are everywhere in modern machine learning

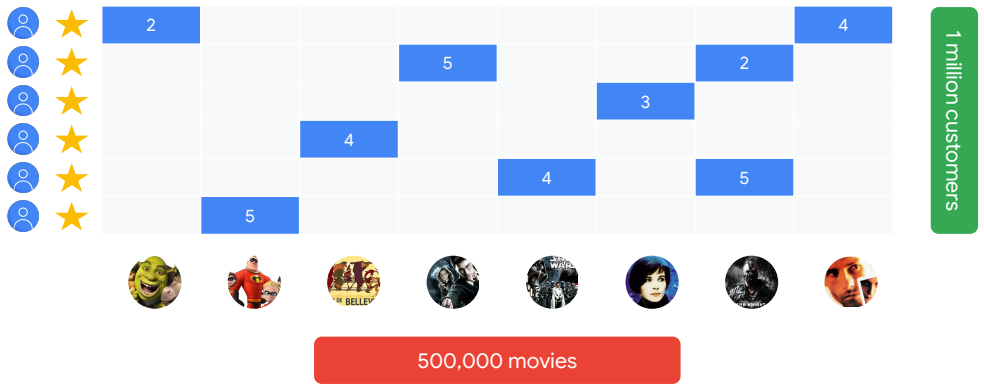


Here I've visualized in TensorBoard all 10,000 points of data where each colored cluster corresponds to a handwritten digit from 0 to 9. You can start to look for insights and even misclassifications by just exploring the dataset in 3D space.

If you take a look at the clusters while they spin, you'll see me clicking on the grey cluster which are handwritten 5s. It would seem in this dataset that people write 5s in many different ways (hence the visual distance between grey squares) as compared to something like a 1 or an 8.

If you take a sparse vector encoding and pass it through a embedding column and then use that embedding column as the input to a DNN and train the DNN, then the trained embeddings will have this similarity property. As long as, of course, you have enough data and your training achieved good accuracy.

How do you recommend movies to customers?



Next, let's look at embeddings in the context of movie recommendations. Let's say that we want to recommend movies to customers.

Let's say that our business has a million users and 500,000 movies. Remember that number. That is quite small, by the way. YouTube and eight other Google properties have a billion users!

For every user, our task is to recommend five to ten movies. We want to pick movies that they will watch, and will rate highly. We need to do this for a million users and for each user, select from 500,000 movies.

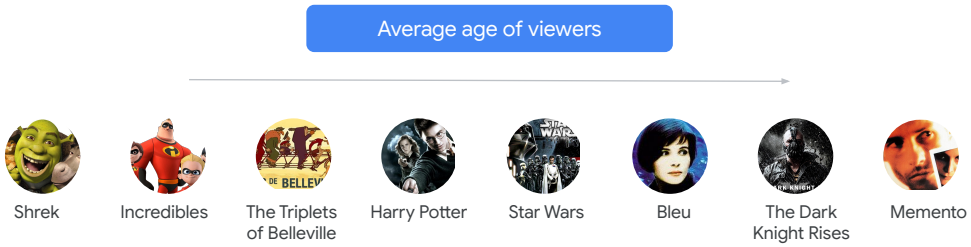
So, what's our input dataset? Our input dataset, if we represent it as a matrix, is 1 million rows and 500,000 columns.

The numbers in the diagram denote movies that customers have watched and rated.

What we need to do is to figure out the rest of the matrix.

To solve this problem some method is needed to determine which movies are similar to each other.

One approach is to organize movies by similarity (1D)



One approach is to organize movies by similarity using some attribute of the movies.

For example, we might look at the **average age of the audience** and put the movies on a line.

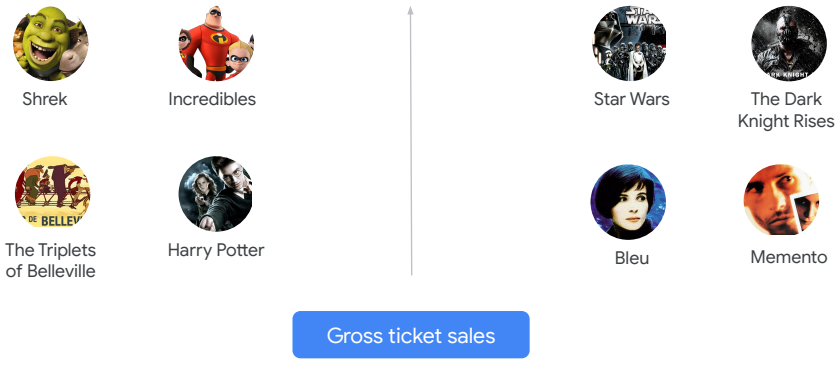
So, the cartoons and animated movies show up on the left-hand side and the darker, adult-oriented movies show up to the right. Then, we can say that if you liked the Incredibles, perhaps you are a child or you have a young child, and so we can recommend Shrek to you.

But … Blue and Memento are arthouse movies whereas Star Wars and the Dark Knight Rises are both blockbusters. If someone watched and like Bleu, they are more likely to like Memento than a movie about Batman.

Similarly, someone who watched and like Star Wars is more likely to like The Dark Knight Rises than some arthouse movie.

How do we solve this problem?

Using a second dimension gives us more freedom in organizing movies by similarity



What if we add a second dimension?

Perhaps the second dimension is the total number of tickets sold for that movie when it was released in theaters.

Now, we see that Star Wars and The Dark Knight Rises are close to each other. Bleu and Memento are close to each other.

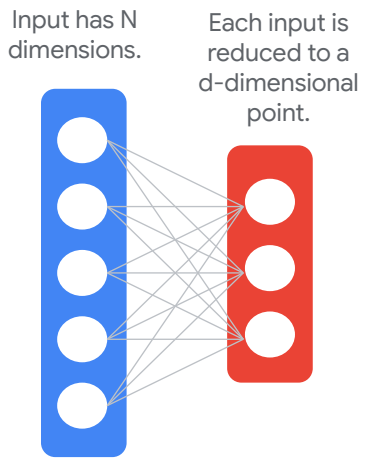
Shrek, Incredibles are close to each other as well. Harry Potter is in between the cartoons and Star Wars in that kids watch it, some adults watch it, and it's a blockbuster.

Notice how adding the second dimension has helped bring movies that are good recommendations closer together. It conforms much better to our intuition.

Do we have to stop at two dimensions? Of course not … by adding even more dimensions we can create finer distinctions. And sometimes, these finer distinctions can translate into better recommendations.

Not always … the danger of overfitting exists here too ...

A d-dimensional embedding assumes that user interest in movies can be approximated by d aspects



So, the idea is that we have an input that has N dimensions.

What is N in the case of the movies we looked at?

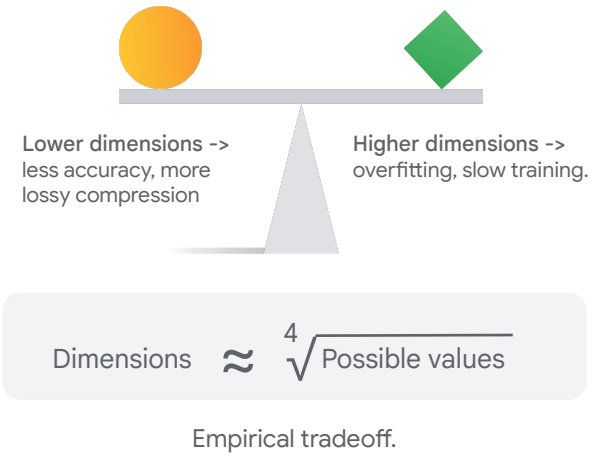
500,000-right? Remember that the movieId is a categorical feature and we'd normally be one-hot encoding it. So N = 500,000

On our case, we represented all the movies in a two-dimensional space. So, d = 2

The key point is that d is much, much less than N.

And the assumption is that user interest in movies can be represented by some d aspects.

A good starting point for number of embedding dimensions



In all our examples, we used 3 for the number of embeddings. You can use different numbers, of course. But what number should you use before you train?

This is a hyperparameter to your machine learning model. Hyperparameter means you set it before training. You will have to try different numbers of embedding dimensions because there is a tradeoff.

Higher-dimensional embeddings can more accurately represent the relationships between input values. However, the more dimensions you have, the greater the chance of overfitting. Also, the model gets larger and leads to slower training.

A good starting point is to go with the fourth root of the total number of possible values. For example, if you are embedding movieId and you have 500,000 movies in your catalog, a good starting point might be the fourth root of 500,000.

The square root of 500,00 is about 700. And the square root of 700 is about 26. So, I'd probably start at around 25.

In the hyperparameter tuning, I would specify a search space of perhaps 15 to 35.

But this is only a rule of thumb, of course.

fc.crossed_column enables a model to learn separate for combination of features

```
fc_crossed_ploc = fc.crossed_column([fc_bucketized_plat, fc_bucketized_plon],
                                    hash_bucket_size=NBUCKETS * NBUCKETS)
```

crossed_column is backed by a hashed_column, so you must set the size of the hash bucket

Another really cool thing you can do with features besides embeddings is actually combine the features into a new synthetic feature. Combining features into a single feature, better known as feature crosses, enables a model to learn separate weights for each combination of features.

A **synthetic feature** formed by crossing (taking a Cartesian product of) individual binary features obtained from **categorical data** or from **continuous features** via **bucketing**. Feature crosses help represent nonlinear relationships.

crossed_column does not build the full table of all possible combinations (which could be very large). Instead, it is backed by a hashed_column, so you can choose how large the table is.

Training input data requires dictionary of features and a label

```
def features_and_label():
   # sq_footage and type
   features = {"sq_footage": [ 1000,    2000,    3000,    1000,  2000,  3000],
               "type":       ["house", "house", "house", "apt", "apt", "apt"]}

   # prices in thousands
   labels =                  [ 500,     1000,    1500,    700,   1300,  1900]
   return features, labels
```

Back to our real estate example.

To train the model, you simply need to write an input function that returns the features named as in the feature columns.

Since you are training, you also need the correct answers called “labels”.

And now you can call the train function from the Keras API or from a custom model build, which will train the model by repeating this dataset 100 times, for example.

Create input pipeline using tf.data

```
def create_dataset(pattern, batch_size=1, mode=tf.estimator.ModeKeys.EVAL):
    dataset = tf.data.experimental.make_csv_dataset(
        pattern, batch_size, CSV_COLUMNS, DEFAULTS)
    dataset = dataset.map(features_and_labels)
    if mode == tf.estimator.ModeKeys.TRAIN:
        dataset = dataset.shuffle(buffer_size=1000).repeat()
    # take advantage of multi-threading; 1=AUTOTUNE
    dataset = dataset.prefetch(1)

return dataset
```

You will see how batching works later but for those of you who already know about the concept of batching, the code as written here trains on a single batch of data at each step and this batch contains the entire dataset.

When passing data to the built-in training loops of a model, you should either use **Numpy arrays** (if your data is small and fits in memory) or **tf.data Dataset** objects.

Use DenseFeatures layer to input feature columns to the Keras model

```
feature_columns = [...]

feature_layer = tf.keras.layers.DenseFeatures(feature_columns)

model = tf.keras.Sequential([
  feature_layer,
  layers.Dense(128, activation='relu'),
  layers.Dense(128, activation='relu'),
  layers.Dense(1, activation='linear')
])

...
```

Once you have defined the feature columns, you can use a DenseFeatures layer to input them to a Keras model. This layer is simply a layer that produces a dense Tensor based on the given feature columns.

What about compiling and training the Keras model?

After your dataset is created, passing it into the Keras model for training is simple:

```
model.fit()
```

You will learn and practice this later after first mastering dataset manipulation!

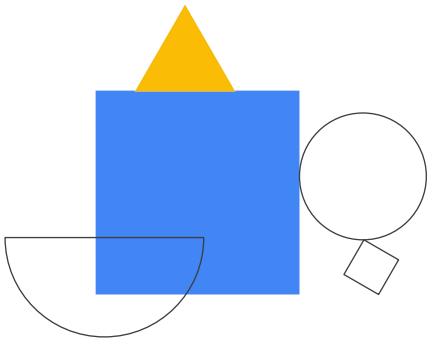
After your dataset is created, passing it into the Keras model for training is simple:

model.fit()

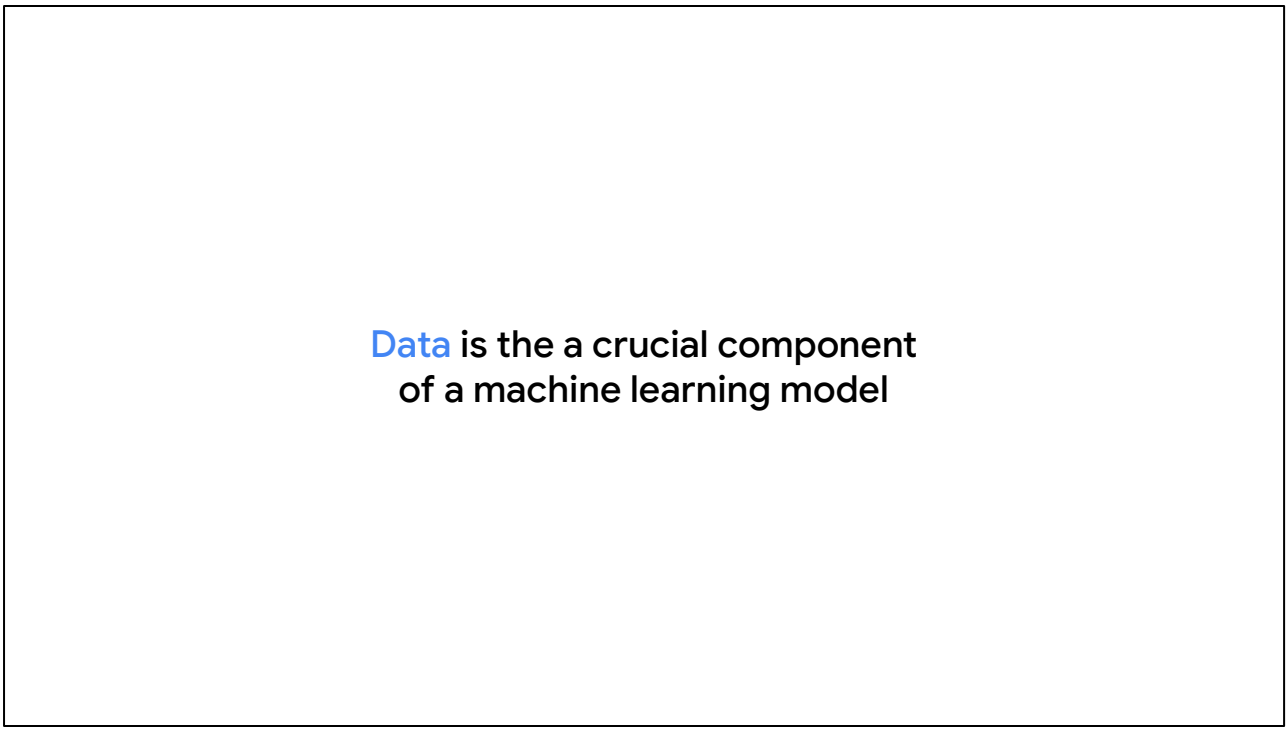
You will learn and practice this later after first mastering dataset manipulation!

Lab intro

TensorFlow Dataset API



In this lab, you'll learn how to train on large datasets that aren't going to fit in memory. Tensorflow has that API called Datasets, that can handle this, and feed your model while loading the data progressively from disk. Practice with the lab yourself by reading through the notebook comments, and executing the code cells. Edit the code to ensure that you understand what each part does, and then come back and watch the wrap up video, will give you a complete walk-through. Note that you have more than one attempt to try this lab, so don't worry if you run out of time on your first attempt. Good luck.



Data is the a crucial component of a machine learning model. Collecting the right data is not enough. You also need to make sure you put the right processes in place to clean, analyze and transform the data, as needed, so that the model can take the most signal of it as possible.

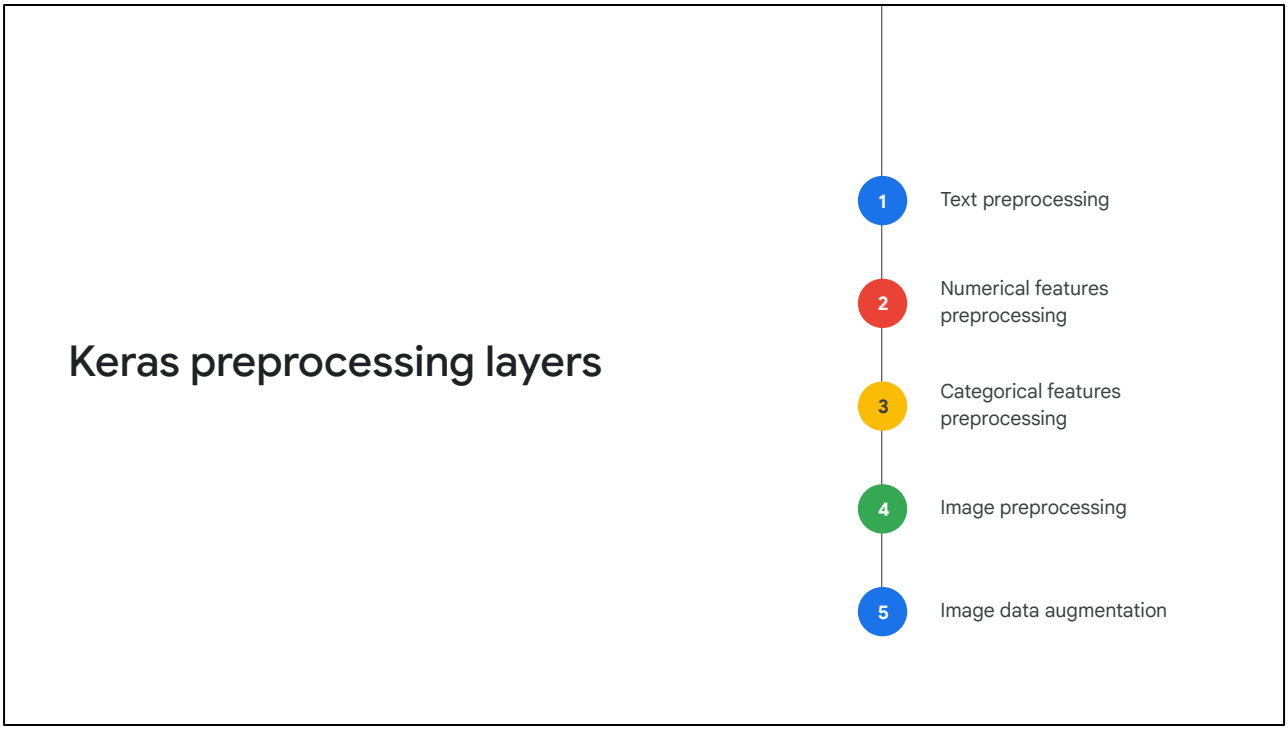
In this lesson, we examine scaling data preprocessing with tf.data and Keras preprocessing layers.

Data preprocessing

- You can build and export end-to-end models that accept raw images or raw structured data as input.
- Models handle feature normalization or feature value indexing on their own.

Combined with TensorFlow, the Keras preprocessing layers API allows TensorFlow developers to build Keras-native input processing pipelines.

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 and models that handle feature normalization or feature value indexing on their own.



The available preprocessing Layers include: Text preprocessing, Numerical features preprocessing, Categorical features preprocessing, Image preprocessing, and Image data augmentation.

Let’s describe a few of them,

Text features preprocessing

```
tf.keras.layers.TextVectorization(
    max_tokens=None,
    standardize="lower_and_strip_punctuation",
    split="whitespace",
    ngrams=None,
    output_mode="int",
    output_sequence_length=None,
    pad_to_max_tokens=False,
    vocabulary=None,
    **kwargs
)
```

Text vectorization layer

tf.keras.layers.TextVectorization:

turns raw strings into an encoded representation that can be read by an Embedding layer or Dense layer.

beginning with text features preprocessing. The **tf.keras.layers.TextVectorization** layer has basic options for managing text in a Keras model. It transforms a batch of strings, where one example equals one string, into one of two things: either a list of token indices, where one example equals 1D tensor of integer token indices; or a dense representation, with one example equal to 1D tensor of float values representing data about the example's tokens. Essentially, the TextVectorization layer turns raw strings into an encoded representation that can be read by an Embedding layer or Dense layer.

If desired, you can call this layer's adapt() method on a dataset. When this layer is adapted, adapt() will analyze the dataset, determine the frequency of individual string values, and create a 'vocabulary' from them. This vocabulary can have unlimited size or be capped, depending on the configuration options for the layer. If there are more unique values in the input than the maximum vocabulary size, the most frequent terms will be used to create the vocabulary.

Numerical features preprocessing

```
tf.keras.layers.Normalization(axis=-1, mean=None, variance=None, **kwargs)
```

Normalization class

Feature-wise normalization of the data

tf.keras.layers.Normalization:

performs feature-wise normalization of input features.

The numerical features preprocessing layer will coerce its inputs into a distribution centered around 0 with standard deviation 1. It accomplishes this by precomputing the mean and variance of the data and calling (input - mean) / sqrt(var) at runtime.

Essentially, the Normalization layer performs feature-wise normalization of input features.

Numerical preprocessing

```
tf.keras.layers.Discretization(
   bin_boundaries=None, num_bins=None, epsilon=0.01, **kwargs
)
```

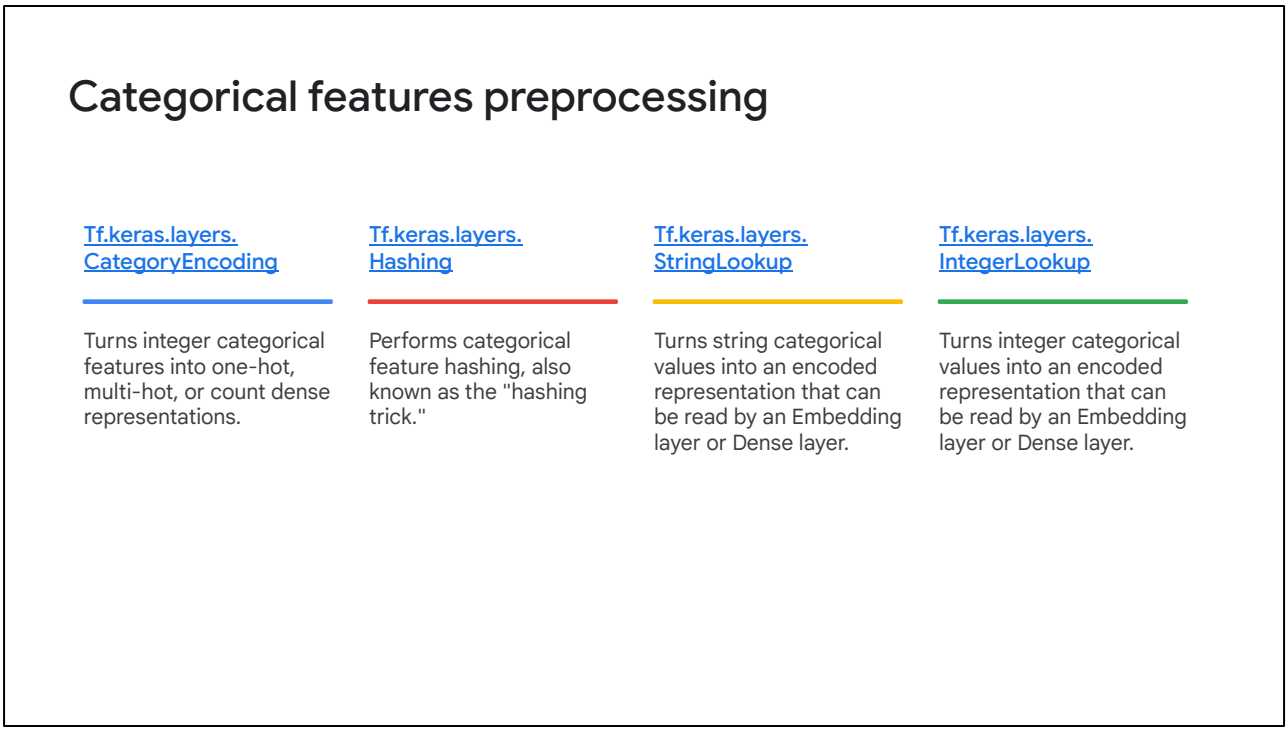
Discretization class

Buckets data into discrete ranges.

tf.keras.layers.Discretization:

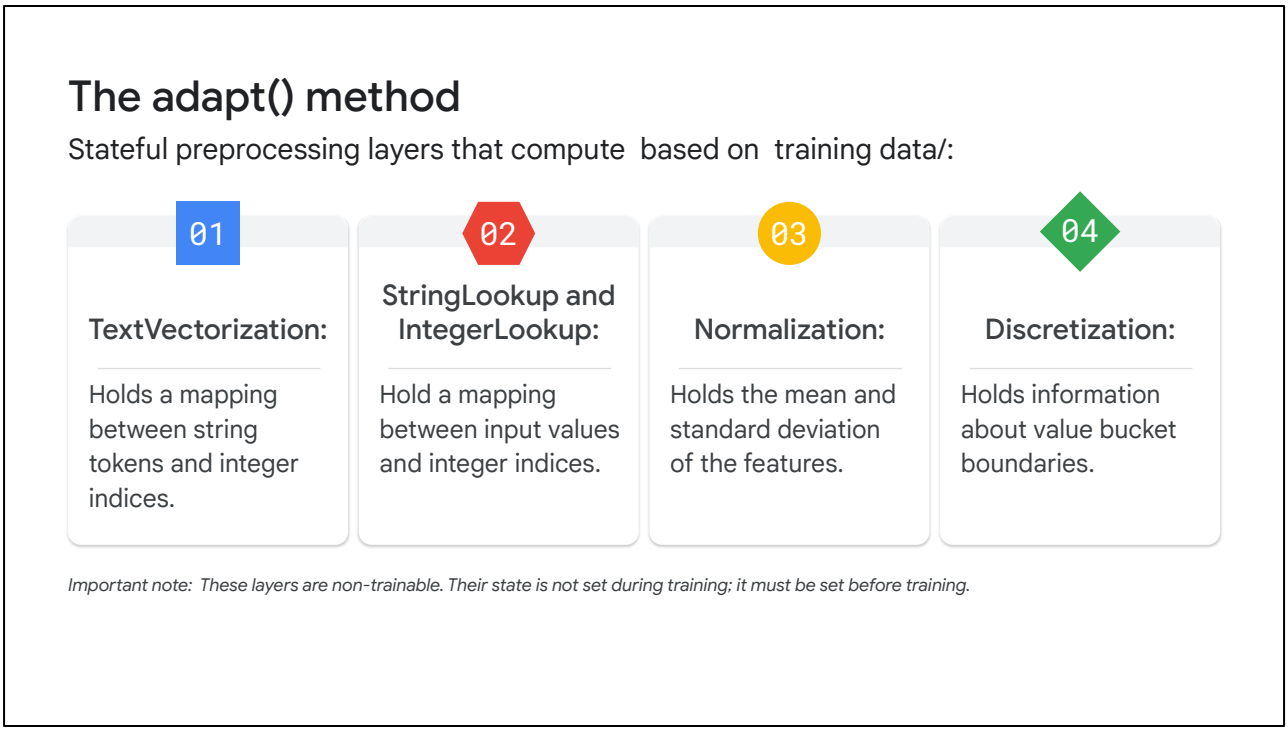
turns continuous numerical features into bucket data with discrete ranges.

The tf.keras.discretization layer will place each element of its input data into one of several contiguous ranges and output an integer index that indicates which range each element was placed in. Essentially, this layer turns continuous numerical features into bucket data with discrete ranges.



You can use a variety of different layers to preprocess categorical features:

- **tf.keras.layers.CategoryEncoding** turns integer categorical features into one-hot, multi-hot, or count dense representations.
- **tf.keras.layers.Hashing** performs categorical feature hashing, also known as the "hashing trick."
- **tf.keras.layers.StringLookup** turns string categorical values into an encoded representation that can be read by an Embedding layer or a Dense layer.
- **tf.keras.layers.IntegerLookup** turns integer categorical values into an encoded representation that can be read by an Embedding layer or Dense layer.



Some preprocessing layers support multiple states that are computed based on the dataset at a given time. These stateful layers include:

- **TextVectorization** (which holds a map between string tokens and integer indices).
- **StringLookup and IntegerLookup** (which holds a mapping between input values and integer indices).
- **Normalization** (which holds the mean and standard deviation of the features).
- **Discretization** (which holds information about value bucket boundaries).

Note, 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. Thus, the use of the adapt() method.

PetFinder Dataset

Column	Description	Feature Type	Data Type
Type	Type of animal (Dog, Cat)	Categorical	string
Age	Age of the pet	Numerical	integer
Breed1	Primary breed of the pet	Categorical	string
Color1	Color 1 of pet	Categorical	string
Color2	Color 2 of pet	Categorical	string
MaturitySize	Size at maturity	Categorical	string
FurLength	Fur length	Categorical	string
Vaccinated	Pet has been vaccinated	Categorical	string
Sterilized	Pet has been sterilized	Categorical	string
Health	Health Condition	Categorical	string
Fee	Adoption Fee	Numerical	integer
Description	Profile write-up for this pet	Text	string
PhotoAmt	Total uploaded photos for this pet	Numerical	integer
AdoptionSpeed	Speed of adoption	Classification	integer

To demonstrate the use of preprocessing layers, we'll use a simplified version of the PetFinder dataset, available as a ready-to-use TensorFlow dataset.

There are several thousand rows in the CSV. Each row describes a pet, and each column describes an attribute. You will use this information to predict whether the pet will be adopted.

Classify structured data using Keras preprocessing layers: Dataset

Column	Description	Feature Type	Data Type
Type	Type of animal (Dog, Cat)	Categorical	string
Age	Age of the pet	Numerical	integer
Breed1	Primary breed of the pet	Categorical	string
Color1	Color 1 of pet	Categorical	string
Color2	Color 2 of pet	Categorical	string
MaturitySize	Size at maturity	Categorical	string
FurLength	Fur length	Categorical	string
Vaccinated	Pet has been vaccinated	Categorical	string
Sterilized	Pet has been sterilized	Categorical	string
Health	Health Condition	Categorical	string
Fee	Adoption Fee	Numerical	integer
Description	Profile write-up for this pet	Text	string
PhotoAmt	Total uploaded photos for this pet	Numerical	integer
AdoptionSpeed	Speed of adoption	Classification	integer

The Feature Types include categorical, numerical, text, and classification.

Import the preprocessing library

```
[]   !pip install -q sklearn

[]   import numpy as np
     import pandas as pd
     import tensorflow as tf

     from sklearn.model_selection import train_test_split
     from tensorflow.keras import layers
     from tensorflow.keras.layers.experimental import preprocessing
```

To get started, you'll need to import the Keras preprocessing layer.

Categorical columns

Column	Description	Feature Type	Data Type
Type	Type of animal (Dog, Cat)	Categorical	string

```
def get_category_encoding_layer(name, dataset, dtype, max_tokens=None):
    # Create a StringLookup layer which will turn strings into integer indices
    if dtype =='string':
        index = preprocessing.StringLookup(max_tokens=max_tokens)
    else:
        index = preprocessing.IntegerLookup(max_tokens=max_tokens)

    # Prepare a Dataset that only yields our feature
    feature_ds = dataset.map(lambda x, y: x[name])

    # Learn the set of possible values and assign them a fixed integer index.
    index.adapt(feature_ds)

    # Create a Discretization for our integer indices.
    encoder = preprocessing.CategoryEncoding(num_tokens=index.vocabulary_size())

    # Apply one-hot encoding to our indices. The lambda function captures the
    # layer so we can use them, or include them in the functional model later.
    return lambda feature: encoder(index(feature))
```

In this dataset, pet type is represented as a string such as 'Dog' or 'Cat.' You cannot feed strings directly to a model. The preprocessing layer takes care of representing strings as a one-hot vector.

Recall that the **get_category_encoding_layer** function returns a layer that maps values from a vocabulary to integer indices and one-hot encodes the features.

Categorical columns

tf.keras.layers.StringLookup:
Turns string categorical values into an encoded representation that can be read by an Embedding layer or Dense layer.

In this dataset, Type is represented as a string (e.g., 'Dog' or 'Cat'). You cannot feed strings directly to a model. The preprocessing layer takes care of representing strings as a one-hot vector.

Column	Description	Feature Type	Data Type
Type	Type of animal (Dog, Cat)	Categorical	string

```
def get_category_encoding_layer(name, dataset, dtype, max_tokens=None):
  # Create a StringLookup layer which will turn strings into integer indices
  if dtype =='string':
      index = preprocessing.StringLookup(max_tokens=max_tokens)
  else:
      index = preprocessing.IntegerLookup(max_tokens=max_tokens)

  # Prepare a Dataset that only yields our feature
  feature_ds = dataset.map(lambda x, y: x[name])

  # Learn the set of possible values and assign them a fixed integer index.
  index.adapt(feature_ds)

  # Create a Discretization for our integer indices.
  encoder = preprocessing.CategoryEncoding(num_tokens=index.vocabulary_size())

  # Apply one-hot encoding to our indices. The lambda function captures the
  # layer so we can use them, or include them in the functional model later.
  return lambda feature: encoder(index(feature))
```

tf.keras.layers.StringLookup turns string categorical values into an encoded representation that can be read by an Embedding layer or Dense layer. By mapping strings from a vocabulary to integer indices, StringLookup turns string categorical values into integer indices.

Categorical columns

tf.keras.layers.IntegerLookup:

Turns integer categorical values into an encoded representation that can be read by an Embedding layer or Dense layer.

```
def get_category_encoding_layer(name, dataset, dtype, max_tokens=None):
    # Create a StringLookup layer which will turn strings into integer indices
    if dtype =='string':
        index = preprocessing.StringLookup(max_tokens=max_tokens)
    else:
        index = preprocessing.IntegerLookup(max_tokens=max_tokens)

    # Prepare a Dataset that only yields our feature
    feature_ds = dataset.map(lambda x, y: x[name])

    # Learn the set of possible values and assign them a fixed integer index.
    index.adapt(feature_ds)

    # Create a Discretization for our integer indices.
    encoder = preprocessing.CategoryEncoding(num_tokens=index.vocabulary_size())

    # Apply one-hot encoding to our indices. The lambda function captures the
    # layer so we can use them, or include them in the functional model later.
    return lambda feature: encoder(index(feature))
```

tf.keras.layers.IntegerLookup also turns integer categorical values into an encoded representation that can be read by an Embedding layer or Dense layer. Simply, IntegerLookup maps integers from a vocabulary to integer indices.

Numeric columns

For each of the Numeric features, you will use a Normalization() layer to ensure that the mean of each feature is 0 and its standard deviation is 1.

Recall that the get_normalization_layer function returns a layer that applies feature-wise normalization to numerical features.

```
def get_normalization_layer(name, dataset):
  # Create a Normalization layer for our feature.
  normalizer = preprocessing.Normalization(axis=None)

  # Prepare a Dataset that only yields our feature.
  feature_ds = dataset.map(lambda x, y: x[name])

  # Learn the statistics of the data.
  normalizer.adapt(feature_ds)

  return normalizer
```

the adapt() method

For each of the Numeric features, you'll use a Normalization() layer to make sure the mean of each feature is 0 and its standard deviation is 1. Recall that get_normalization_layer function returns a layer that applies feature-wise normalization to numerical features.

The normalization preprocessing layer has an internal state that can be computed based on a sample of the training data. In this case, normalizer. adapt holds the mean and standard deviation of the feature.
Although *preprocessing.Normalization*() normalizes features with the feature means and the variances of the training dataset, these parameters are not part of the trainable parameters. Instead, you *adapt* the normalization layer to the training samples (*data*). This method will calculate the means and the variances automatically. Note that the adapt() method takes either a Numpy array or a tf.data.Dataset object.

Simple normalization example

```
#Load some data
(x_train, y_train), _ = keras.datasets.cifar10.load_data()
x_train = x_train.reshape((len(x_train), -1))
input_shape = x_train.shape[1:]
classes = 10

#Create a Normalization layer and set its internal state using the training data
normalizer = layers.Normalization()
normalizer.adapt(x_train)

#Create a model that include the normalization layer
inputs = keras.Input(shape=input_shape)
x = normalizer(inputs)
outputs = layers.Dense(classes, activation="softmax")(x)
model = keras.Model(inputs, outputs)

#Train the model
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
model.fit(x_train, y_train)
```

Let's examine a simple pipeline that uses the normalization layer.

Create a normalization layer

```
#Load some data
(x_train, y_train), _ = keras.datasets.cifar10.load_data()
x_train = x_train.reshape((len(x_train), -1))
input_shape = x_train.shape[1:]
classes = 10

#Create a Normalization layer and set its internal state using the training data
normalizer = layers.Normalization()
normalizer.adapt(x_train)

#Create a model that include the normalization layer
inputs = keras.Input(shape=input_shape)
x = normalizer(inputs)
outputs = layers.Dense(classes, activation="softmax")(x)
model = keras.Model(inputs, outputs)

#Train the model
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
model.fit(x_train, y_train)
```

Here the normalizer has instantiated its internal state, which is set using the adapt method. Remember that although preprocessing.Normalization() normalizes features with the feature means and the variances of the training dataset, these parameters are not part of the trainable parameters. Instead, you need to *adapt* the normalization layer to the training samples (*data*) in order to calculate the means and the variances automatically.

Create a model with a normalization layer

```
#Load some data
(x_train, y_train), _ = keras.datasets.cifar10.load_data()
x_train = x_train.reshape((len(x_train), -1))
input_shape = x_train.shape[1:]
classes = 10

#Create a Normalization layer and set its internal state using the training data
normalizer = layers.Normalization()
normalizer.adapt(x_train)

#Create a model that include the normalization layer
inputs = keras.Input(shape=input_shape)
x = normalizer(inputs)
outputs = layers.Dense(classes, activation="softmax")(x)
model = keras.Model(inputs, outputs)

#Train the model
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
model.fit(x_train, y_train)
```

Here, you simply need to create a model that includes the normalization layer.

Model layers versus preprocessing dataset

(Preprocessing data before the model or inside the model)

Option 1

```
inputs = kera.Input(shape=input_shape)
x = preprocessing_layer(inputs)
outputs =  rest_of_the model(x)
model = keras.Model(inputs, outputs)
```

Option 2

```
dataset = dataset.map(
   lambda x, y: (preprocessing_layer(x),y))
```

Keras preprocessing provides two different options in applying the data transformation.

Preprocessing as part of the model

Option 1

```
inputs = kera.Input(shape=input_shape)
x = preprocessing_layer(inputs)
outputs =  rest_of_the model(x)
model = keras.Model(inputs, outputs)
```

Option 2

```
dataset = dataset.map(
   lambda x, y: (preprocessing_layer(x),y))
```

In option 1, the preprocessing layer is part of the model. It is part of the model computational graph that can be optimized and executed on a device like a GPU. This is the best option for the *Normalization* layer and all image preprocessing and data augmentation layers if GPUs are available.

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

Preprocessing will happen on the device synchronously with the rest of the model execution.

```
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 the device synchronously with the rest of the model execution, which means that it will benefit from GPU acceleration.

Preprocessing before the model

Option 1

```
inputs = kera.Input(shape=input_shape)
x = preprocessing_layer(inputs)
outputs =  rest_of_the model(x)
model = keras.Model(inputs, outputs)
```

Option 2

```
dataset = dataset.map(
   lambda x, y: (preprocessing_layer(x),y))
```

Option 2 uses dataset.map to convert data in the dataset. Data augmentation will happen asynchronously on the CPU and is non-blocking. Its key focus is to take advantage of multiple threading in the CPU.

Option 2:
Apply it to your tf.data.Dataset

Preprocessing will happen on the CPU asynchronously and will be buffered before going into the model.

```
dataset = dataset.map(lambda x,y: (preprocessing_layer(x),y))
dataset = dataset.prefetch(tf.data.AUTOTUNE)
model.fit(dataset,...)
```

When running on a TPU, you should always place preprocessing layers in the tf.data pipeline (with the exception of Normalization and Rescaling, which run well on TPU and are commonly used as the first layer in an image model).

With this option, your preprocessing will happen on the 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. Apply it to your tf.data.Dataset to obtain a dataset that yields batches of preprocessed data.

When running preprocessing on a TPU, you should always place preprocessing layers in the tf.data pipeline (with the exception of Normalization and Rescaling, which run fine on TPU and are commonly used as the first layer in an image model).

Benefits of doing preprocessing inside the model at inference time

Even if you choose option 2, you may later want to export an inference-only end-to-end model that will include the preprocessing layers.

The key benefits:

- Makes your model portable.
- Helps reduce the **training-serving skew.**

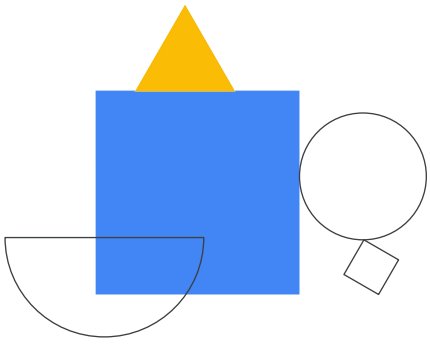
Even if you choose option 2, you may later want to export an inference-only end-to-end model that will include the preprocessing layers. As a key benefit, this makes your model portable and it helps reduce the training/serving skew.

When all data preprocessing is part of the model, others can load and use your model without having to be aware of how each feature is expected to be encoded and normalized. Your inference model can process raw images or raw structured data and will not require users of the model to be aware of details such as the tokenization scheme used for text, the indexing scheme used for categorical features, and whether image pixel values are normalized to [-1, +1] or to [0, 1].

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.

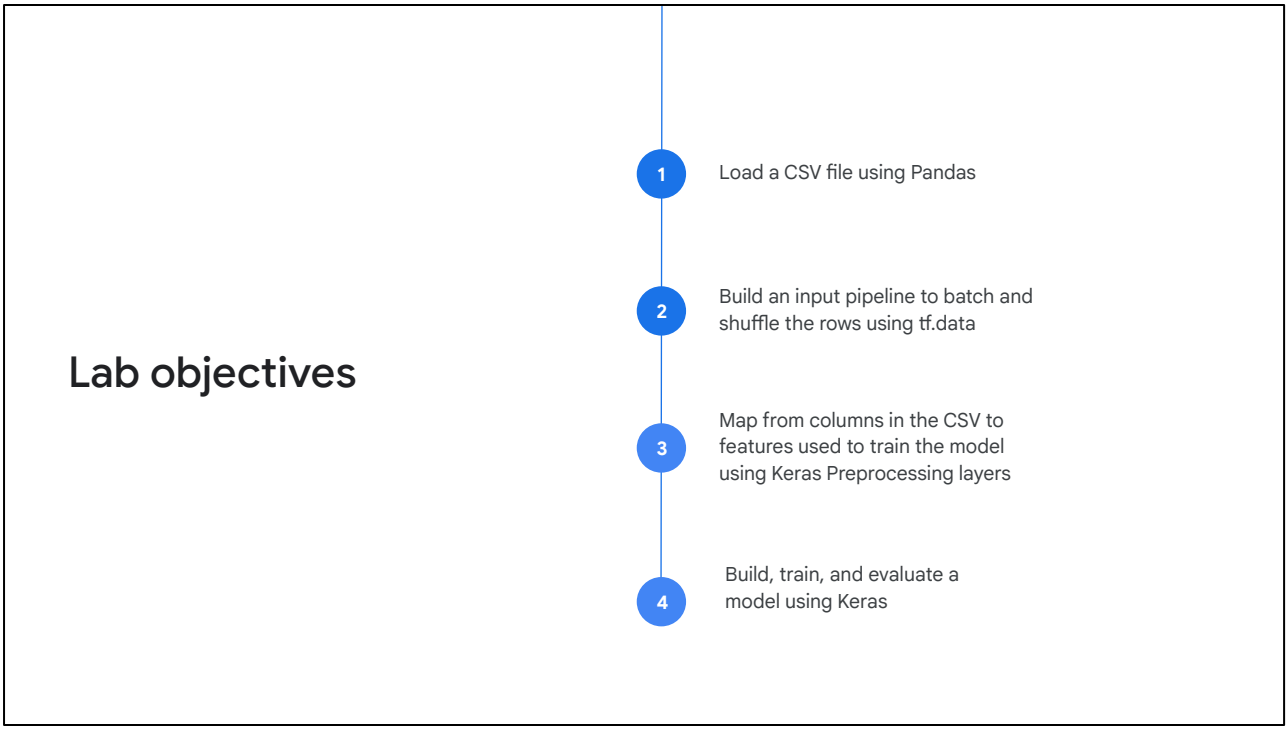
Lab intro

Classifying structured data using Keras Preprocessing Layers



This lab provides a hands-on introduction on how to Classify structured data using Keras Preprocessing Layers.

Link to Lab: [Classifying structured data using Keras Preprocessing Layers]



- You'll begin by loading a CSV file using Pandas.
- Next, you'll begin by building an input pipeline to batch and shuffle the rows using tf.data.
- Next, map from columns in the CSV to features used to train the model using Keras Preprocessing layers.
- Finally, you'll build, train, and evaluate a model using Keras.