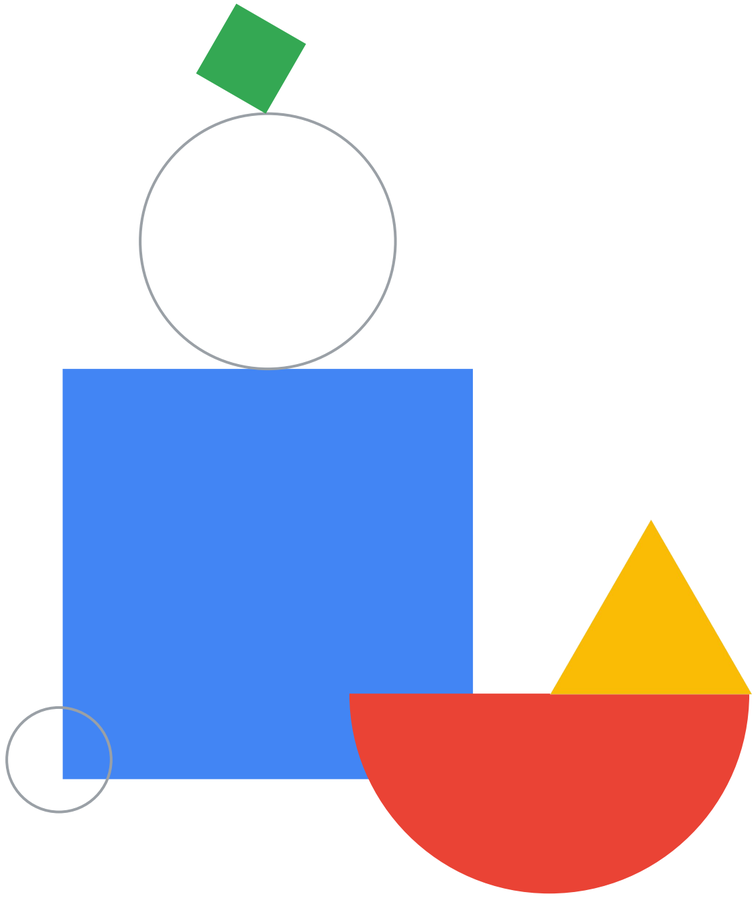
Google Cloud



TensorFlow on Google Cloud

In this course, you learn to ...

01 Create TensorFlow 2.x and Keras machine learning models

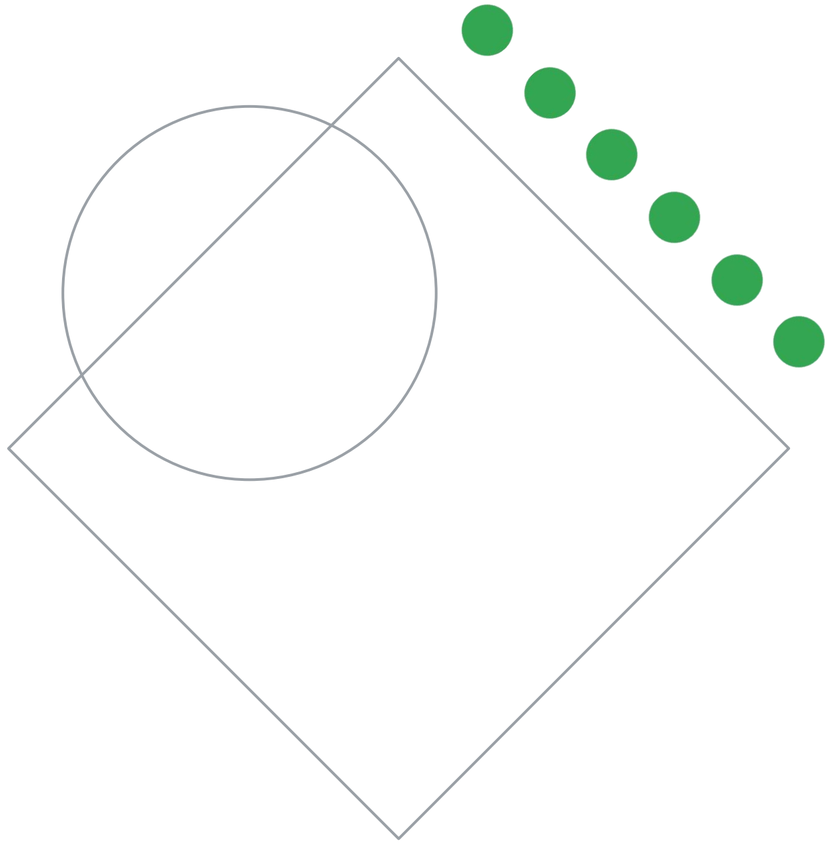
02 Describe Tensorflow 2.x key components

03 Use the tf.data library to manipulate data and large datasets

04 Use the Keras Sequential and Functional APIs for simple and advanced model creation

05 Train, deploy, and productionalize ML models at scale with Vertex AI

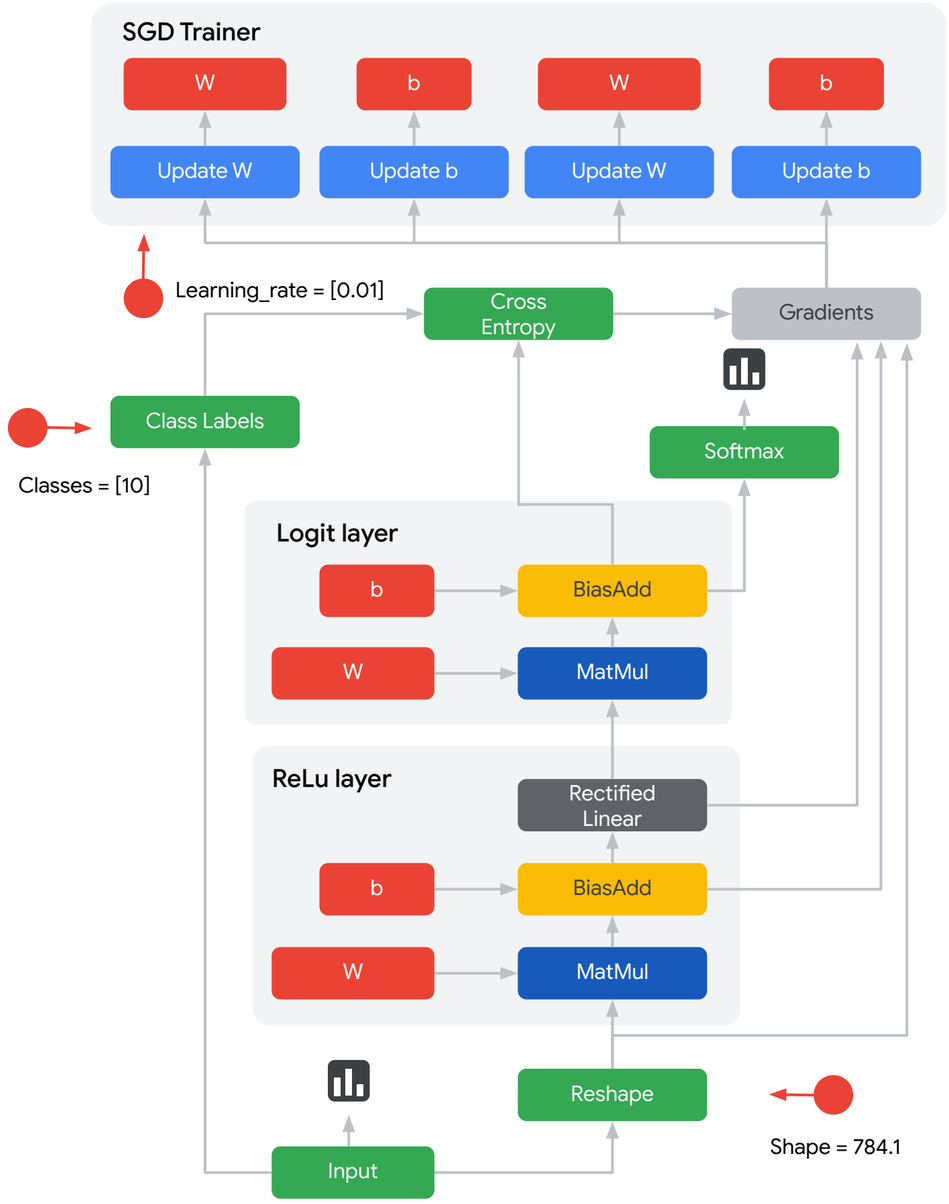
Introduction to the TensorFlow Ecosystem

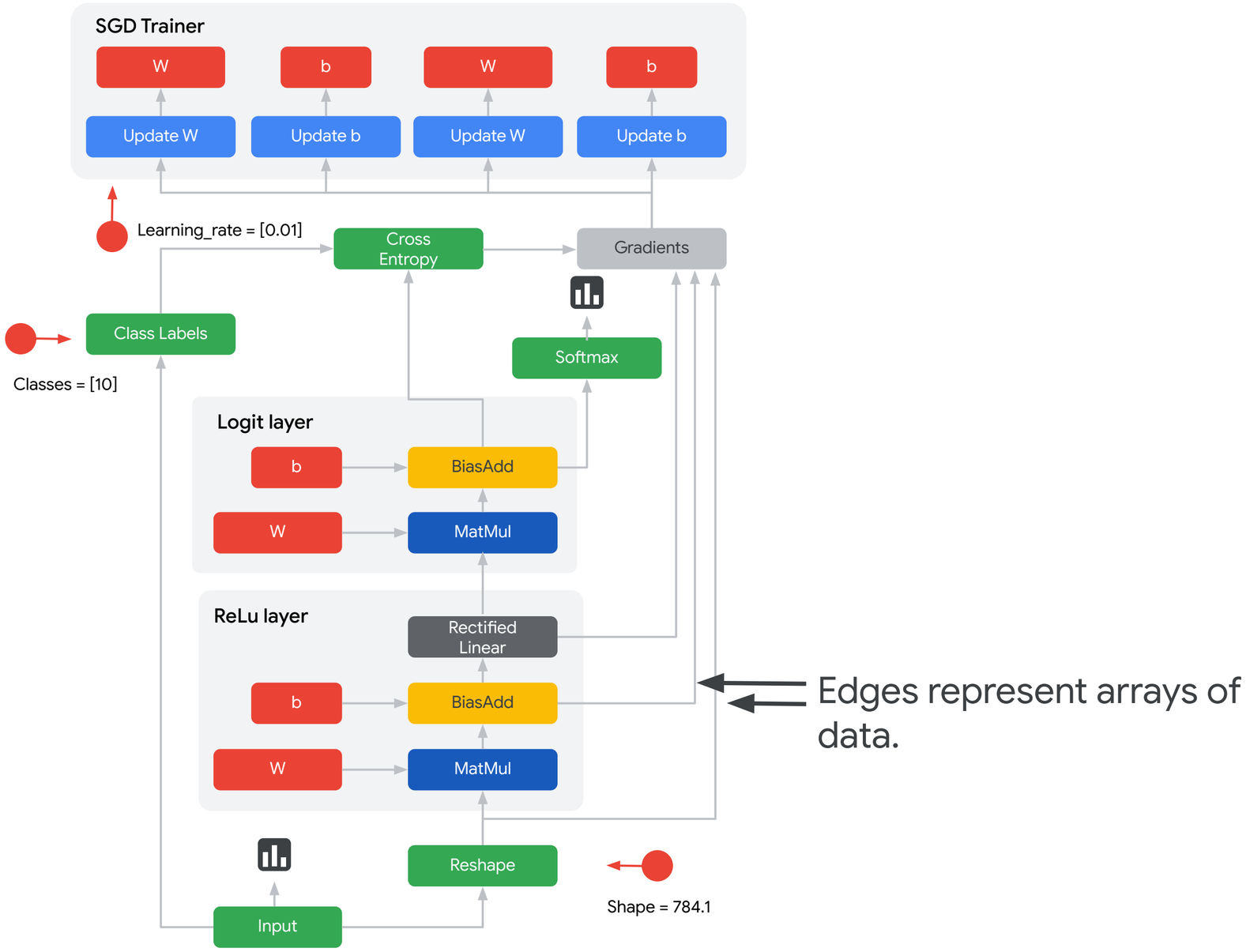


In this module, you learn to ...

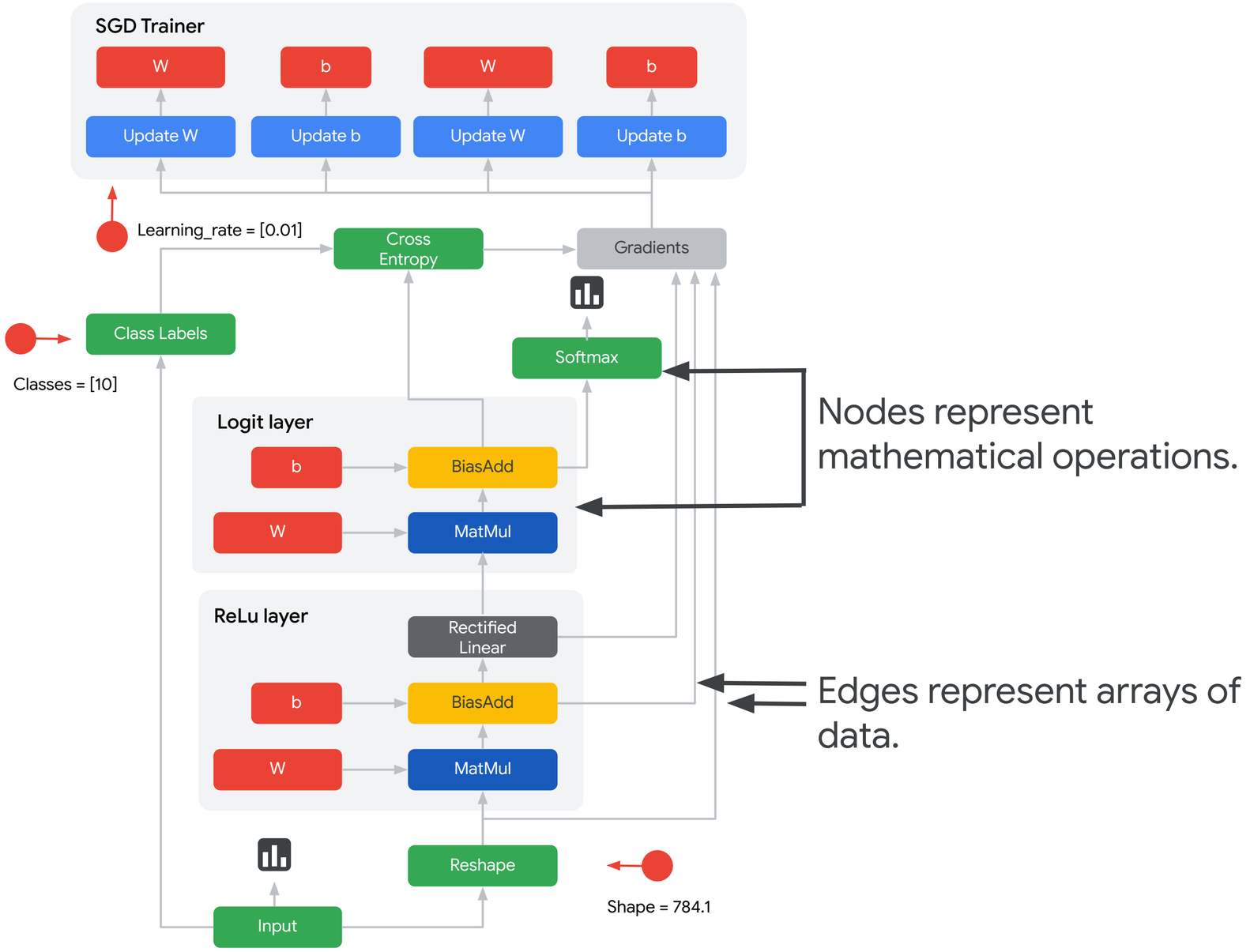
01	Recall the TensorFlow API hierarchy
02	Understand TensorFlow's building blocks: Tensors and operations
03	Write low-level TensorFlow programs

TensorFlow is an open-source, high-performance library for numerical computation that uses directed graphs





Edges represent arrays of data.


SGD Trainer
W
b
W
b
Update W
Update b
Update W
Update b
Learning_rate = [0.01]
Cross Entropy
Gradients
Class Labels
Classes = [10]
Softmax
Logit layer
b
BiasAdd
W
MatMul
ReLu layer
Rectified Linear
b
BiasAdd
W
MatMul
Reshape
Input
Shape = 784.1
Nodes represent mathematical operations.
Edges represent arrays of data.

A tensor is an N-dimensional array of data

Rank 0 Tensor

Rank 1 Tensor

Rank 2 Tensor

Rank 3 Tensor

Rank 4 Tensor

A tensor is an N-dimensional array of data

Rank 0 Tensor

scalar

Rank 1 Tensor

Rank 2 Tensor

Rank 3 Tensor

Rank 4 Tensor

A tensor is an N-dimensional array of data

Rank 0 Tensor

scalar

Rank 1 Tensor

vector

Rank 2 Tensor

Rank 3 Tensor

Rank 4 Tensor

A tensor is an N-dimensional array of data

Rank 0 Tensor

scalar

Rank 1 Tensor

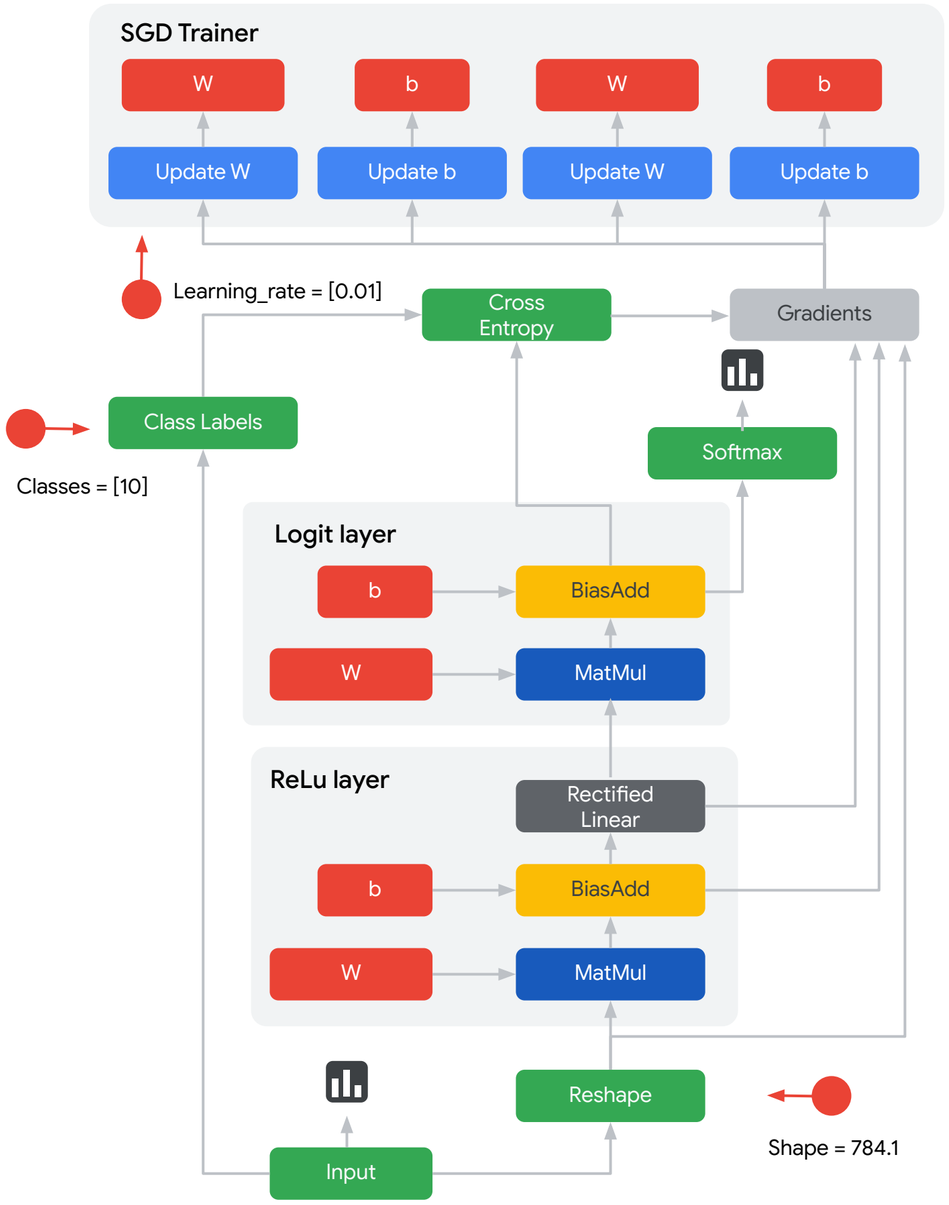
vector

Rank 2 Tensor

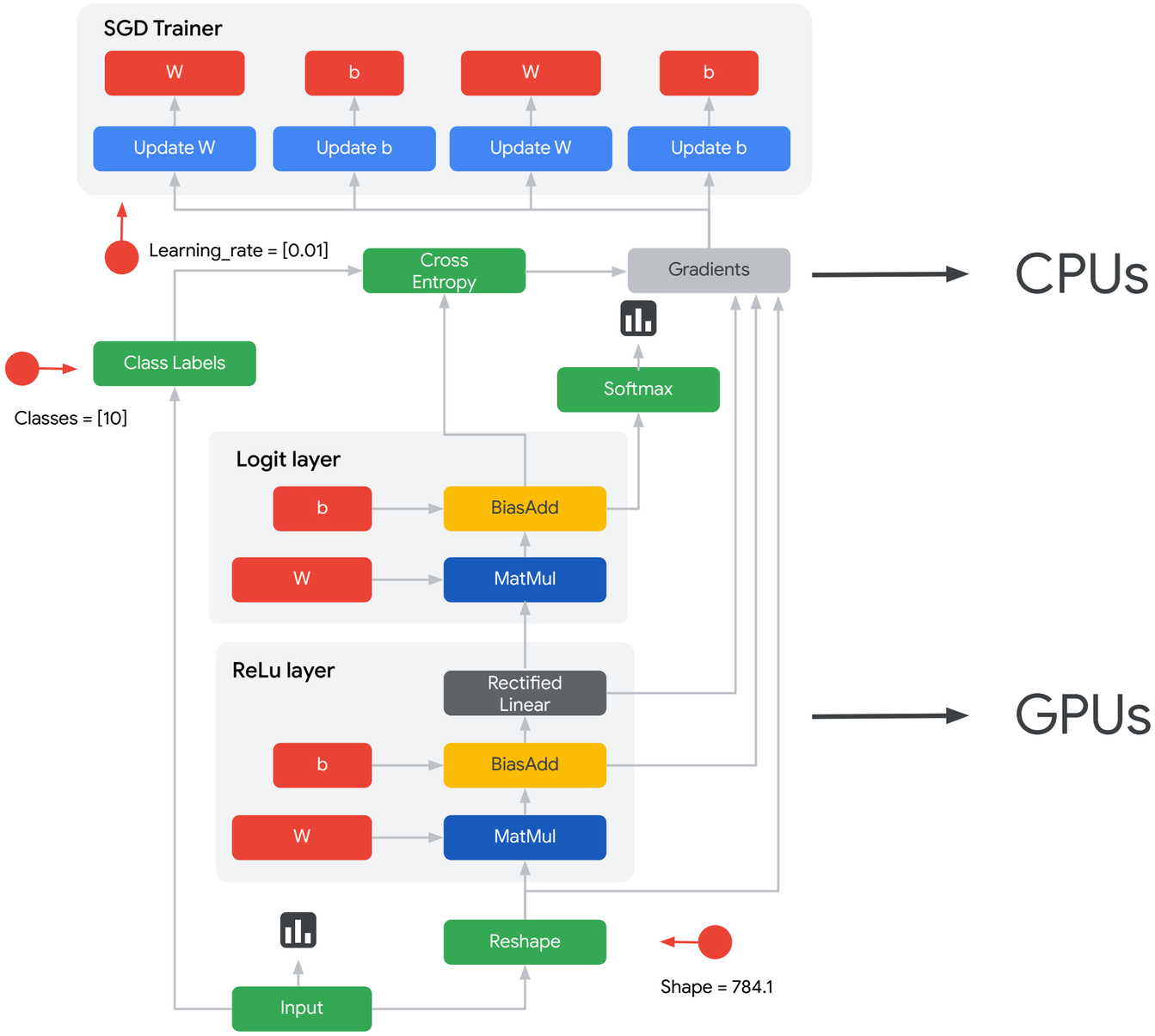
matrix

Rank 3 Tensor

Rank 4 Tensor



TensorFlow graphs are portable between different devices



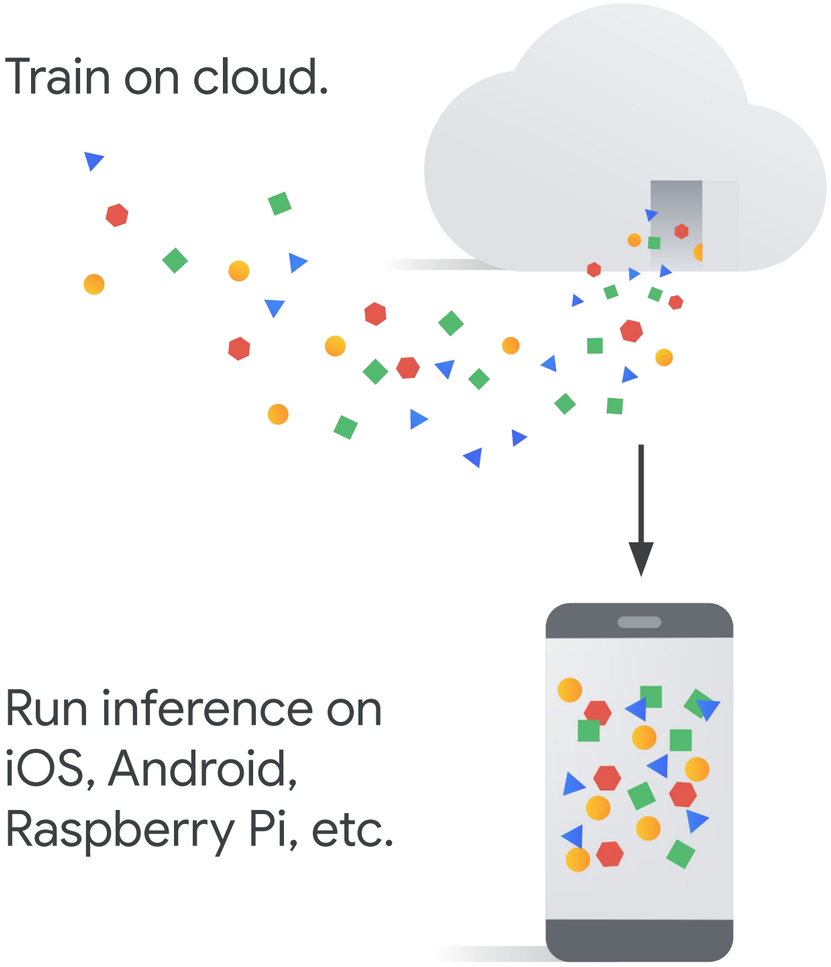
CPUs

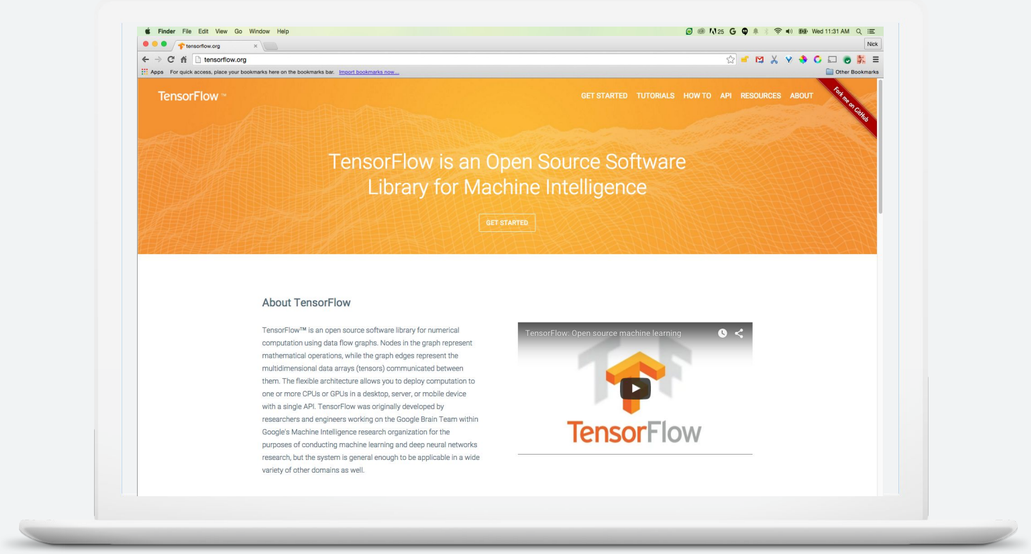
GPUs

TensorFlow graphs are portable between different devices

TensorFlow Lite provides on-device inference of ML models on mobile devices and is available for a variety of hardware.

Announcing TensorFlow Lite: https://developers.googleblog.com/2017/11/announcing-tensorflow-lite.html





TensorFlow is popular among both deep learning researchers and machine learning engineers.

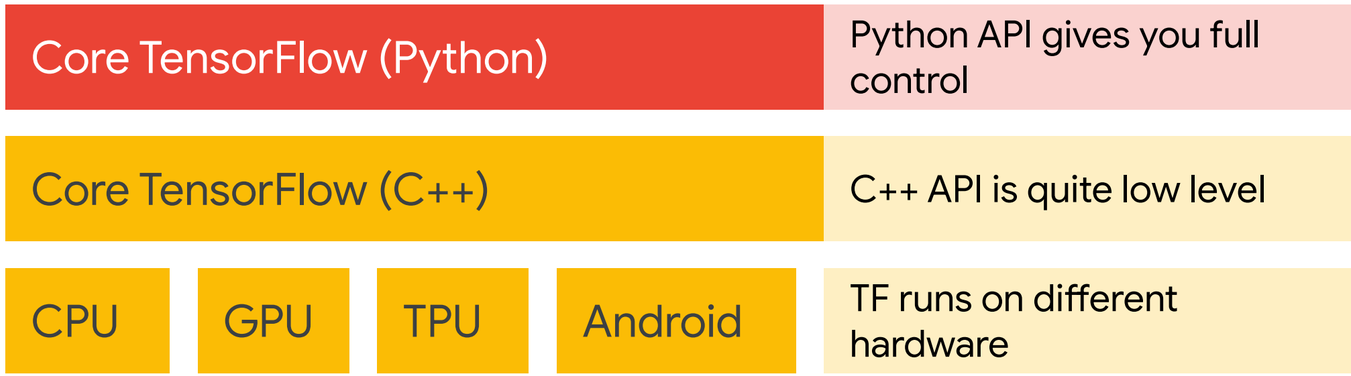
#1 repository for “machine learning”category on GitHub

TensorFlow API hierarchy

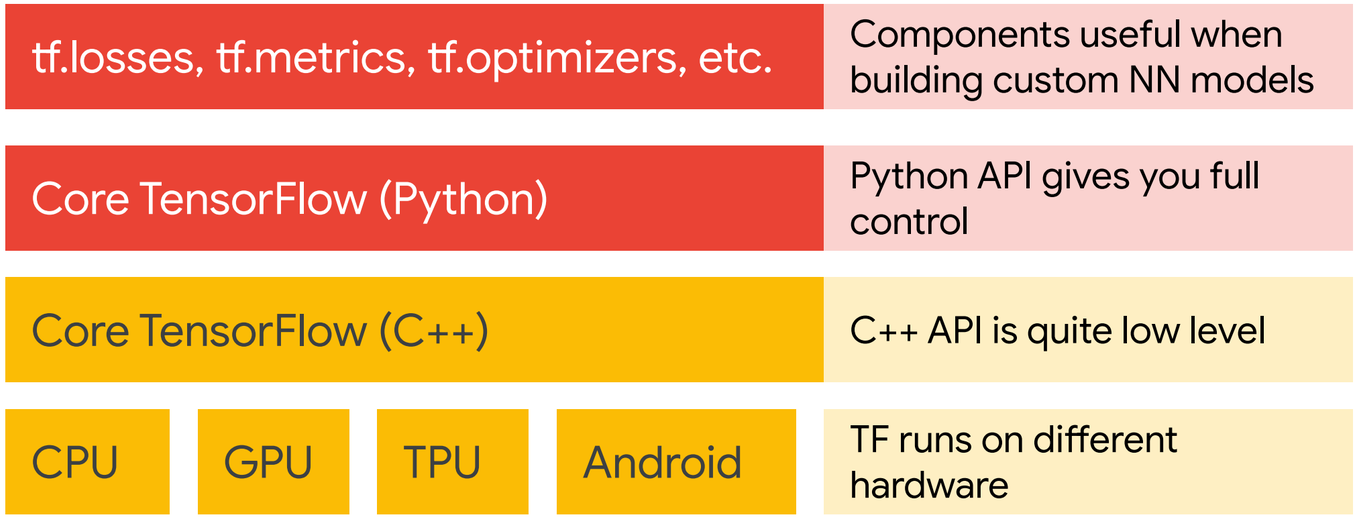
TensorFlow contains multiple abstraction layers.

TensorFlow contains multiple abstraction layers.

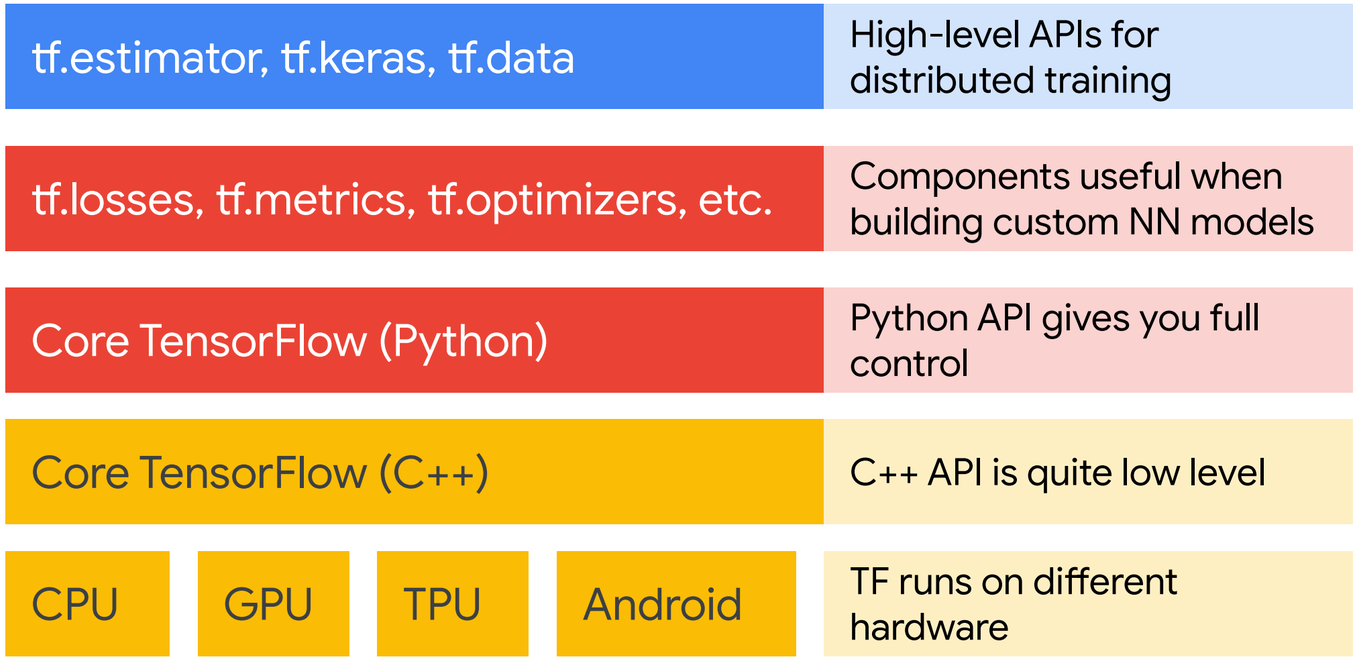
TensorFlow contains multiple abstraction layers.



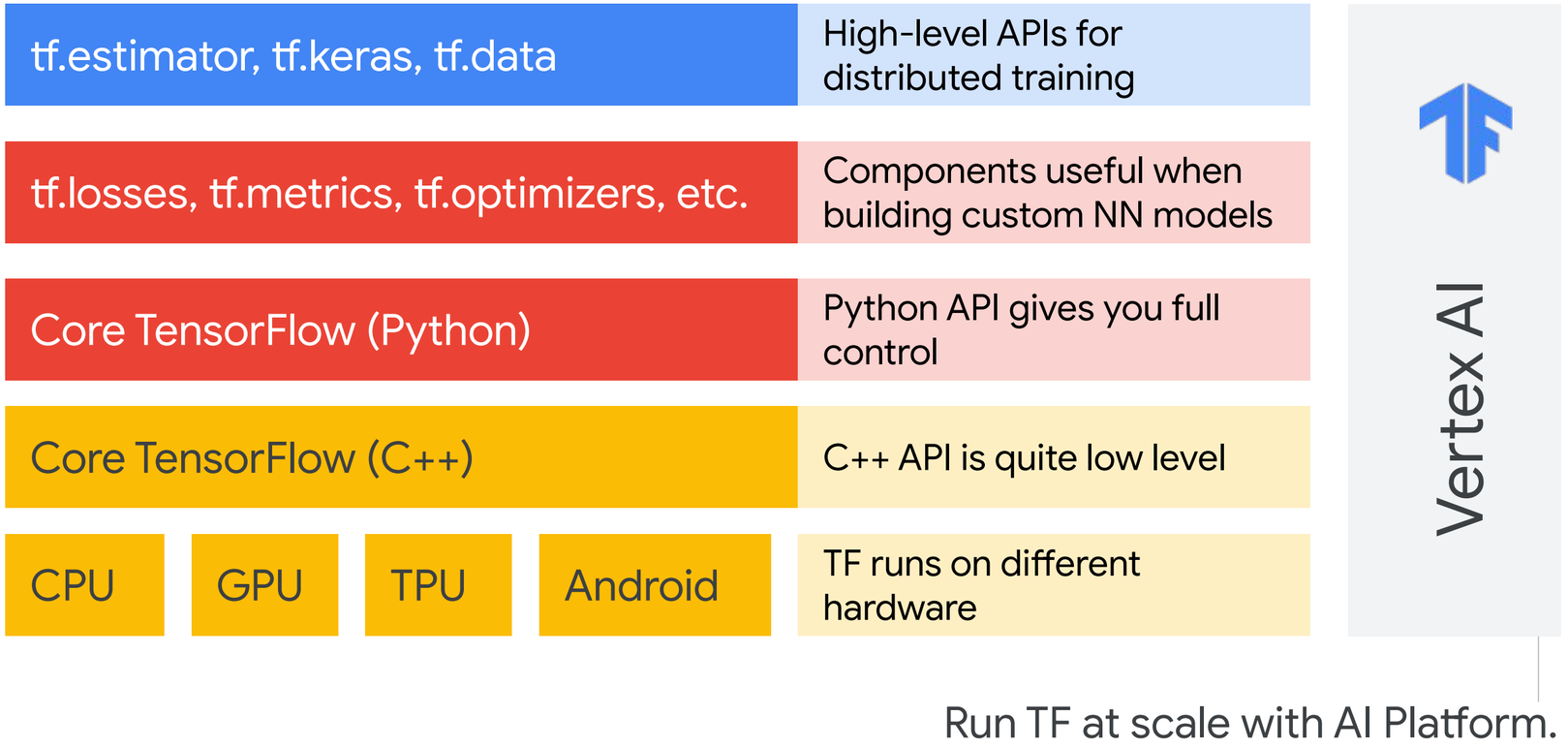
TensorFlow contains multiple abstraction layers.



TensorFlow contains multiple abstraction layers.

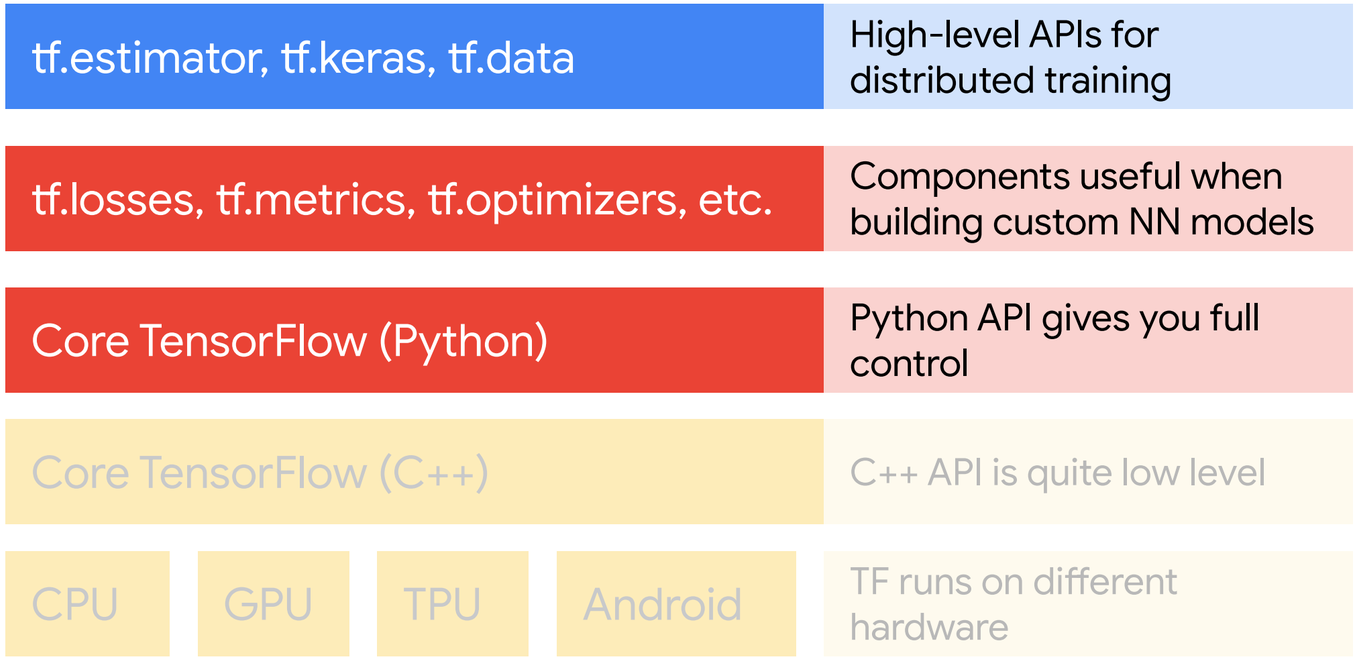


TensorFlow contains multiple abstraction layers.



Note that AI Platform is now Vertex AI.

TensorFlow toolkit hierarchy



TensorFlow tensors and variables

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
■	Scalar	0	`x = tf.constant(3)`	()

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

	3D Tensor	3	`tf.constant([[[3, 5, 7],[4, 6, 8]],`
			`[[1, 2, 3],[4, 5, 6]]`
			`])`
				(2, 2, 3)

A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

	3D Tensor	3	`tf.constant([[[3, 5, 7],[4, 6, 8]],`
			`[[1, 2, 3],[4, 5, 6]]`
			`])`
				(2, 2, 3)

	nD Tensor	n	`x1 = tf.constant([2, 3, 4])`
			`x2 = tf.stack([x1, x1])`
			`x3 = tf.stack([x2, x2, x2, x2])`
			`x4 = tf.stack([x3, x3])`
			`...`
				(3,)
				(2, 3)
				(4, 2, 3)
				(2, 4, 2, 3)

A tensor is an N-dimensional array of data

They behave like numpy n-dimensional arrays except that:

- tf.constant produces constant tensors
- tf.Variable produces tensors that can be modified

tf.constant produces constant tensors...

```
import tensorflow as tf

x = tf.constant([
    [3, 5, 7],
    [4, 6, 8]
])
```

Tensors can be sliced

```
import tensorflow as tf

x = tf.constant([
    [3, 5, 7],
    [4, 6, 8]
])

y = x[:, 1]

[5, 6]
```

Tensors can be reshaped

```
import tensorflow as tf

x = tf.constant([
    [3, 5, 7],
    [4, 6, 8]
])

y = tf.reshape(x, [3, 2])

[[3 5]
 [7 4]
 [6 8]
]
```

A variable is a tensor whose value can be changed...

```
import tensorflow as tf

# x <- 2
x = tf.Variable(2.0, dtype=tf.float32,
name='my_variable')
```

A variable is a tensor whose value can be changed...

tf.Variable will typically hold model weights
that need to be updated in a training loop.

```
import tensorflow as tf

# x <- 2
x = tf.Variable(2.0, dtype=tf.float32,name='my_variable')

# x <- 48.5
x.assign(45.8)

# x <- x + 4
x.assign_add(4)

# x <- x - 3
x.assign_sub(3)
```

A variable is a tensor whose value can be changed...

```
import tensorflow as tf

# w * x
w = tf.Variable([[1.], [2.]])
x = tf.constant([[3., 4.]])
tf.matmul(w, x)
```

GradientTape records operations for automatic differentiation

TensorFlow can compute the derivative of a function with respect to any parameter.

- the computation is recorded with GradientTape
- the function is expressed with TensorFlow ops only!

GradientTape records operations for automatic differentiation

```
def compute_gradients(X, Y, w0, w1):
    with tf.GradientTape() as tape:
        loss = loss_mse(X, Y, w0, w1)
    return tape.gradient(loss, [w0, w1])



w0 = tf.Variable(0.0)
w1 = tf.Variable(0.0)

dw0, dw1 = compute_gradients(X, Y, w0, w1)
```

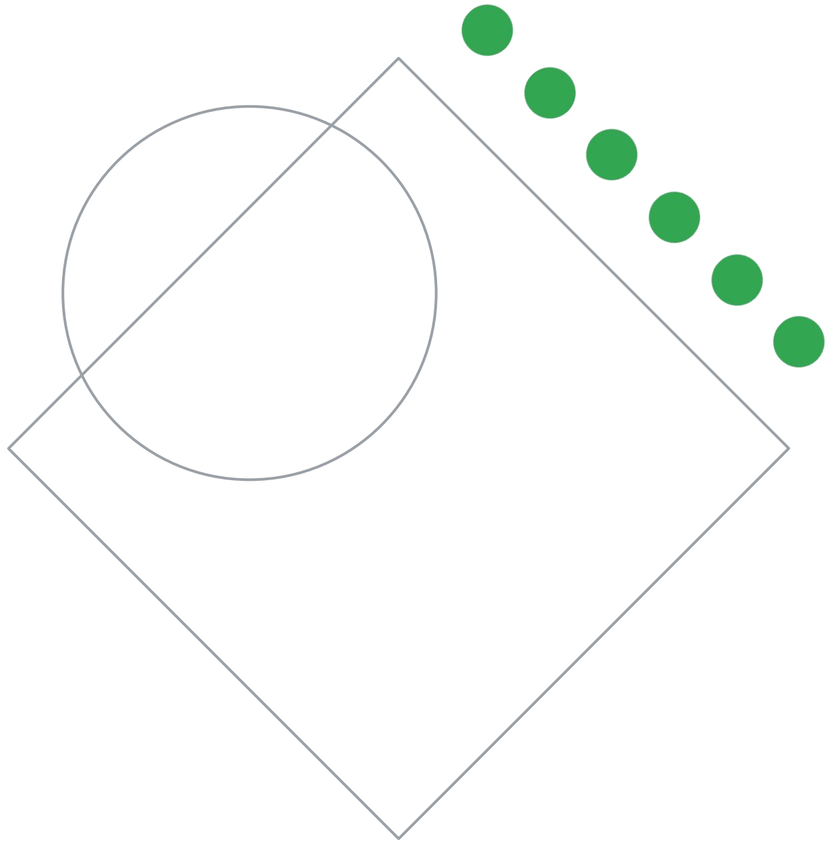
Record the computation with GradientTape when it’s executed (not when it’s defined!)

GradientTape records operations for automatic differentiation

```
def compute_gradients(X, Y, w0, w1):
    with tf.GradientTape() as tape:
        loss = loss_mse(X, Y, w0, w1)
    return tape.gradient(loss, [w0, w1])



w0 = tf.Variable(0.0)
w1 = tf.Variable(0.0)

dw0, dw1 = compute_gradients(X, Y, w0, w1)
```

Specify the function (loss) as well as the parameters you want to take the gradient with respect to ([w0, w1])

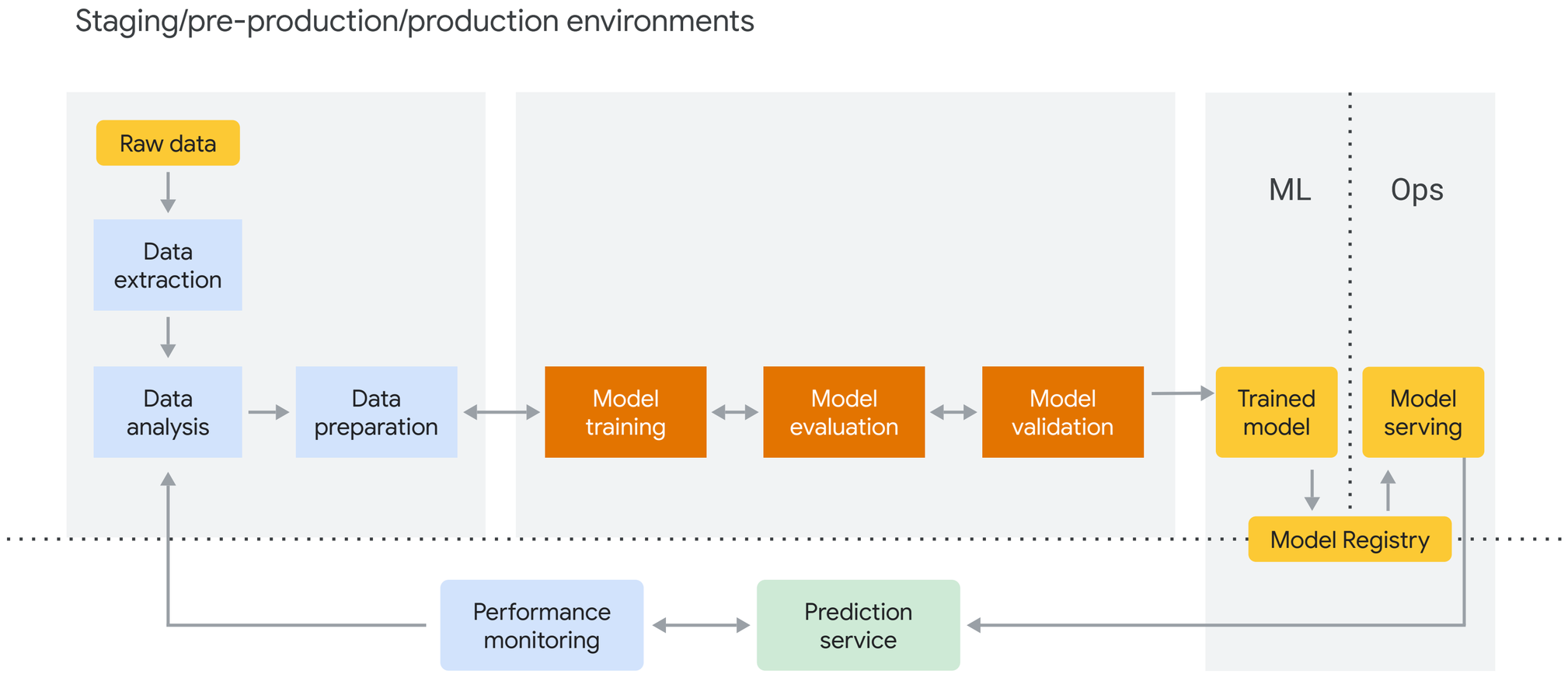
Design and Build an Input Data Pipeline



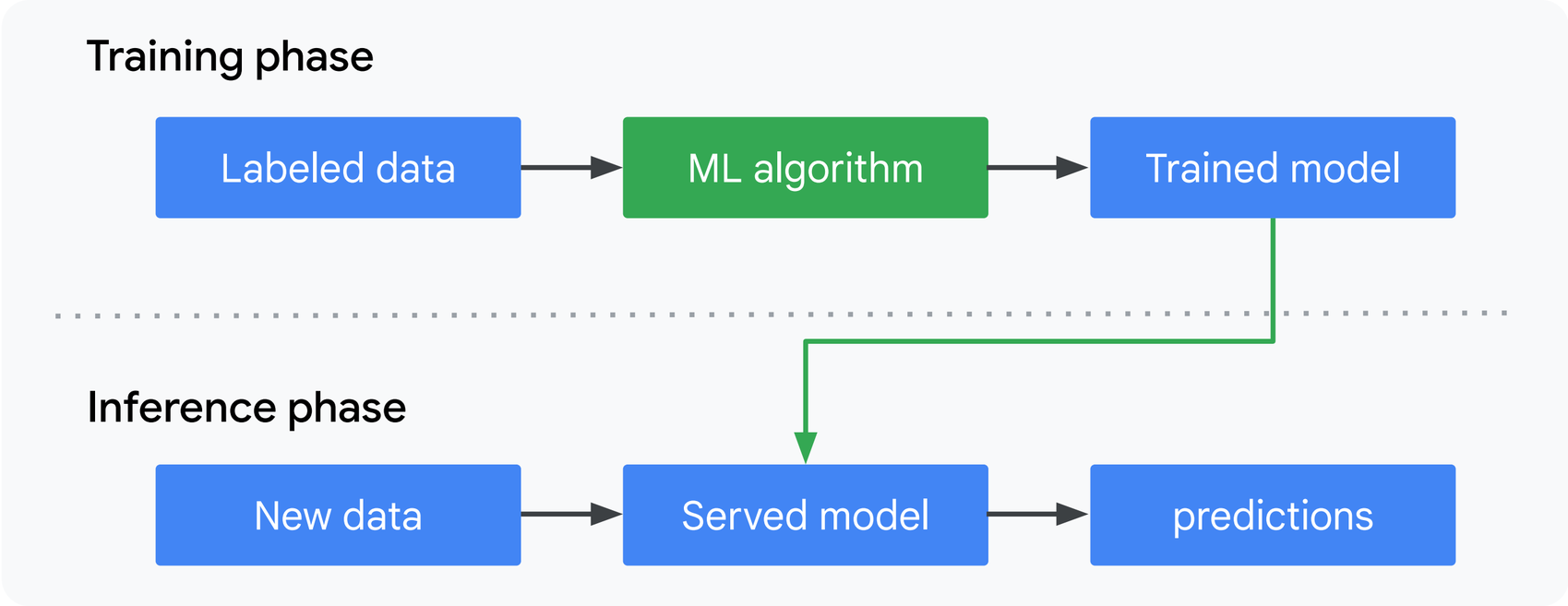
In this module, you learn to ...

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

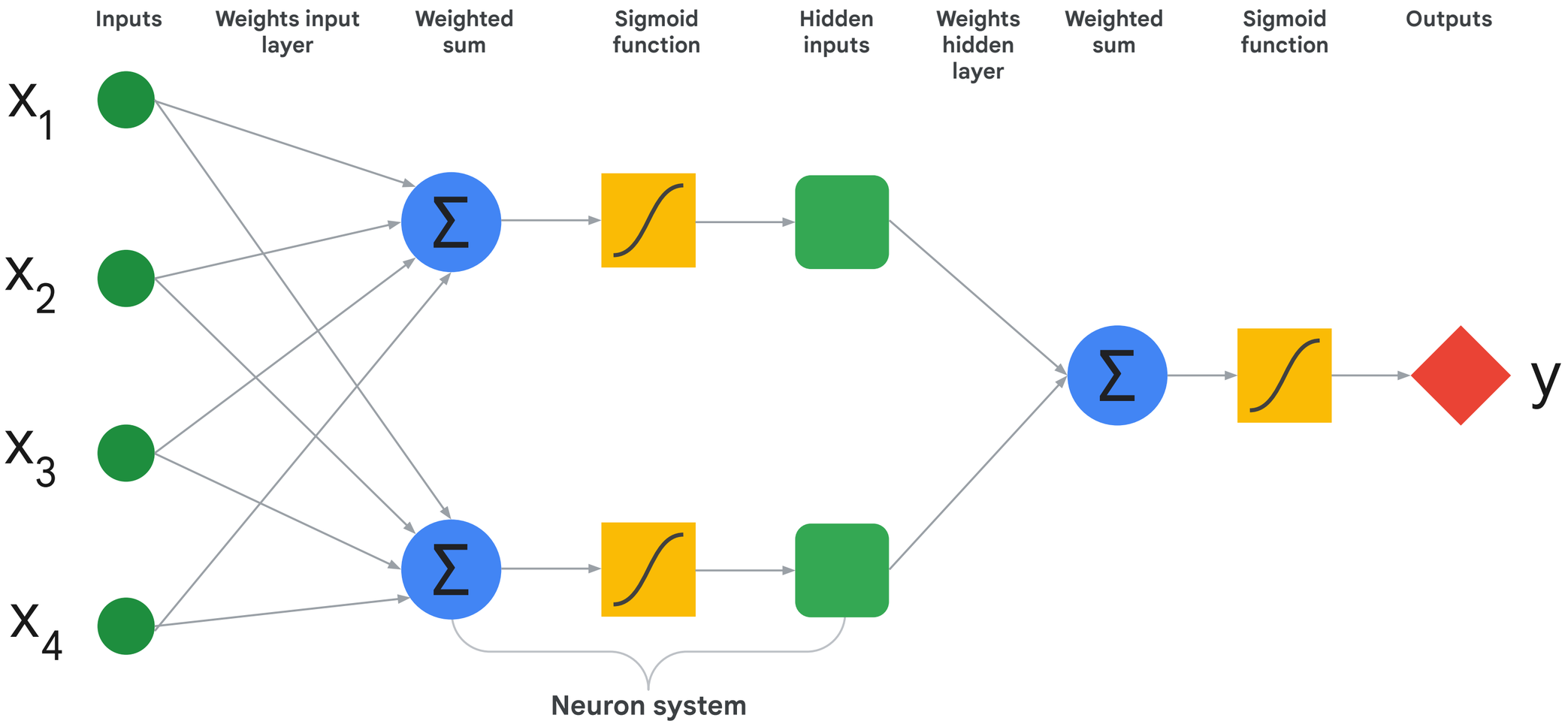
An ML recap


Staging/pre-production/production environments
Raw data
Data extraction
Data analysis
Data preparation
Model training
Model evaluation
Model validation
ML
Ops
Trained model
Model serving
Model Registry
Performance monitoring
Prediction service
Experimentation/development/test environments

An ML recap


Training phase
Labeled data
ML algorithm
Trained model
Inference phase
New data
Served model
predictions

An ML recap


Inputs
Weights input layer
Weighted sum
Sigmoid function
Hidden inputs
Weights hidden layer
Weighted sum
Sigmoid function
Outputs
x_1
x_2
x_3
x_4
Σ
Σ
Σ
y
Neuron system

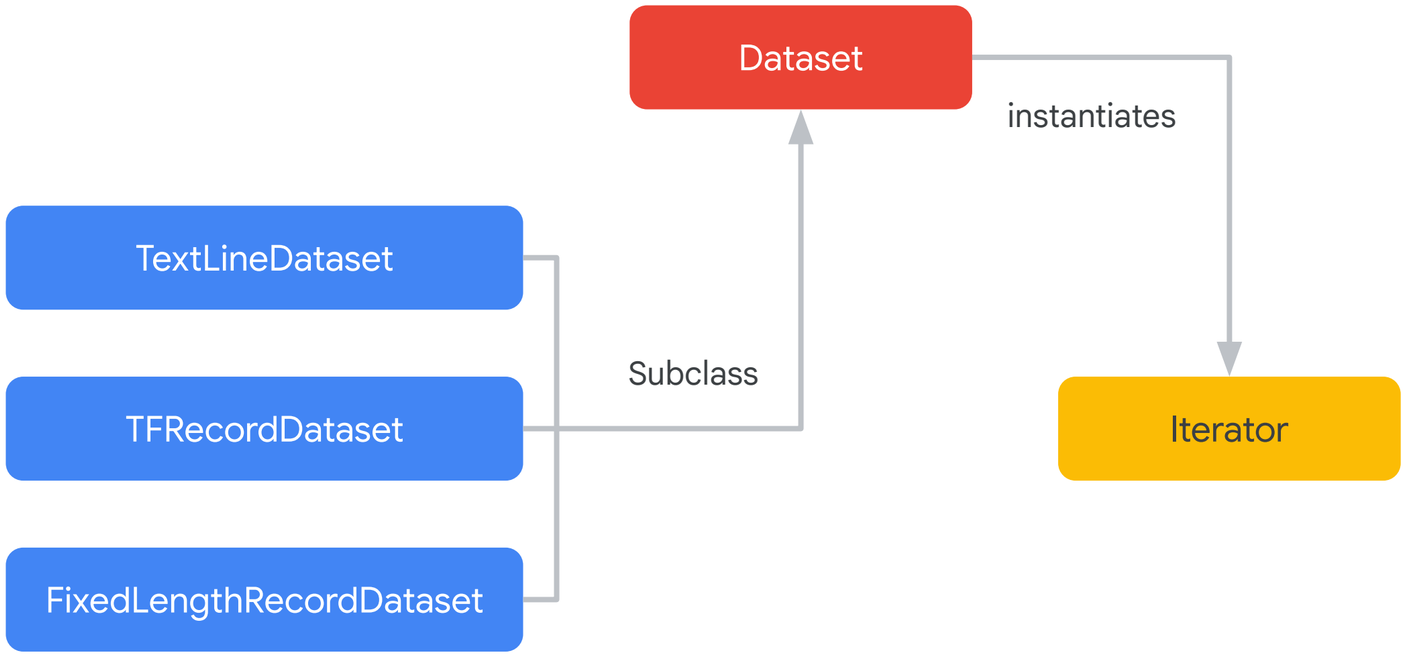
Performing a training step involves

1. Opening a file
(if it isn’t open already)
2. Fetching a data entry
from the file
3. Using the data for
training

tf.data API

1. Build complex input pipelines from simple, reusable pieces.
2. Build pipelines for multiple data types
3. Handle large amounts of data; perform complex transformations

Multiple ways to feed TensorFlow models with data



A tf.data.Dataset allows you to

- Create data pipelines from
  - in-memory dictionary and lists of tensors
  - out-of-memory sharded data files

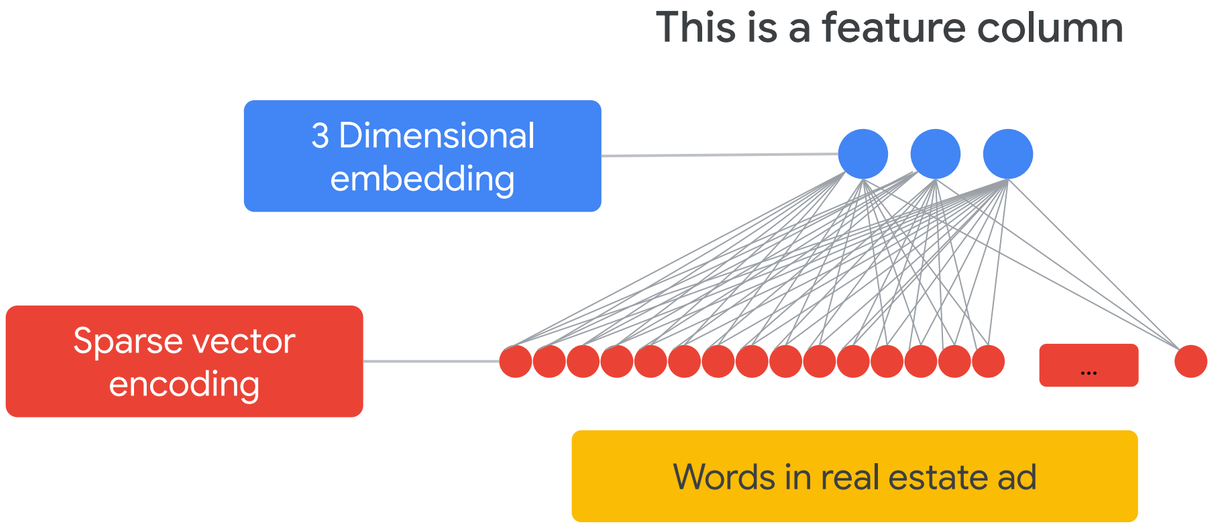
- Preprocess data in parallel (and cache result of costly operations)
  `dataset = dataset.map(preproc_fun).cache()`

- 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)`

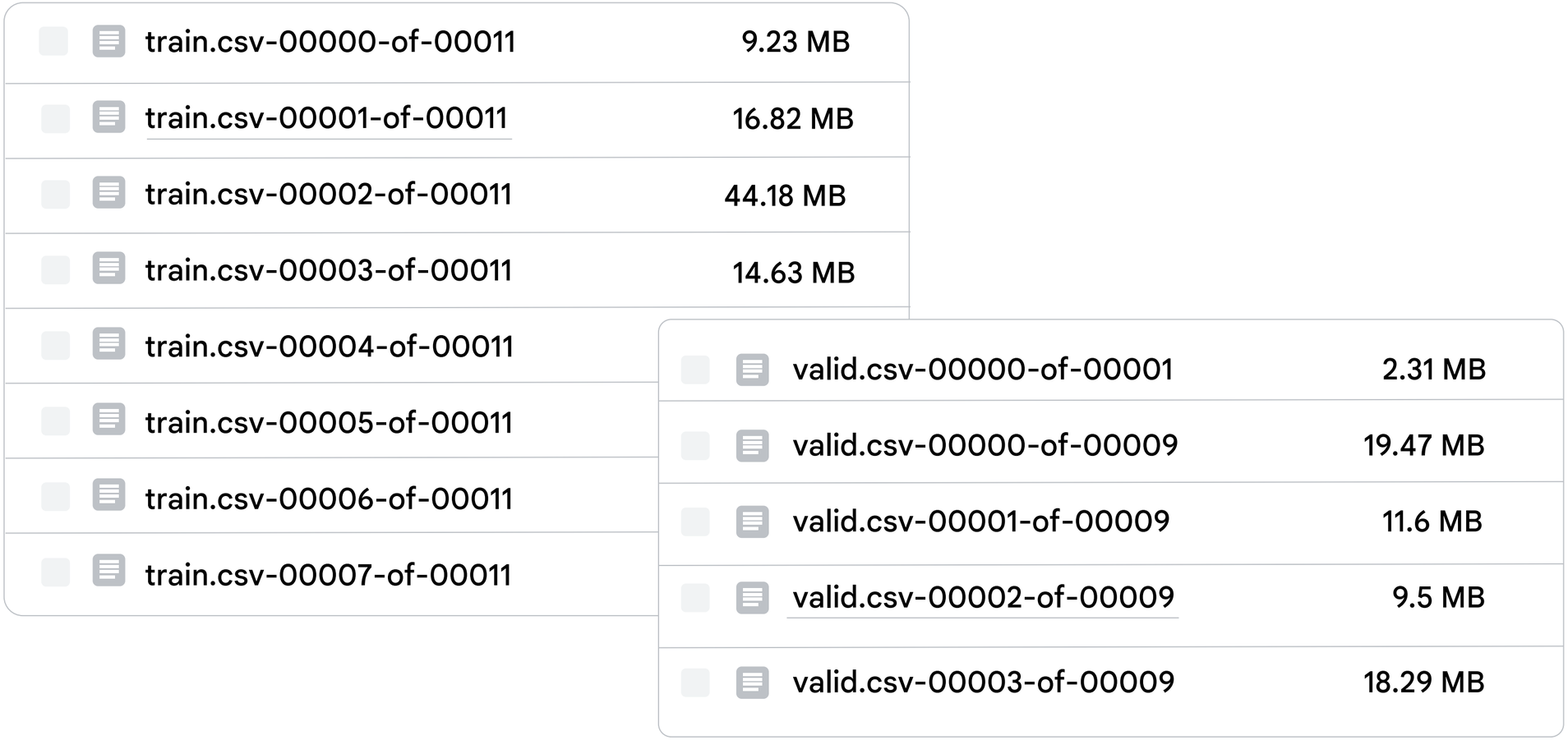
in a easy and very compact way

Embeddings are feature columns that function like layers

```
import tf.feature_column as fc

sparse_word =
fc.categorical_column_with_vocabulary_list
('word', vocabulary_list=englishWords)

embedded_word =
fc.embedding_column(sparse_word, 3)
```



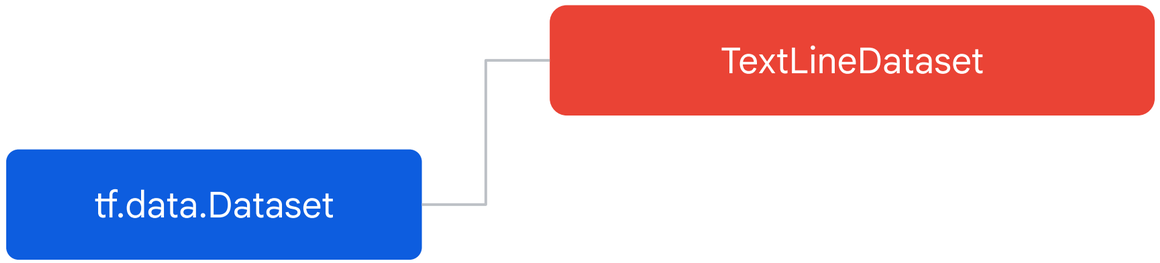
What about out-of-memory sharded datasets?



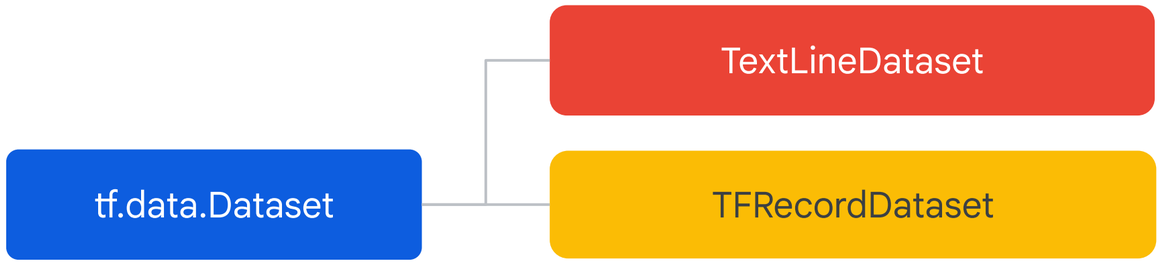
Datasets can be created from different file formats

tf.data.Dataset

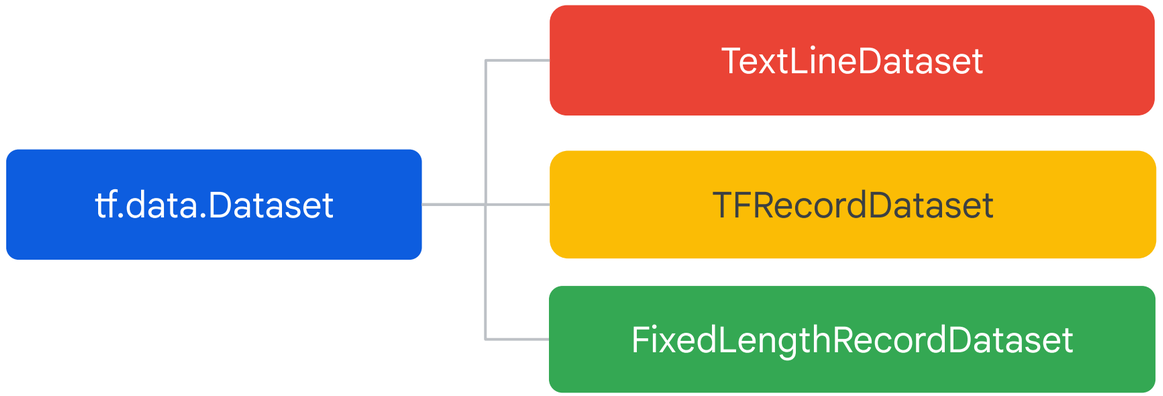
Datasets can be created from different file formats



Datasets can be created from different file formats

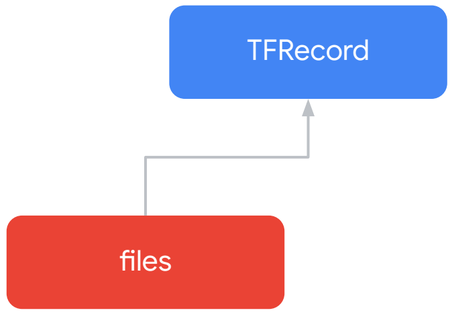


Datasets can be created from different file formats



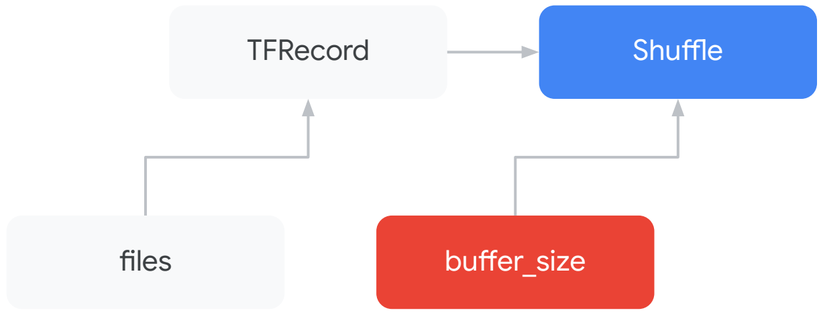
TFRecordDataset example

```
dataset = tf.data.TFRecordDataset(files)
```



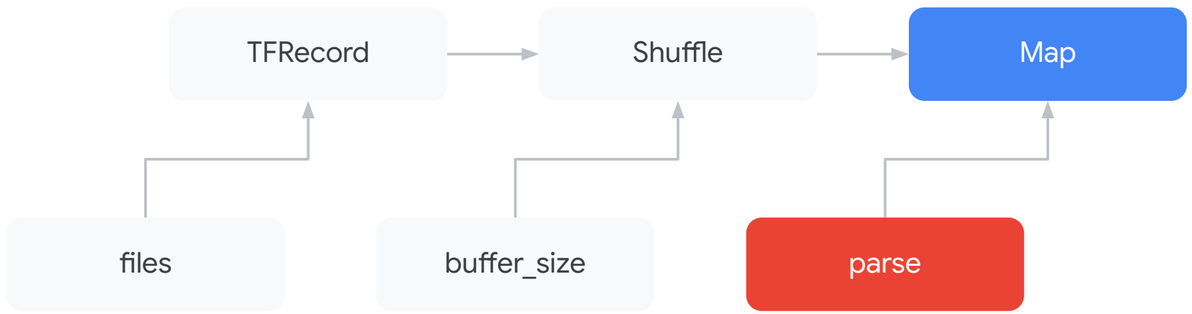
TFRecordDataset example

```
dataset = tf.data.TFRecordDataset(files)
dataset = dataset.shuffle(buffer_size=X)
```



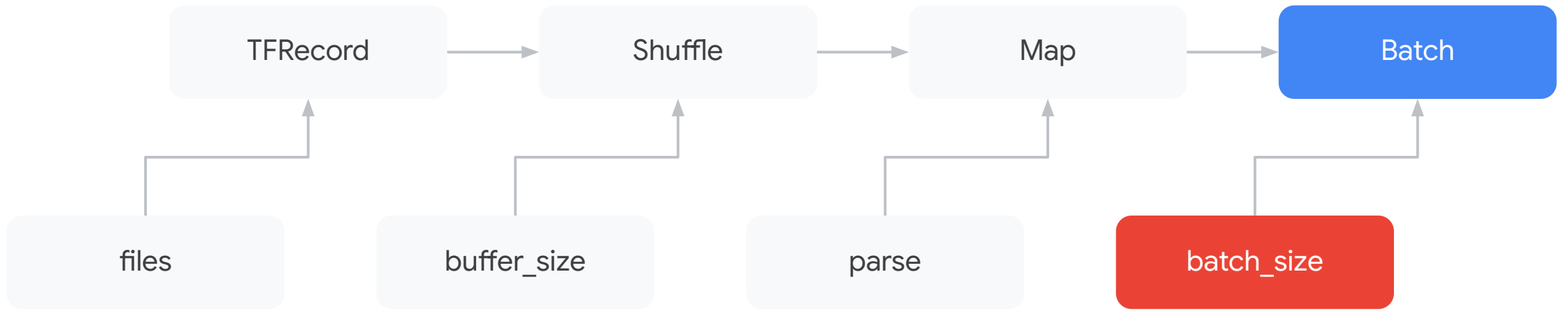
TFRecordDataset example

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



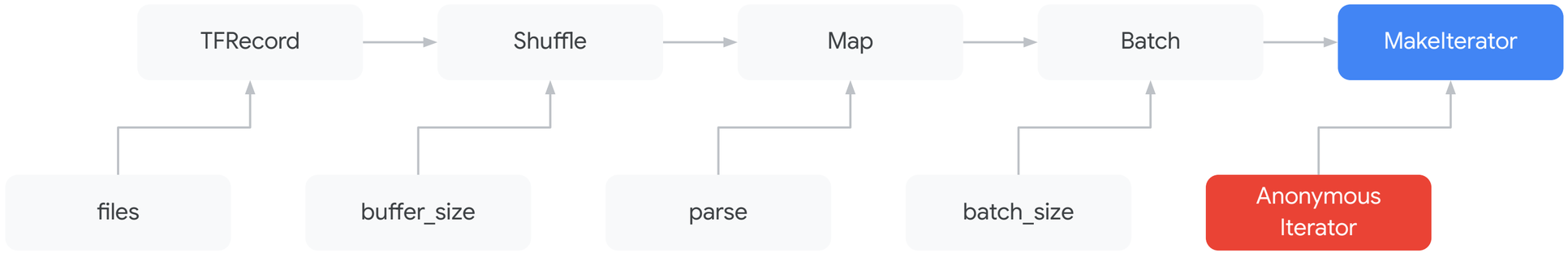
TFRecordDataset example

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



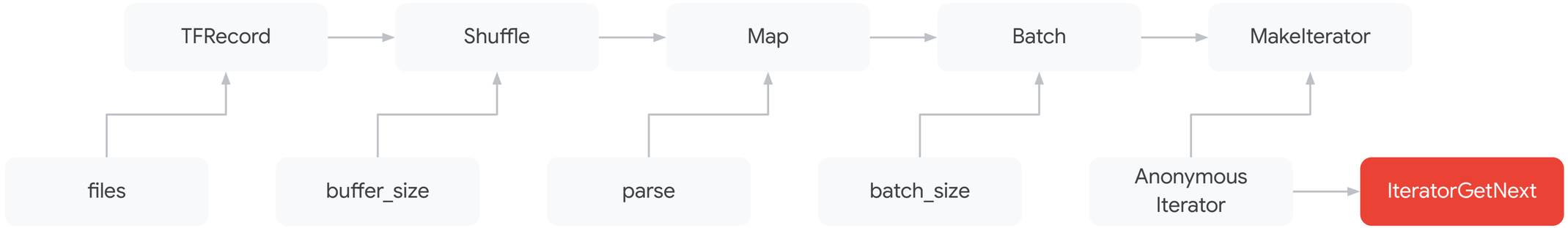
TFRecordDataset example

```
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
  ...
```



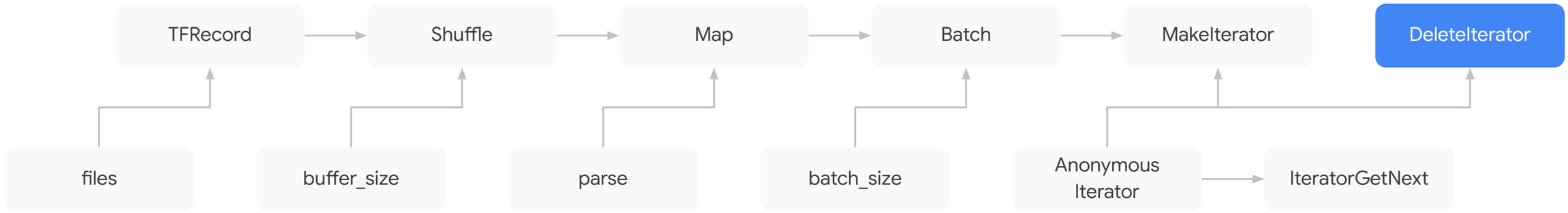
TFRecordDataset example

```
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
```



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
```



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

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]
```

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.]”

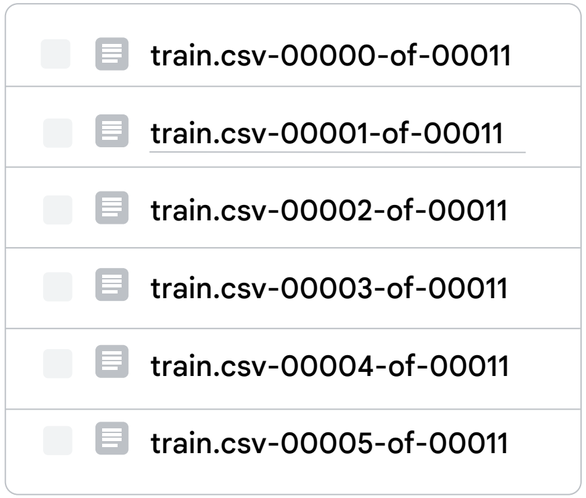
dataset = “[parse_row(line1), parse_row(line2), etc.]”

property type

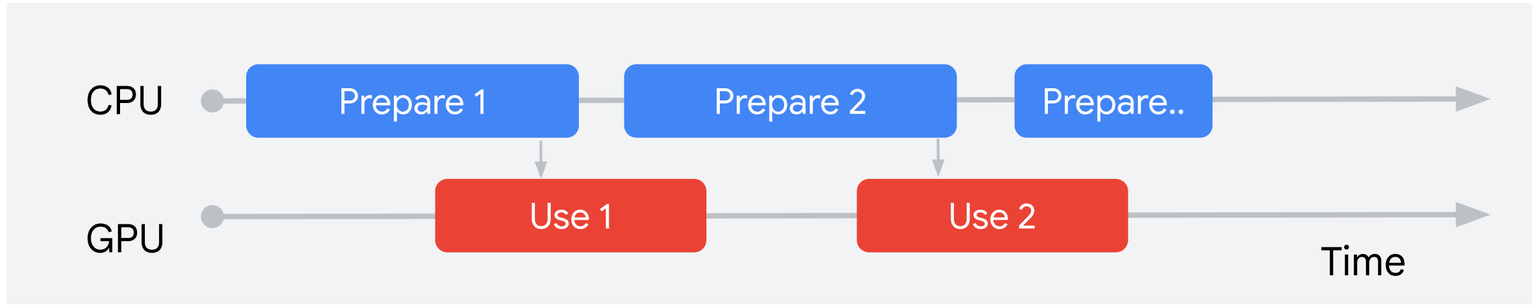
sq_footage		PRICE in K$
1001,	house,	501
2001,	house,	1001
3001,	house,	1501
1001,	apt,	701
2001,	apt,	1301
3001,	apt,	1901
1101,	house,	526
2101,	house,	1026

Read a set of sharded CSV files using TextLineDataset

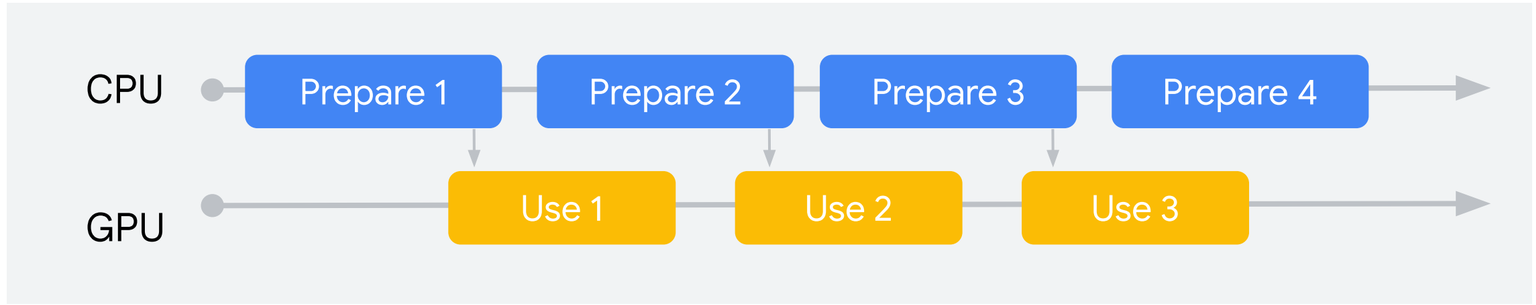
```
def parse_row(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label


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
```



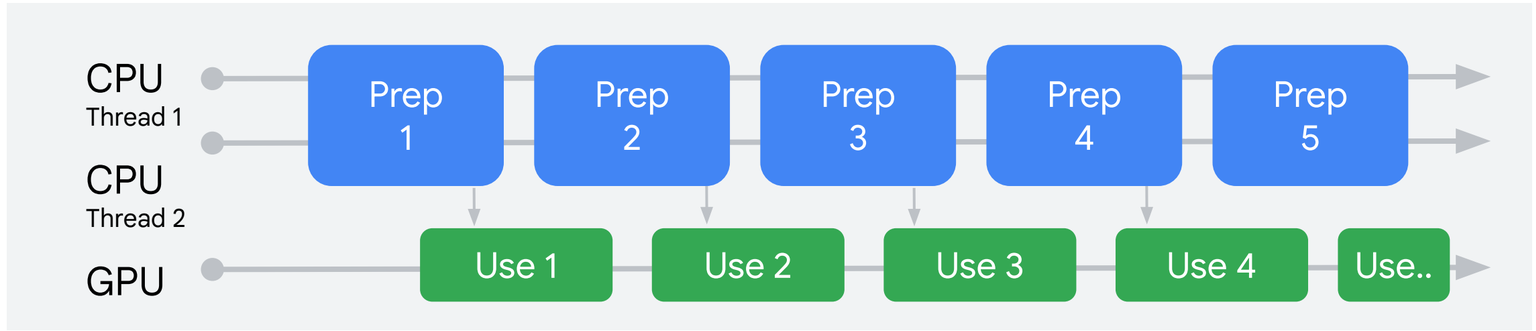
Without prefetching



With prefetching



With prefetching + multithreaded loading & preprocessing

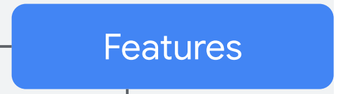


The real benefit of Dataset is that you can do more than just ingest data

```
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()
```

Feature columns tell the model what inputs to expect

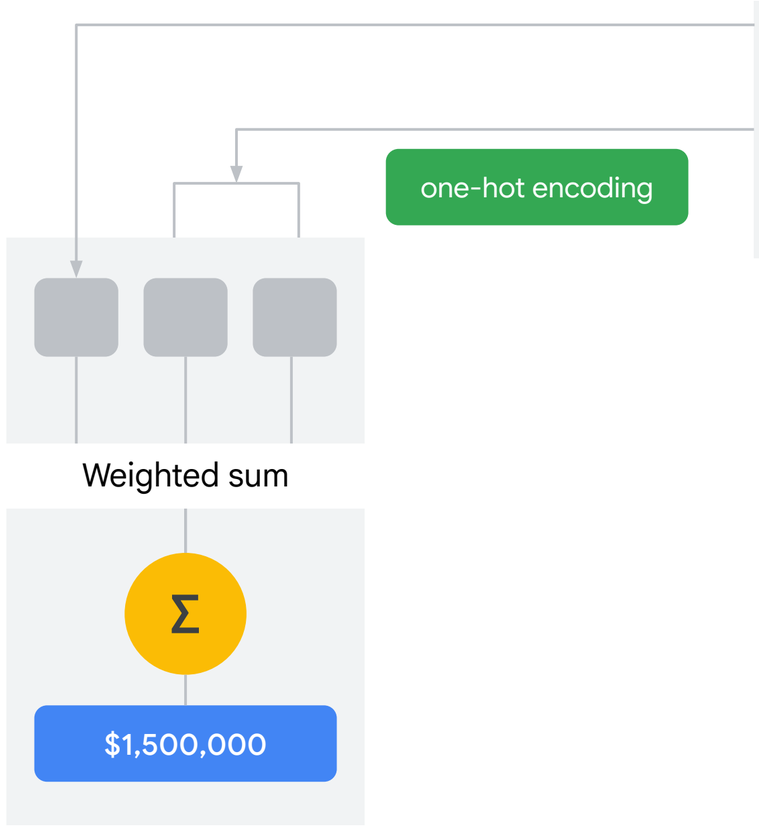
```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type",
                                                            ["house", "apt"])
]

...
```



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

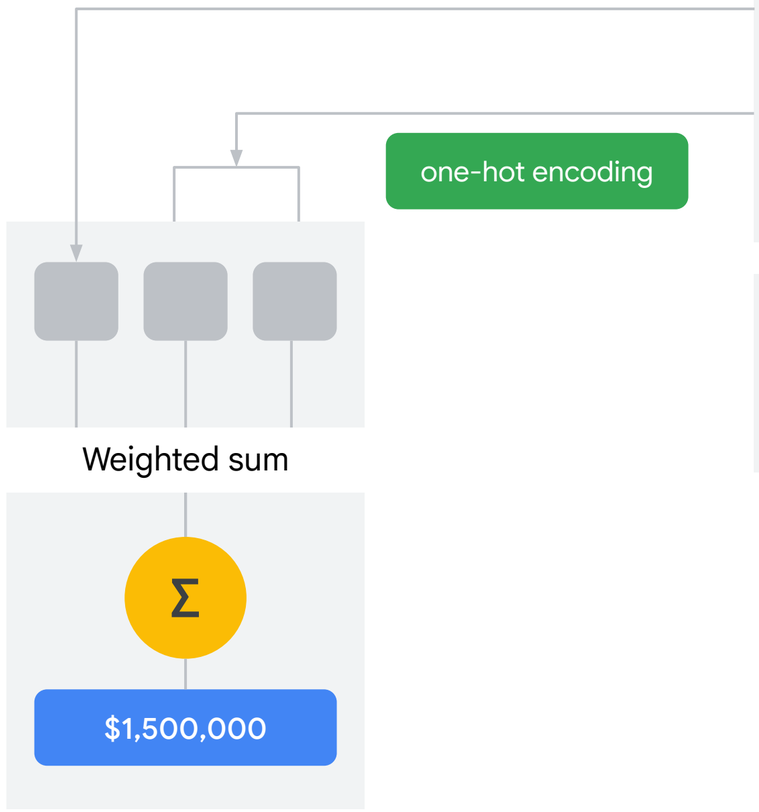
```
tf.feature_column.numeric_column("sq_footage")

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

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"])
```

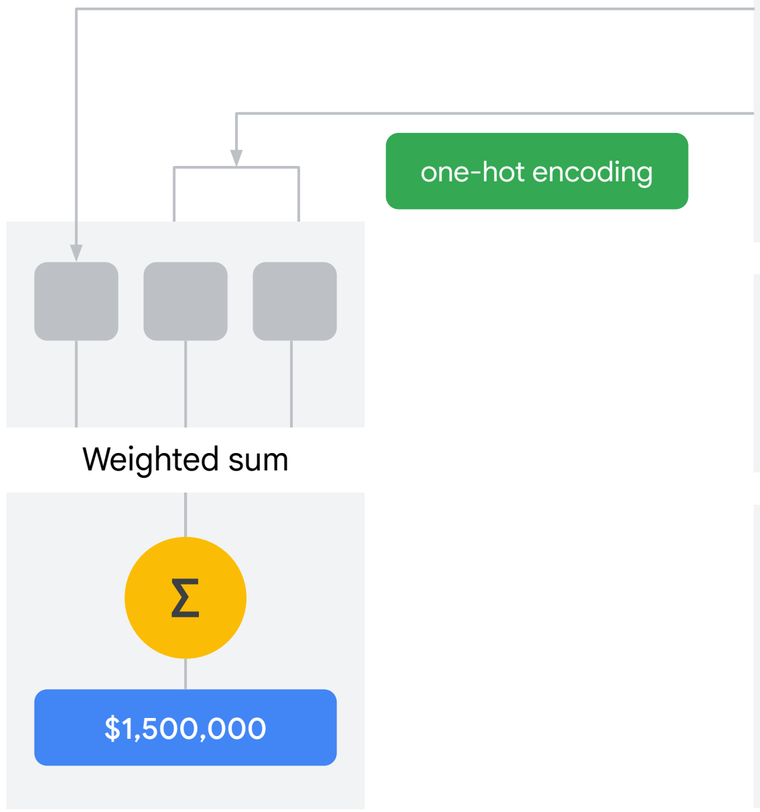
Under the hood: Feature columns take care of packing the inputs into the input vector of the model




```
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...
```

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(..)
...
```

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()
```

set up numeric ranges

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

create bucketized columns for pickup latitude and pickup longitude

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

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)
```

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

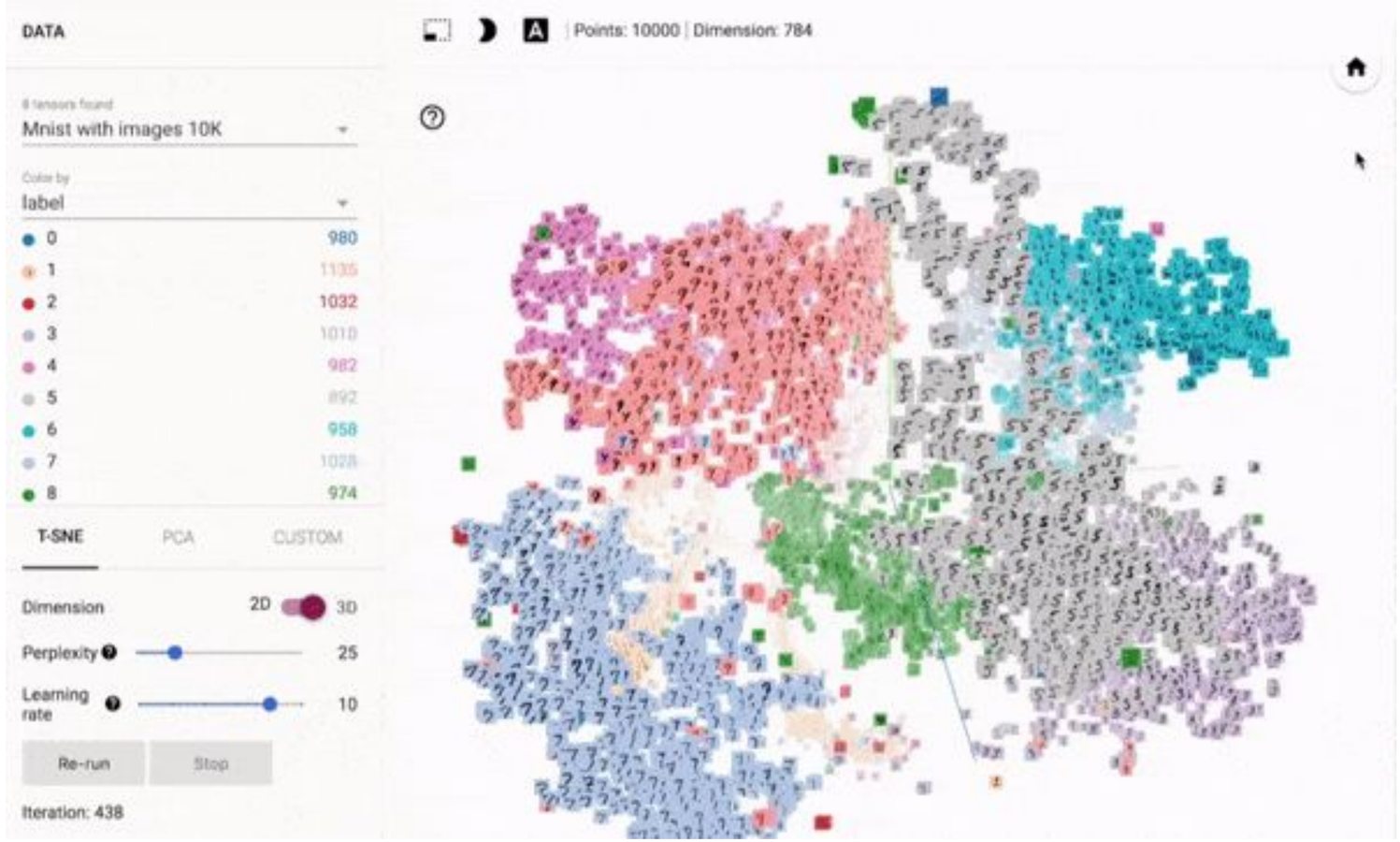
```
fc_ploc = fc.embedding_column(categorical_column=fc_crossed_ploc,
                              dimension=3)
```

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

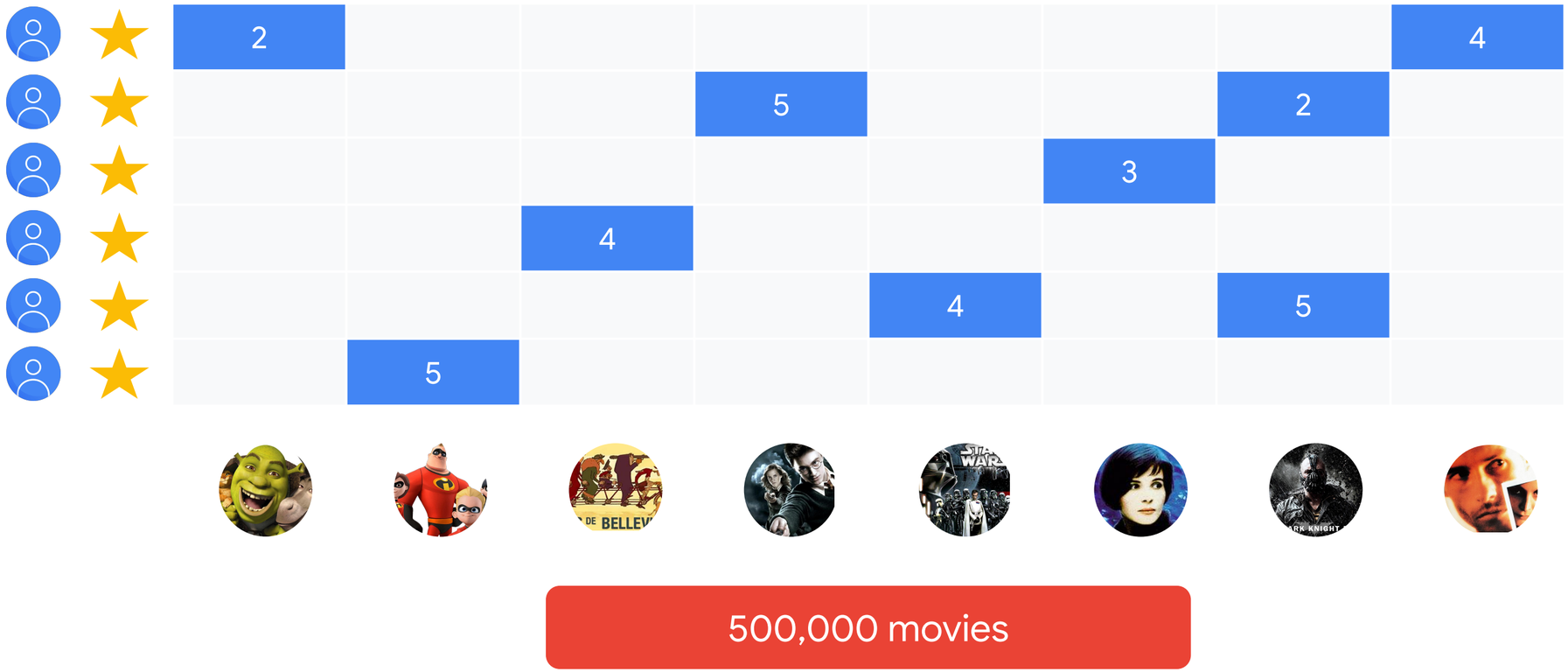
How can we visually cluster 10,000 variations of handwritten digits to look for similarities?

Embeddings!

Embeddings are everywhere in modern machine learning



How do you recommend movies to customers?



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

Average age of viewers


Shrek


Incredibles

The Triplets of Belleville


Harry Potter

Star Wars


Bleu


The Dark Knight Rises


Memento

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

Shrek

Incredibles

The Triplets of Belleville

Harry Potter

Star Wars

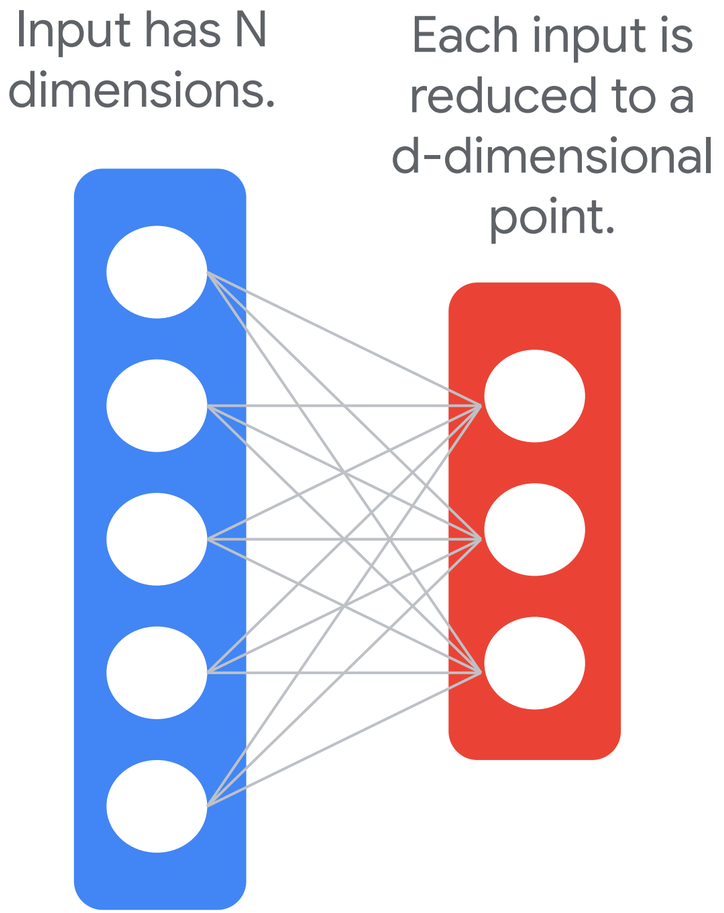
The Dark Knight Rises

Bleu

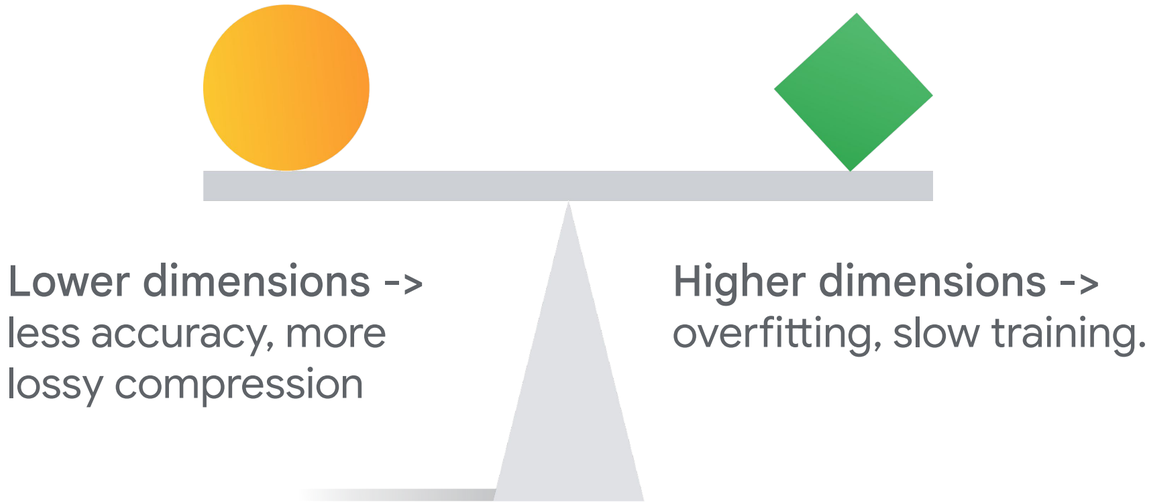
Memento

Gross ticket sales

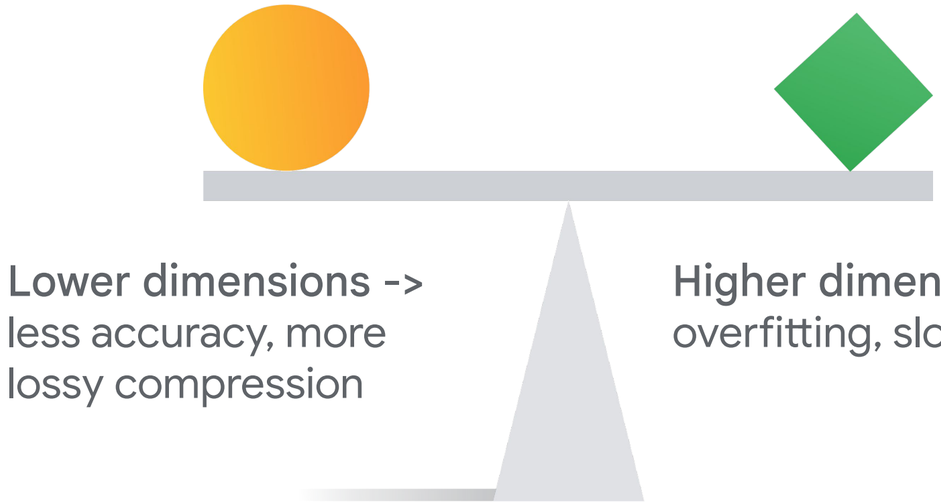
A d-dimensional embedding assumes that user interest in movies can be approximated by d aspects



A good starting point for number of embedding dimensions



A good starting point for number of embedding dimensions



Lower dimensions -> less accuracy, more lossy compression

Higher dimensions -> overfitting, slow training.

$$\text{Dimensions} \approx \sqrt[4]{\text{Possible values}}$$

Empirical tradeoff.

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

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
```

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
```

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')
])

...
```

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()
```

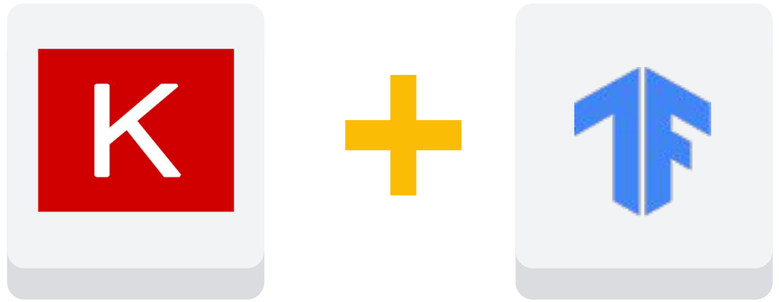
You will learn and practice this later after first mastering dataset manipulation!

Data is the a crucial component
of a machine learning model

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.



Keras preprocessing layers

1. Text preprocessing
2. Numerical features preprocessing
3. Categorical features preprocessing
4. Image preprocessing
5. Image data augmentation

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.

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.

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.

Categorical features preprocessing

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 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.

Categorical features preprocessing

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 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.

Categorical features preprocessing

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 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.

Categorical features preprocessing

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 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.

The adapt() method

Stateful preprocessing layers that compute based on training data/:

01

TextVectorization:

Holds a mapping between string tokens and integer indices.

02

StringLookup and IntegerLookup:

Hold a mapping between input values and integer indices.

03

Normalization:

Holds the mean and standard deviation of the features.

04

Discretization:

Holds information about value bucket boundaries.

Important note: These layers are non-trainable. Their state is not set during training; it must be set before training.

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

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

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
```

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))
```

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))
```

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))
```

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
```

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

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)
```

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)
```

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)
```

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))
```

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))
```

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)
```

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:
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).

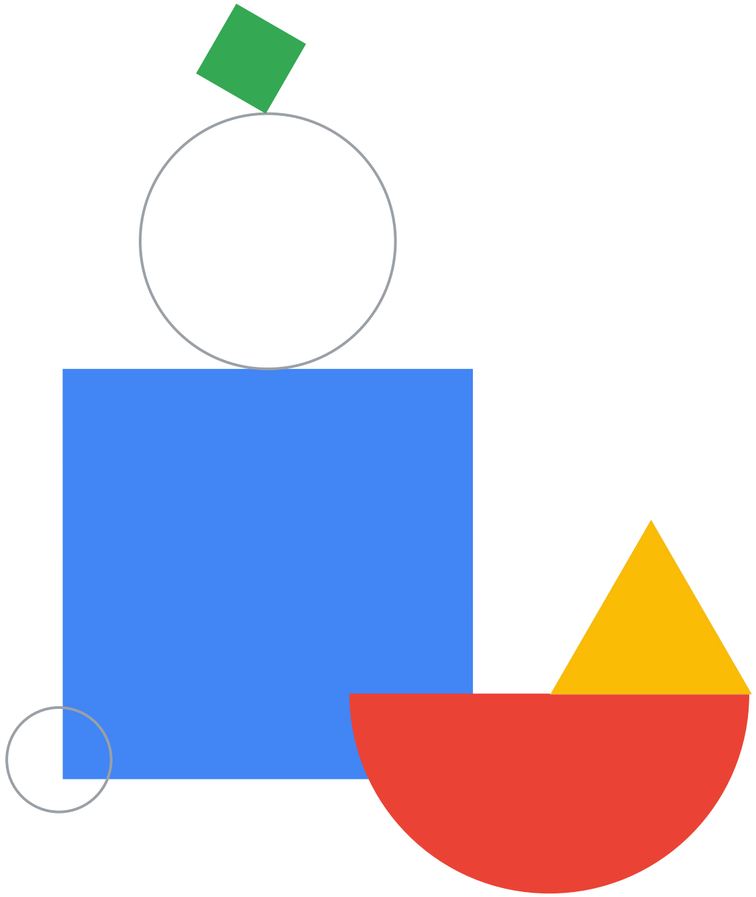
Benefits of doing preprocessing inside the model at inference time

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.

Google Cloud



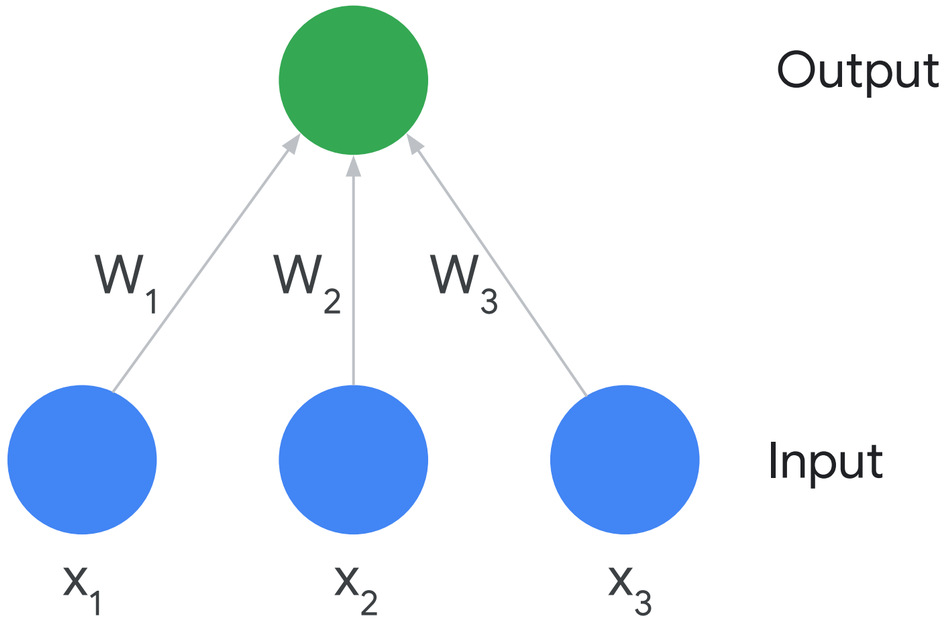
Building Neural Networks with the TensorFlow and Keras API

In this module, you learn to ...

01	Describe activation functions, loss, and optimizers
02	Build a DNN model using the Keras Sequential and Functional APIs
03	Use Keras preprocessing layers
04	Save/load and deploy a Keras model
05	Describe model subclassing

A Linear Model can be represented as nodes and edges

$$\text{Output} = W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$

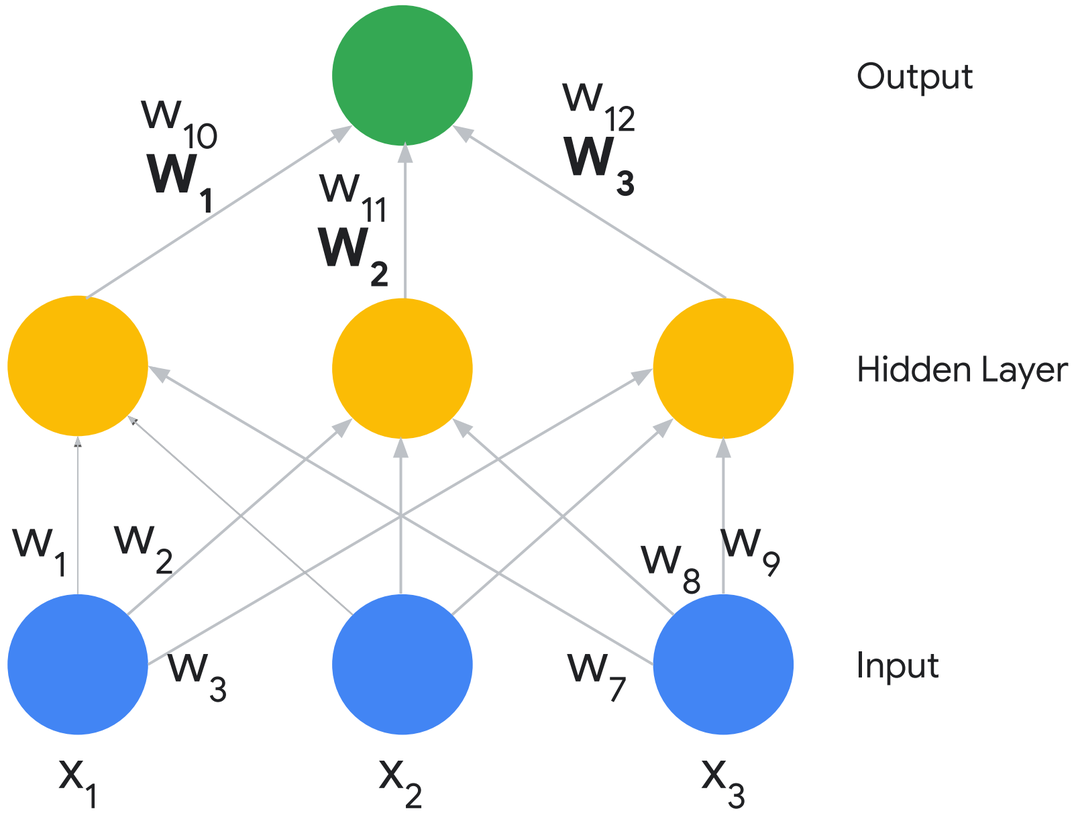


Add Complexity: Non-Linear?

$$\text{Output} = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

$$((w_{10} * w_1) + (w_{10} * w_4) + (w_{10} * w_7)) * x1$$
$$+ ((w_{11} * w_1) + (w_{11} * w_4) + (w_{11} * w_7)) * x2$$
$$+ ((w_{12} * w_1) + (w_{12} * w_4) + (w_{12} * w_7)) * x3$$

$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



Add Complexity: Non-Linear?

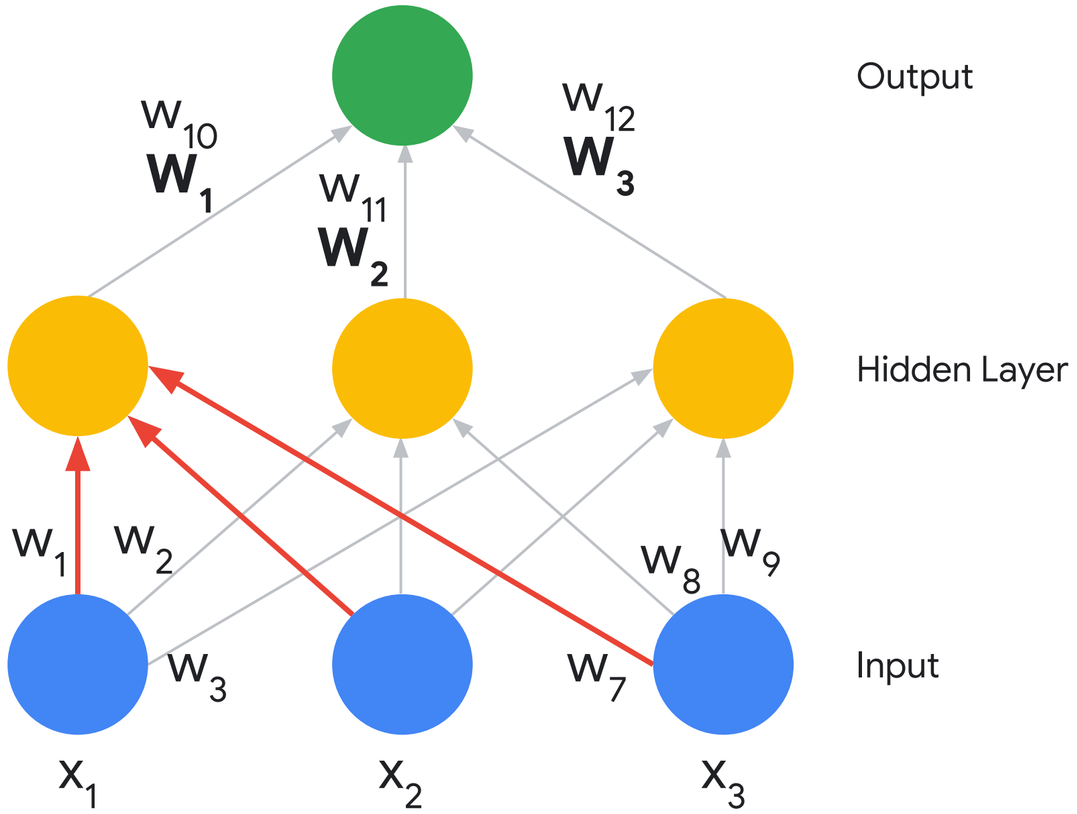
$$\text{Output} = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

$$((w_{10} * w_1) + (w_{10} * w_4) + (w_{10} * w_7)) * x1$$

$$+ ((w_{11} * w_1) + (w_{11} * w_4) + (w_{11} * w_7)) * x2$$

$$+ ((w_{12} * w_1) + (w_{12} * w_4) + (w_{12} * w_7)) * x3$$

$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



Add Complexity: Non-Linear?

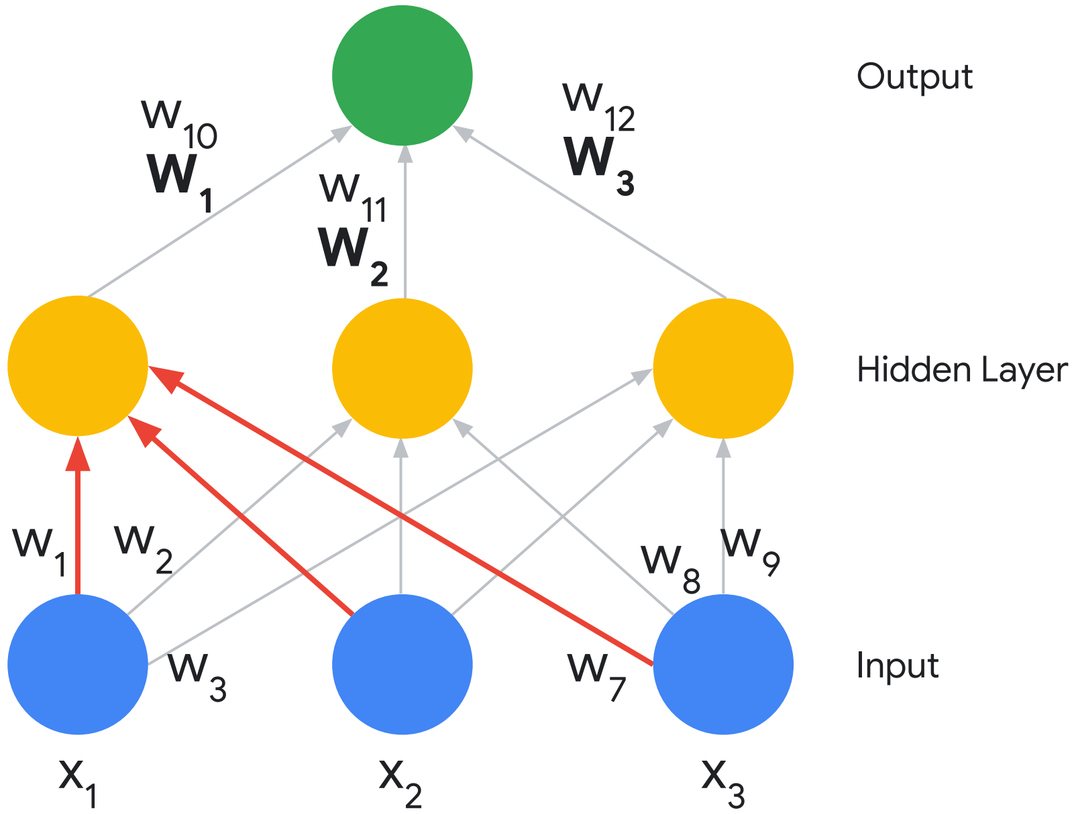
$$\text{Output} = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

$$((w_{10} * w_1) + (w_{10} * w_4) + (w_{10} * w_7)) * x1$$

$$+ ((w_{11} * w_1) + (w_{11} * w_4) + (w_{11} * w_7)) * x2$$

$$+ ((w_{12} * w_1) + (w_{12} * w_4) + (w_{12} * w_7)) * x3$$

$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



Add Complexity: Non-Linear?

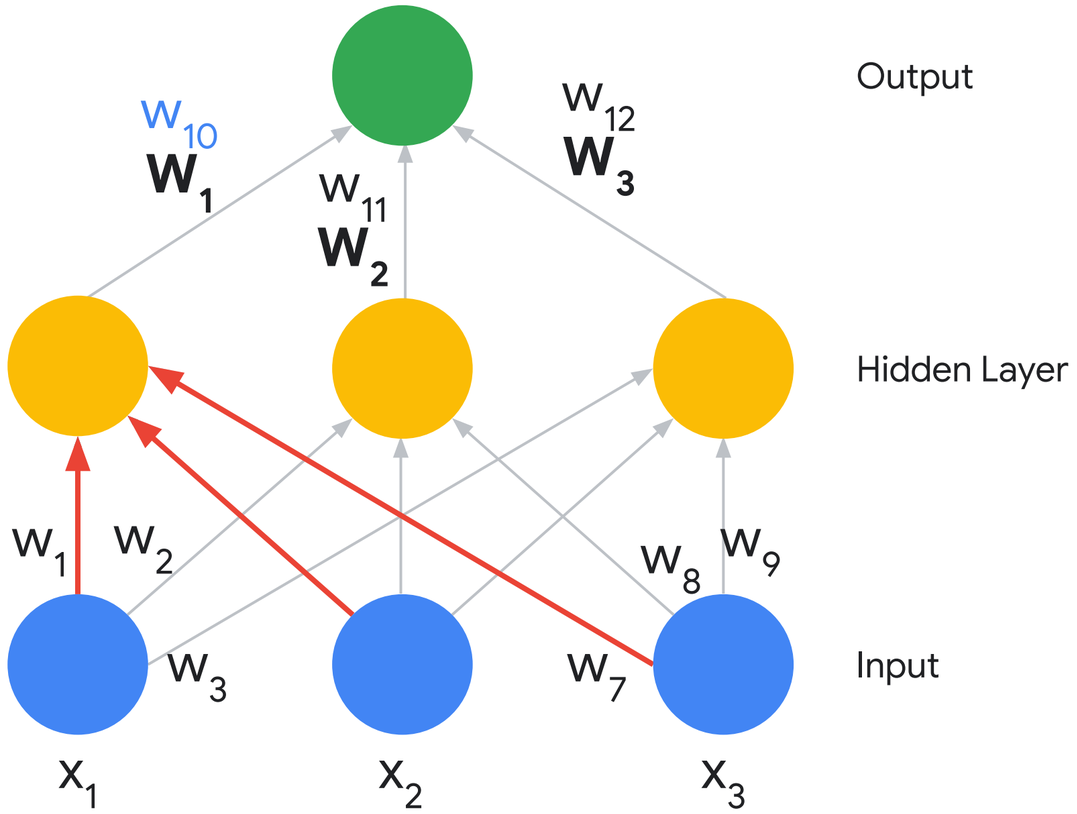
$$\text{Output} = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

$$((w_{10} * w_1) + (w_{10} * w_4) + (w_{10} * w_7)) * x1$$

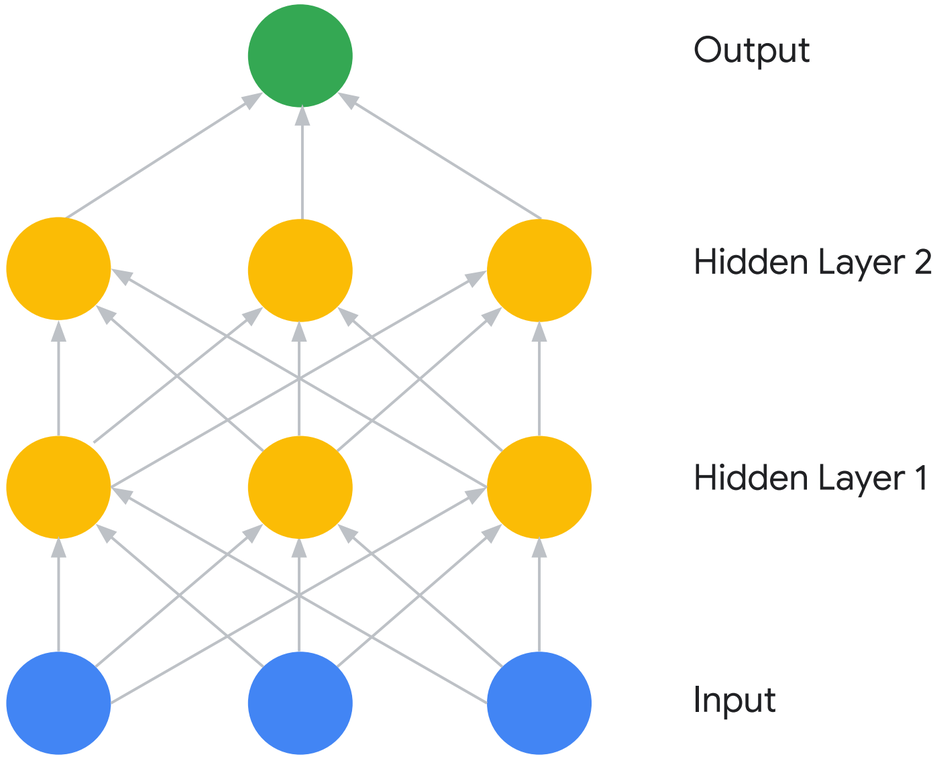
$$+ ((w_{11} * w_1) + (w_{11} * w_4) + (w_{11} * w_7)) * x2$$

$$+ ((w_{12} * w_1) + (w_{12} * w_4) + (w_{12} * w_7)) * x3$$

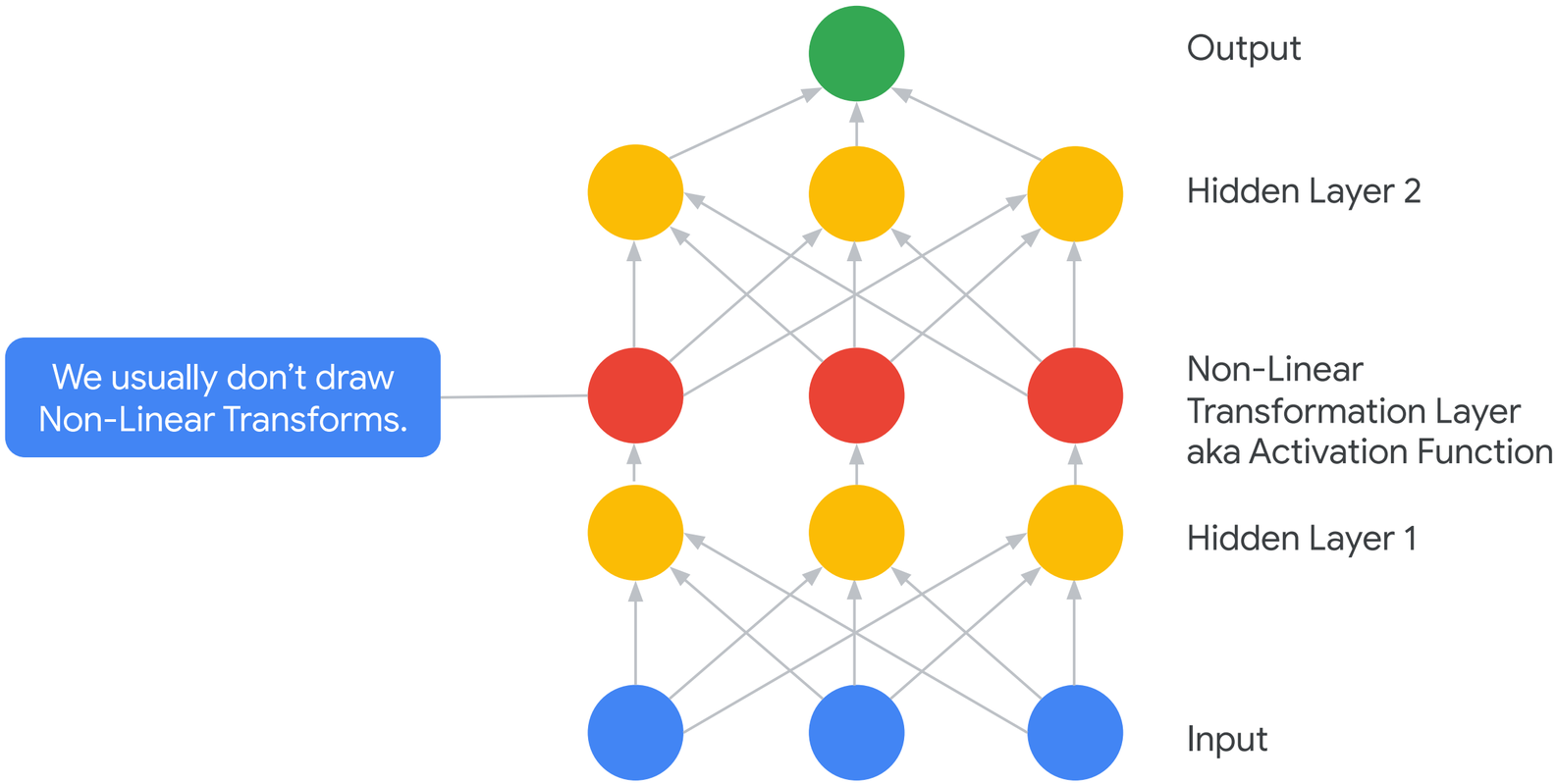
$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



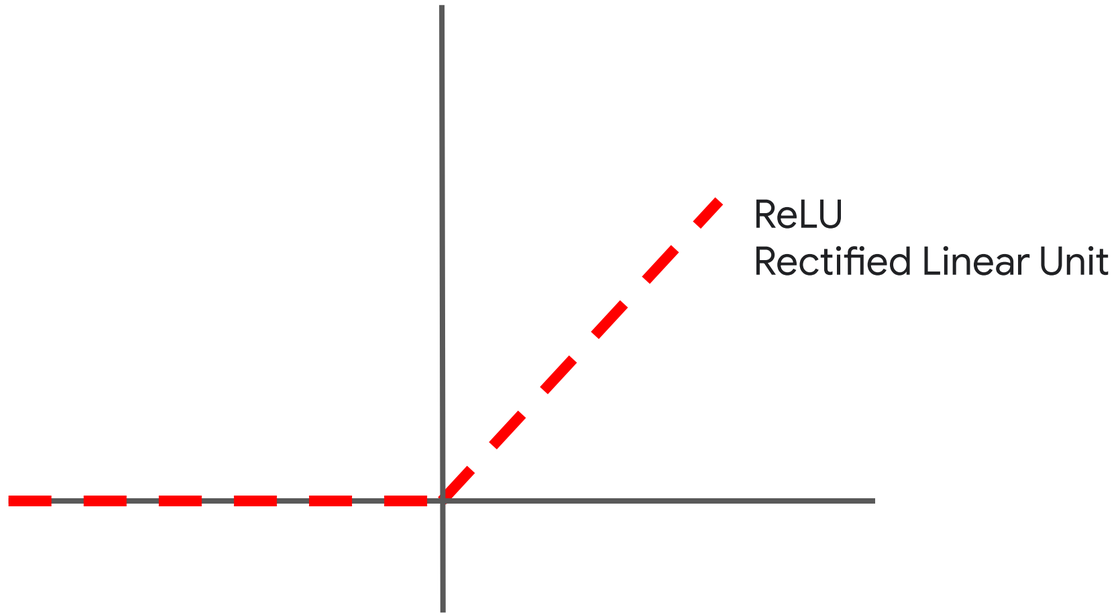
Add Complexity: Non-Linear?



Adding a Non-Linearity



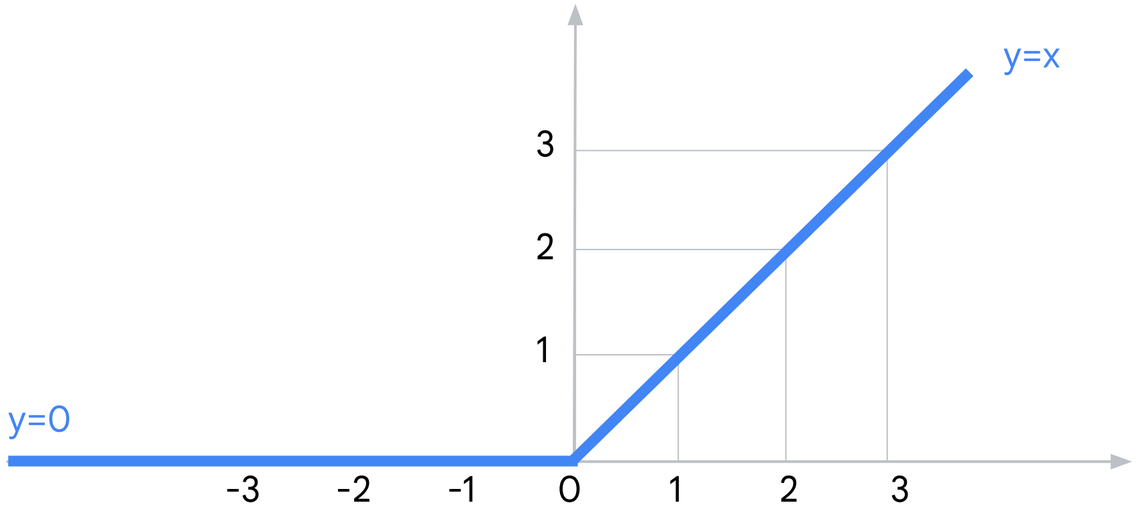
Our favorite non-linearity is the Rectified Linear Unit



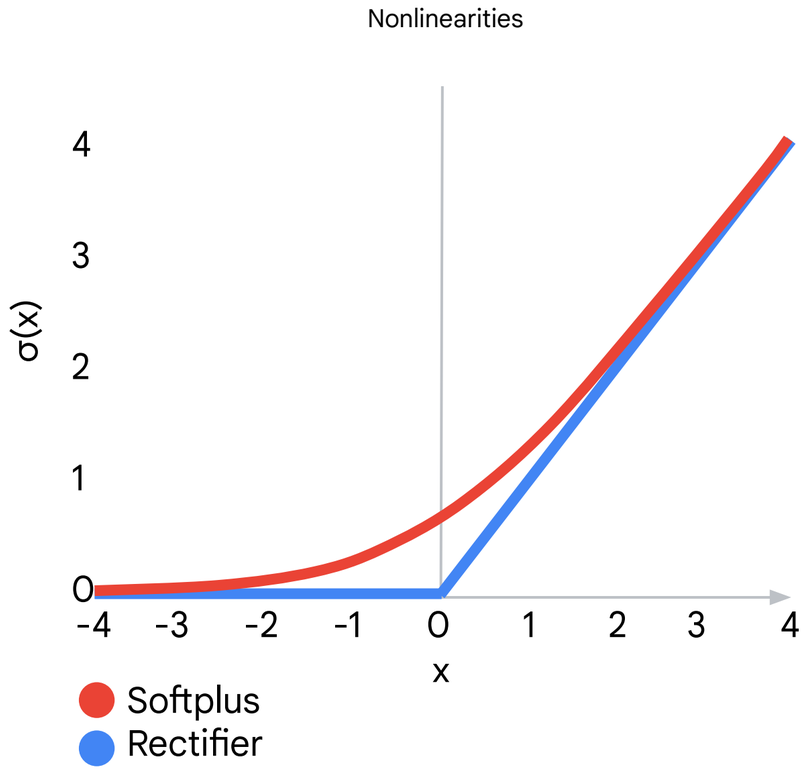
$f(x) = \max(0, x)$

There are many different ReLU variants

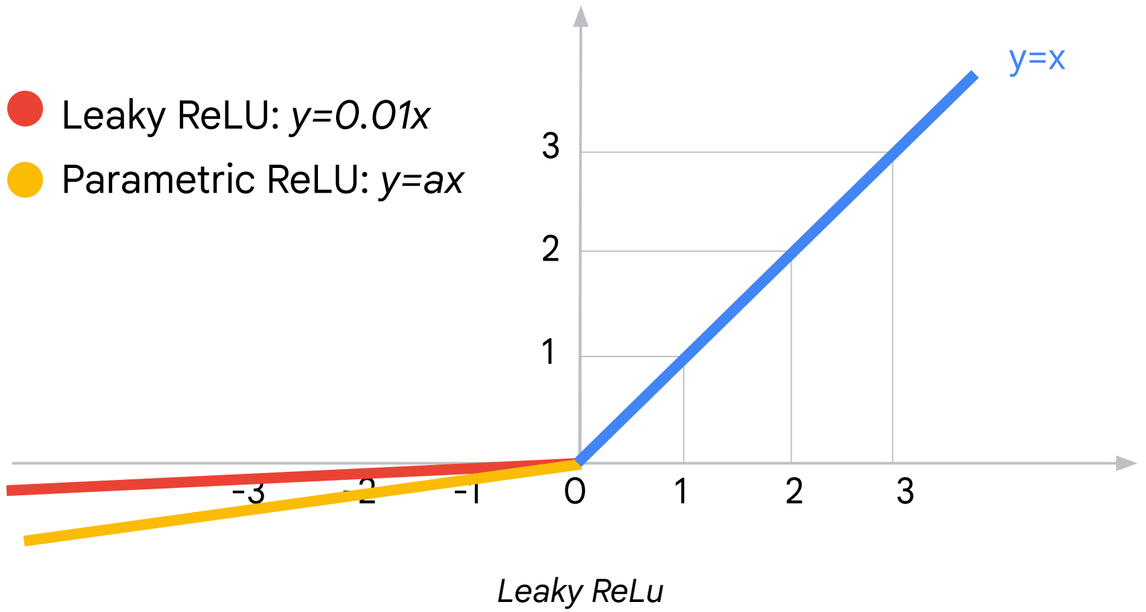
Normal ReLU activation function



There are many different ReLU variants

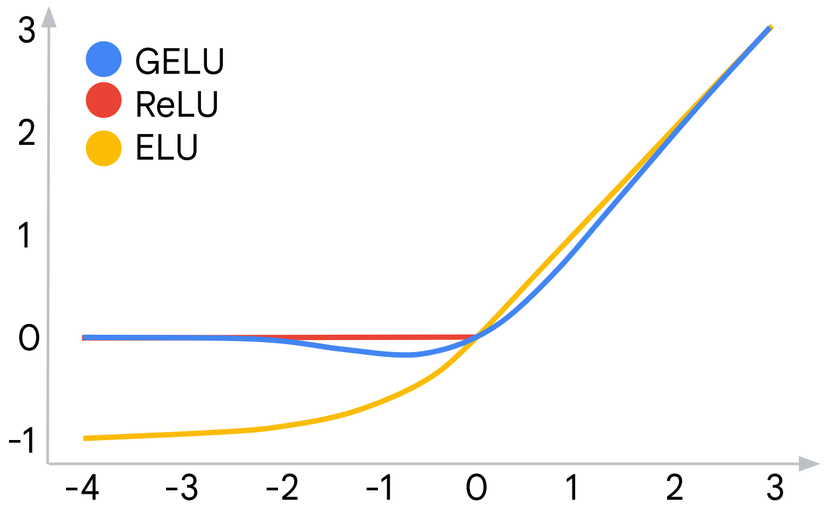


There are many different ReLU variants

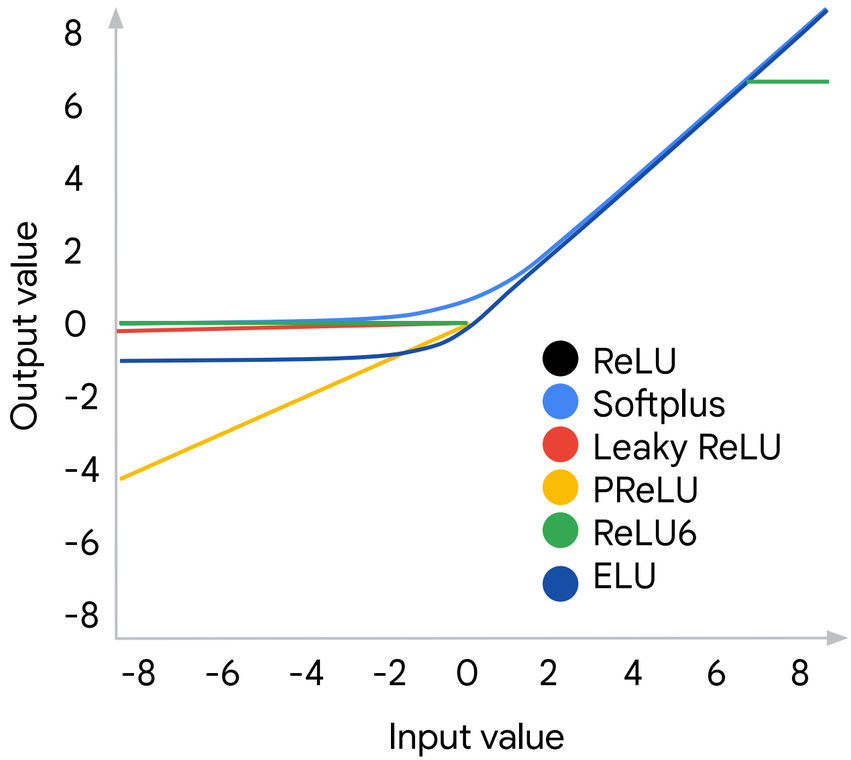


Leaky ReLu

There are many different ReLU variants



There are many different ReLU variants



Keras is built-in to TF 2.x

Keras is built-in to TF 2.x

- User-friendly
- Modular and composable
- Easy to extend

Stacking layers with Keras Sequential model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


model = Sequential([
    Input(shape=(64,))
    Dense(units=32, activation="relu", name="hidden1"),
    Dense(units=8, activation="relu", name="hidden2"),
    Dense(units=1, activation="linear", name="output")
])
```

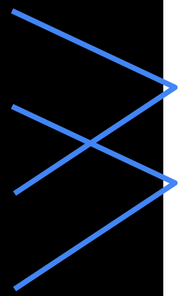
The Keras sequential model stacks layers on the top of each other.

The batch size is omitted. Here the model expects batches of vectors with 64 components.

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

A linear model (multiclass logistic regression)

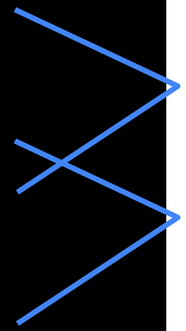
```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(10, activation='softmax')
])
```



A linear model (multiclass logistic regression)

A linear model (a single Dense layer) aka multiclass logistic regression

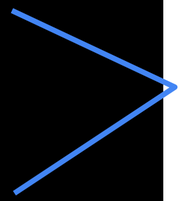
```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```



A neural network with one hidden layer

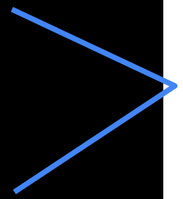
A neural network with one hidden layer

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```



A neural network with multiple hidden layers (a deep neural network)

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()

# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```



A deeper neural network

Compiling a Keras model

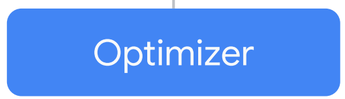
```
def rmse(y_true, y_pred):
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))


model.compile(optimizer="adam", loss="mse", metrics=[rmse, "mse"])
```

Custom Metric

Loss function

Compiling a Keras model

```
def rmse(y_true, y_pred):
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))


model.compile(optimizer="adam", loss="mse", metrics=[rmse, "mse"])
```



Compiling a Keras model

```
def rmse(y_true, y_pred):
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))


model.compile(optimizer="adam", loss="mse", metrics=[rmse, "mse"])
```

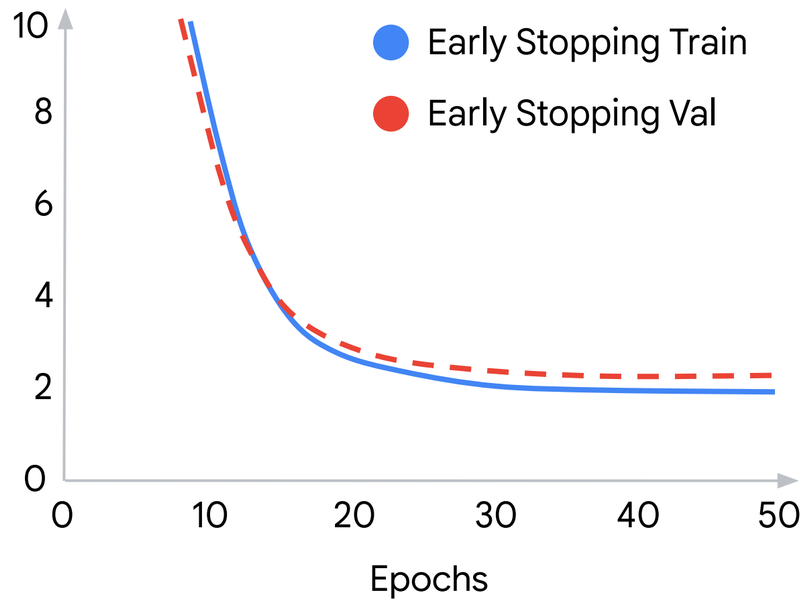
Training a Keras model

```
from tensorflow.keras.callbacks import TensorBoard


steps_per_epoch = NUM_TRAIN_EXAMPLES // (TRAIN_BATCH_SIZE * NUM_EVALS)


history = model.fit(
   x=trainds,
   steps_per_epoch=steps_per_epoch,
   epochs=NUM_EVALS,
   validation_data=evalds,
   callbacks=[TensorBoard(LOGDIR)]
)
```

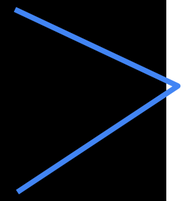
This is a trick so that we have control on the total number of examples the model trains on (NUM_TRAIN_EXAMPLES) and the total number of evaluation we want to have during training (NUM_EVALS).

Training a Keras model

```
from tensorflow.keras.callbacks import TensorBoard


steps_per_epoch = NUM_TRAIN_EXAMPLES //
(TRAIN_BATCH_SIZE * NUM_EVALS)


history = model.fit(
   x=trainds,
   steps_per_epoch=steps_per_epoch,
   epochs=NUM_EVALS,
   validation_data=evalds,
   callbacks=[TensorBoard(LOGDIR)]
)
```



```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data()
# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
# Configure and train
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```



A deeper neural network

Once trained, the model can be used for prediction

```
predictions = model.predict(input_samples, steps=1)
```

returns a Numpy array of predictions

With the predict() method you can pass

- a Dataset instance
- Numpy array
- a TensorFlow tensor, or list of tensors
- a generator of input samples

steps determines the total number of steps before declaring the prediction round finished. Here, since we have just one example, steps=1.

To serve our model for others to use, we export the model file and deploy the model as a service.

SavedModel is the universal serialization format for TensorFlow models

```
OUTPUT_DIR = "./export/savedmodel"
shutil.rmtree(OUTPUT_DIR, ignore_errors=True)

EXPORT_PATH = os.path.join(OUTPUT_DIR,
datetime.datetime.now().strftime("%Y%m%d%H%M%S"))

tf.saved_model.save(model, EXPORT_PATH)
```

exports a model object to a SavedModel format

a trackable object such as a trained keras model

the directory in which to write the SavedModel

Create a model object in Vertex AI

```
MODEL_NAME=propertyprice
VERSION_NAME=dnn

if [[ $(gcloud ai-platform models list --format='value(name)' | grep $MODEL_NAME) ]];
then
    echo "$MODEL_NAME already exists"
else
    echo "Creating $MODEL_NAME"
    gcloud ai-platform models create --regions=$REGION $MODEL_NAME
fi
...
```

create the model in AI Platform

Create a version of the model in Vertex AI

```
MODEL_NAME=propertyprice
VERSION_NAME=dnn

...

if [[ $(gcloud ai-platform versions list --model $MODEL_NAME --format='value(name)' |
grep $VERSION_NAME) ]]; then
    echo "Deleting already existing $MODEL_NAME:$VERSION_NAME ... "
    echo yes | gcloud ai-platform versions delete --model=$MODEL_NAME $VERSION_NAME
    echo "Please run this cell again if you don't see a Creating message ... "
    sleep 2
fi
```

create the model version in AI Platform

Deploy SavedModel using gcloud ai-platform

```
gcloud ai-platform versions create \
       --model=$MODEL_NAME $VERSION_NAME \
       --framework=tensorflow \
       --python-version=3.5 \
       --runtime-version=2.1 \
       --origin=$EXPORT_PATH \
       --staging-bucket=gs://$BUCKET
```

specify the model name and version

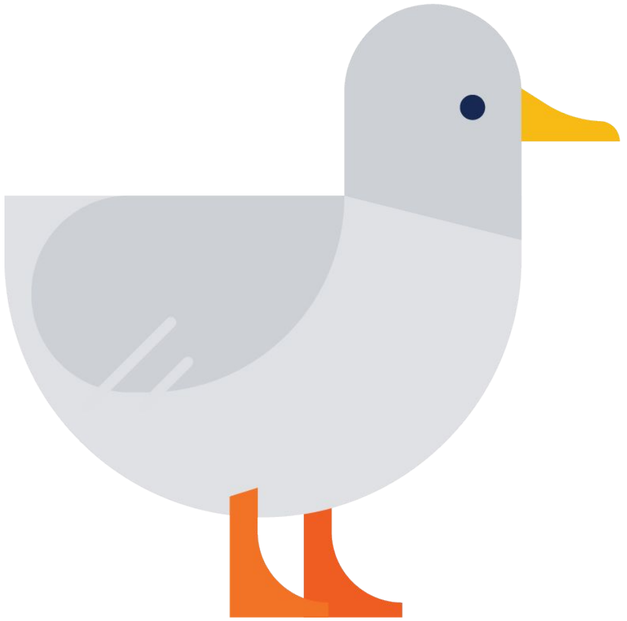
EXPORT_PATH denotes location of SavedModel directory

Make predictions using gcloud ai-platform

```
input.json = {"sq_footage": 3140,
              "type": 'house'}


gcloud ai-platform predict \
    --model propertyprice \
    --version dnn \
    --json-instances input.json
```

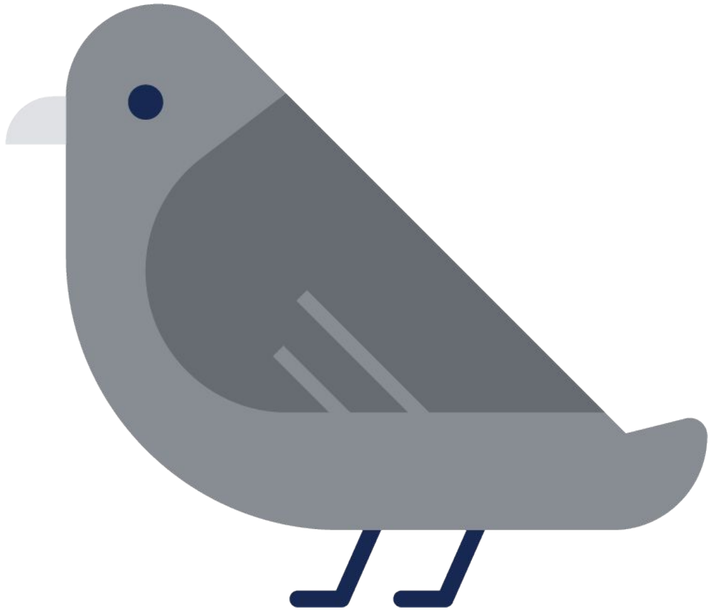
specify the name and version of the deployed model

json file for prediction



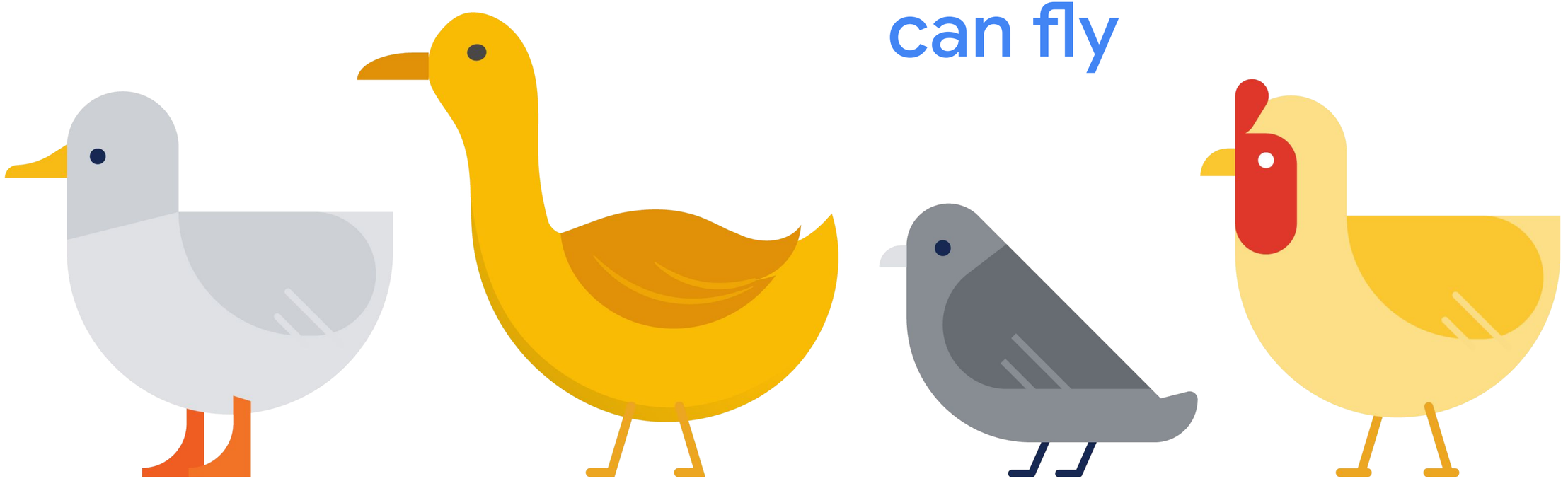
Seagulls can fly

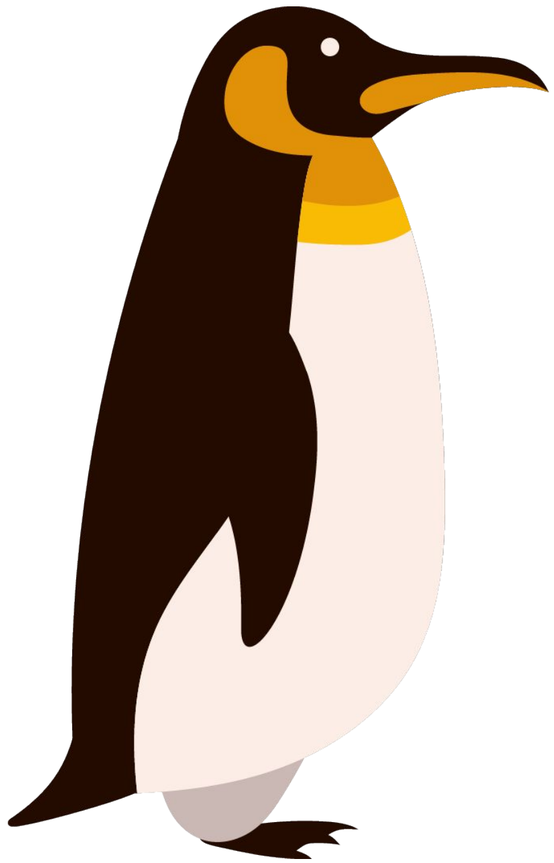
Pigeons can fly



Animals with wings

can fly

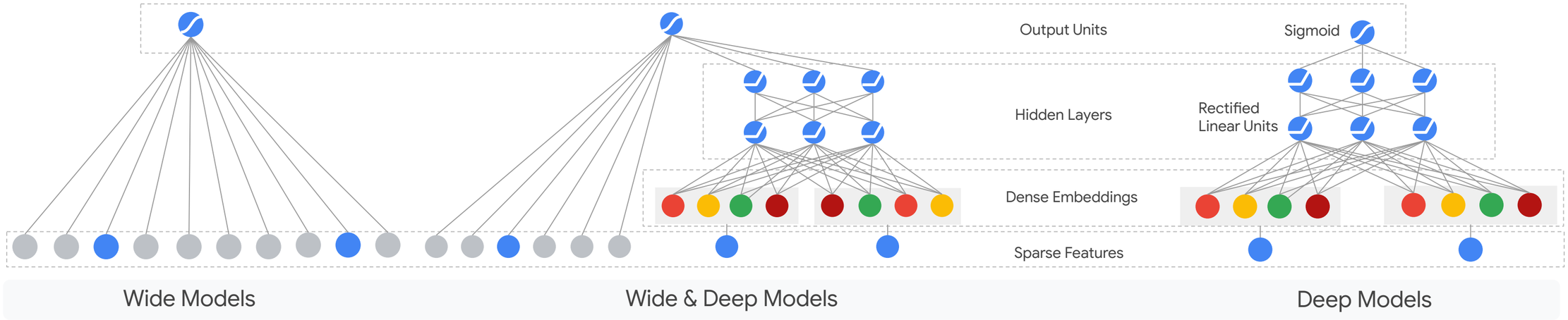




Penguins...

Using Wide and Deep learning

“ Combine the power of memorization and generalization on one unified machine learning model. ”



Memorization + Generalization

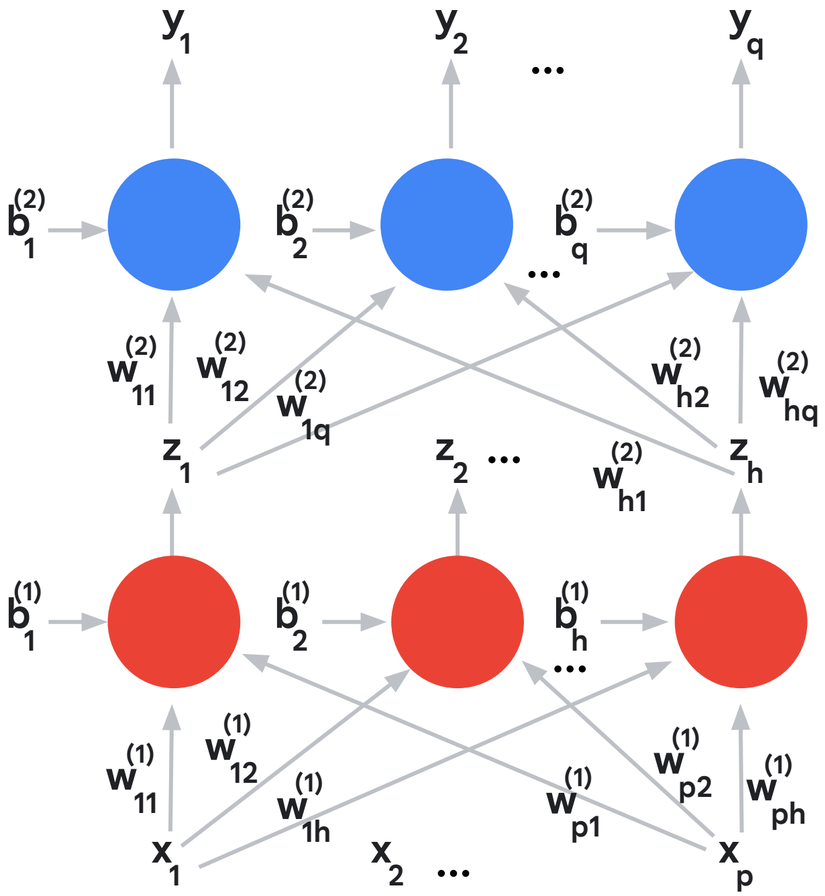
- Memorization: “Seagulls can fly.” “Pigeons can fly.”
- Generalization: “**Animals with wings** can fly.”
- Generalization + memorizing exceptions: “Animals with wings can fly, but penguins cannot fly.”

Linear models are good for sparse, independent features

```
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
```

DNNs are good for dense, highly correlated features

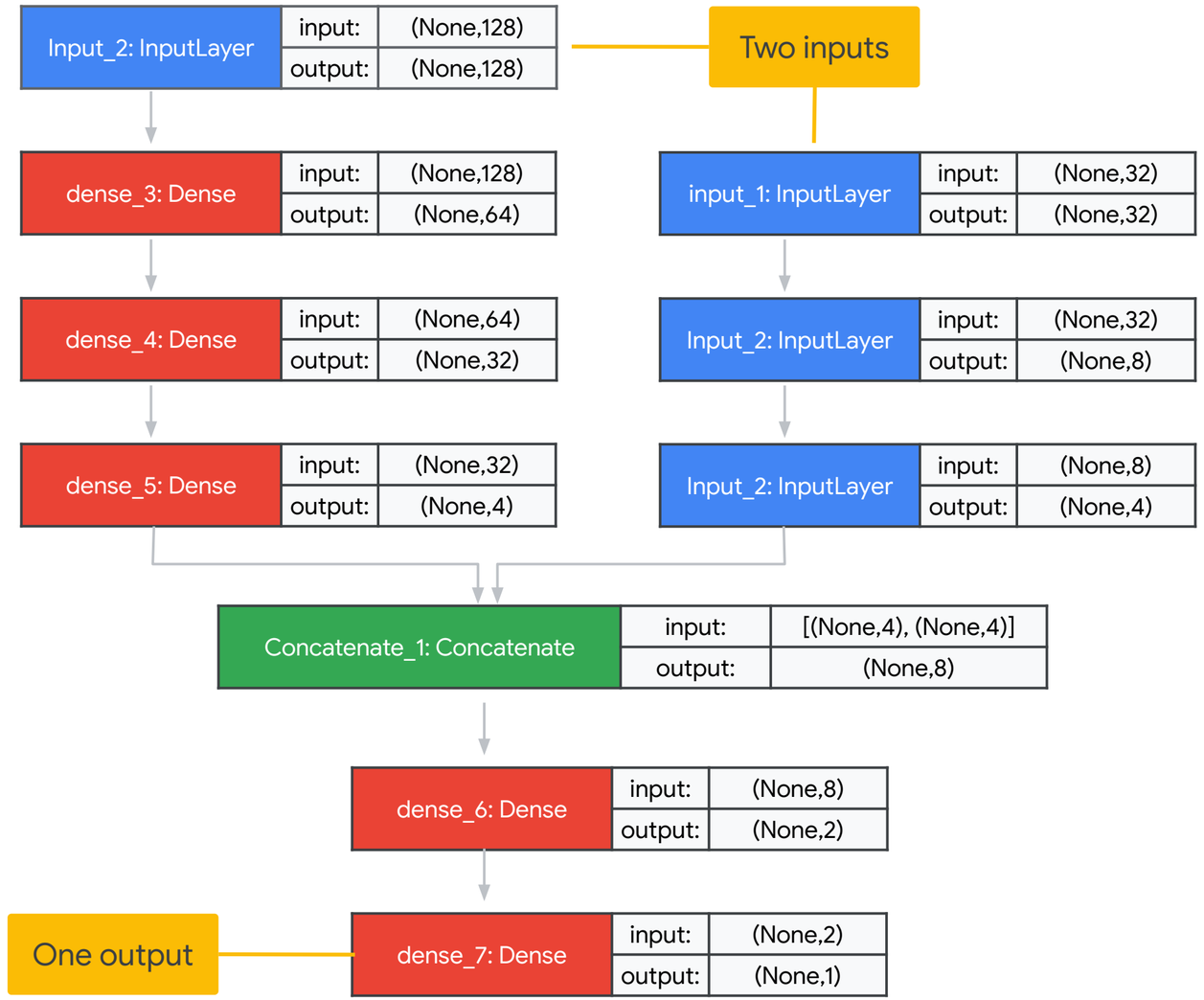
pixel_values ()

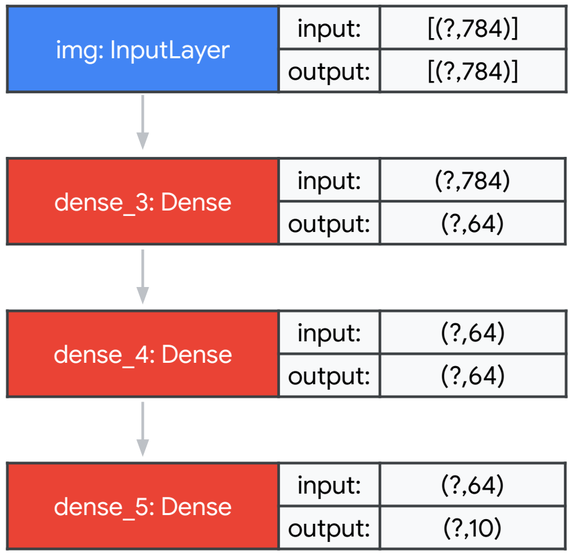


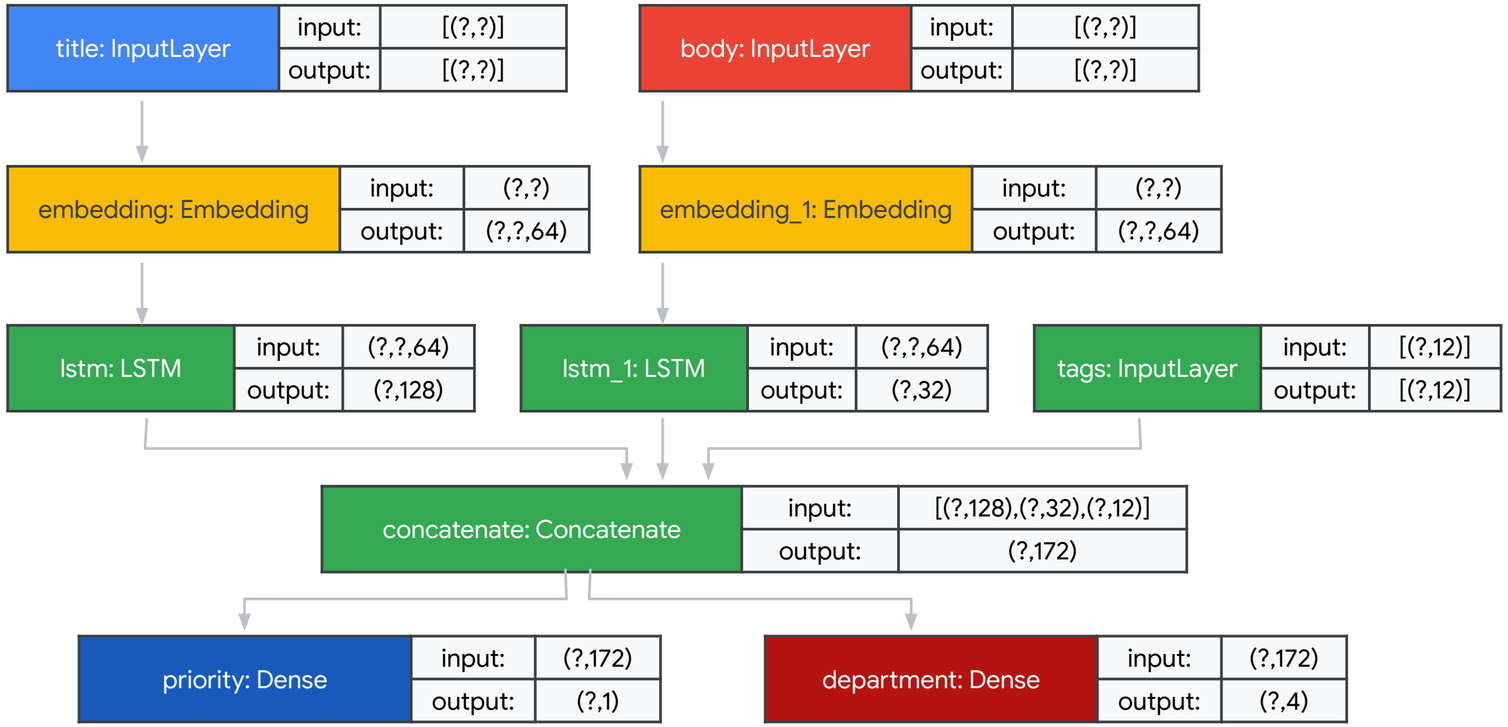
10 hidden nodes -> 10 image features

1024^2 input nodes

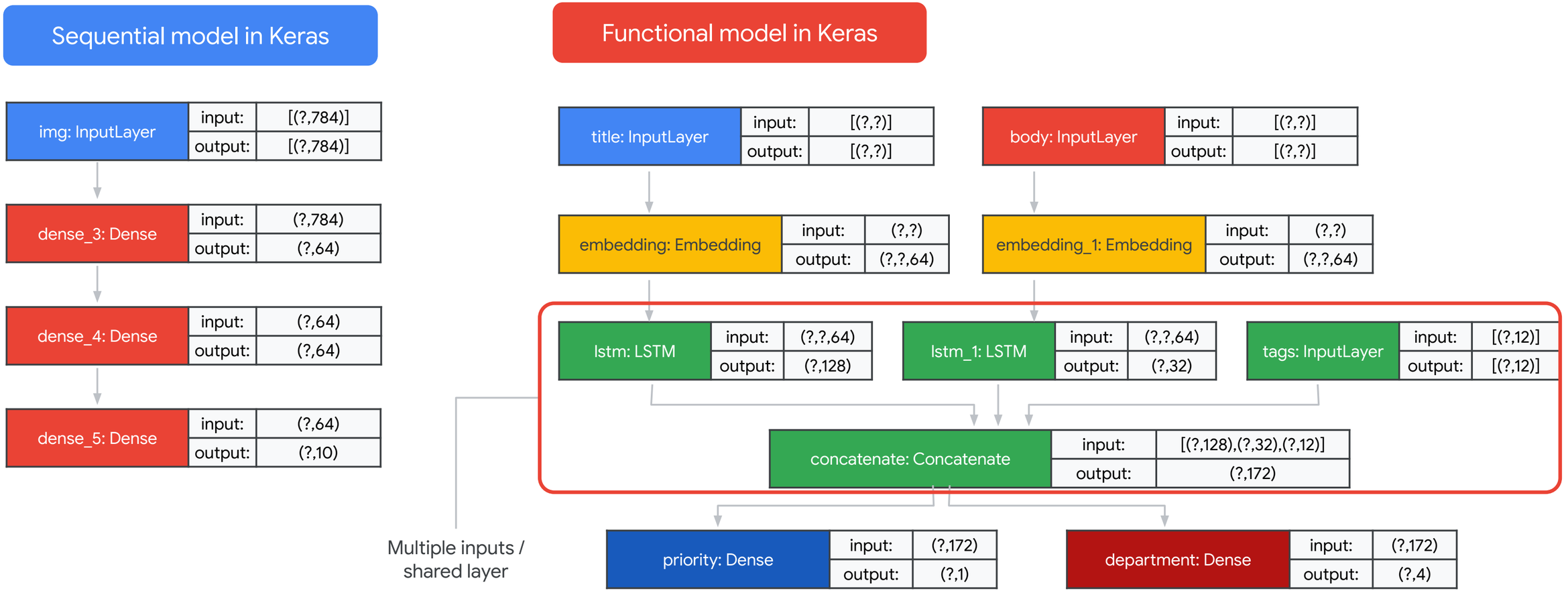
Wide and deep networks using Keras Functional API




img: InputLayer
input: [(?,784)]
output: [(?,784)]
dense_3: Dense
input: (?,784)
output: (?,64)
dense_4: Dense
input: (?,64)
output: (?,64)
dense_5: Dense
input: (?,64)
output: (?,10)

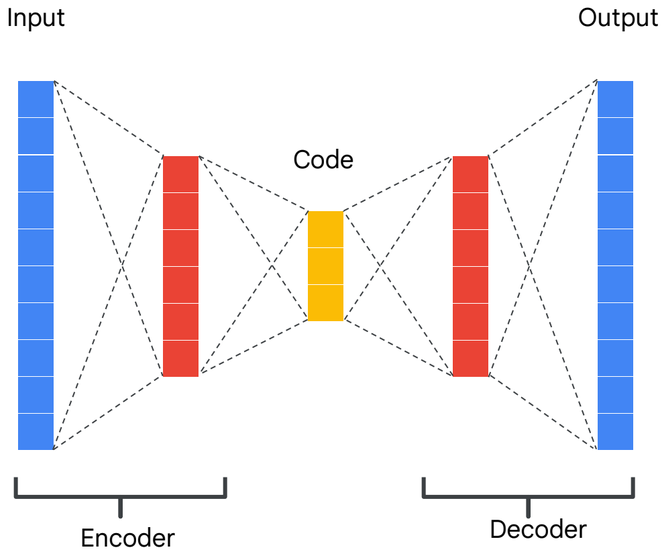

title: InputLayer
input: [(?,?)]
output: [(?,?)]
body: InputLayer
input: [(?,?)]
output: [(?,?)]
embedding: Embedding
input: (?,?)
output: (?,?,64)
embedding_1: Embedding
input: (?,?)
output: (?,?,64)
lstm: LSTM
input: (?,?,64)
output: (?,128)
lstm_1: LSTM
input: (?,?,64)
output: (?,32)
tags: InputLayer
input: [(?,12)]
output: [(?,12)]
concatenate: Concatenate
input: [(?,128),(?,32),(?,12)]
output: (?,172)
priority: Dense
input: (?,172)
output: (?,1)
department: Dense
input: (?,172)
output: (?,4)

Functional API is more flexible than Sequential API



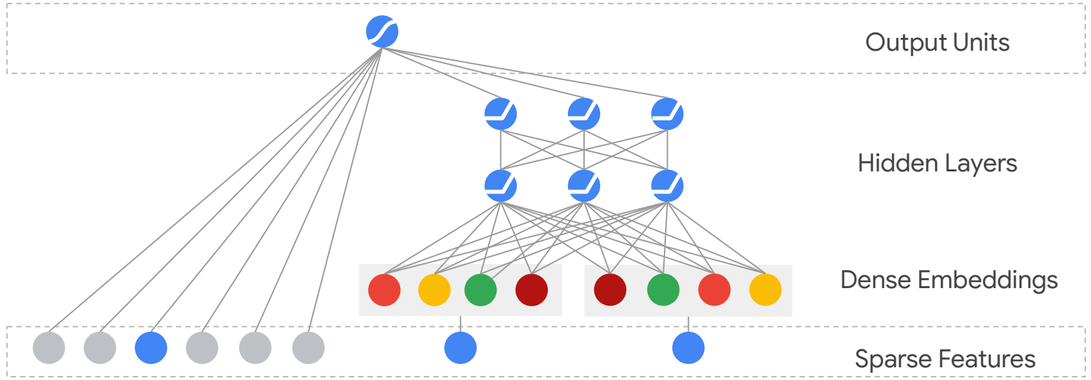
Models are created by specifying their inputs and outputs in a graph of layers

```
encoder_input = keras.Input(shape=(28, 28, 1), name='img')
x = layers.Dense(16, activation='relu')(encoder_input)
x = layers.Dense(10, activation='relu')(x)
x = layers.Dense(5, activation='relu')(x)
encoder_output = layers.Dense(3, activation='relu')(x)

encoder = keras.Model(encoder_input, encoder_output, name='encoder')

x = layers.Dense(5, activation='relu')(encoder_output)
x = layers.Dense(10, activation='relu')(x)
x = layers.Dense(16, activation='relu')(x)
decoder_output = layers.Dense(28, activation='linear')(x)

autoencoder = keras.Model(encoder_input, decoder_output, name='autoencoder')
```



Creating a Wide and Deep model in Keras

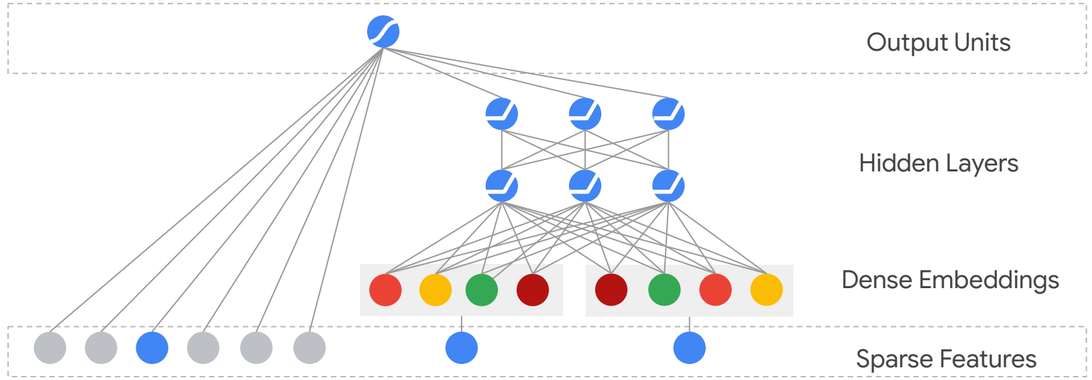
```
INPUT_COLS = [
    'pickup_longitude',
    'pickup_latitude',
    'dropoff_longitude',
    'dropoff_latitude',
    'passenger_count'
]

# Prepare input feature columns
inputs = {colname : layers.Input(name=colname,
shape=(), dtype='float32')
          for colname in INPUT_COLS
}

...
```



Wide & Deep Models

Creating a Wide and Deep model in Keras

```
# Create deep columns
deep_columns = [
    # Embedding_column to "group" together ...
    fc.embedding_column(fc_crossed_pd_pair, 10),

    # Numeric columns
    fc.numeric_column("pickup_latitude"),
    fc.numeric_column("pickup_longitude"),
    fc.numeric_column("dropoff_longitude"),
    fc.numeric_column("dropoff_latitude")]

# Create the deep part of model
deep_inputs = layers.DenseFeatures
      (deep_columns, name='deep_inputs')(inputs)
x = layers.Dense(30, activation='relu')(deep_inputs)
x = layers.Dense(20, activation='relu')(x)

deep_output = layers.Dense(10, activation='relu')(x)
```



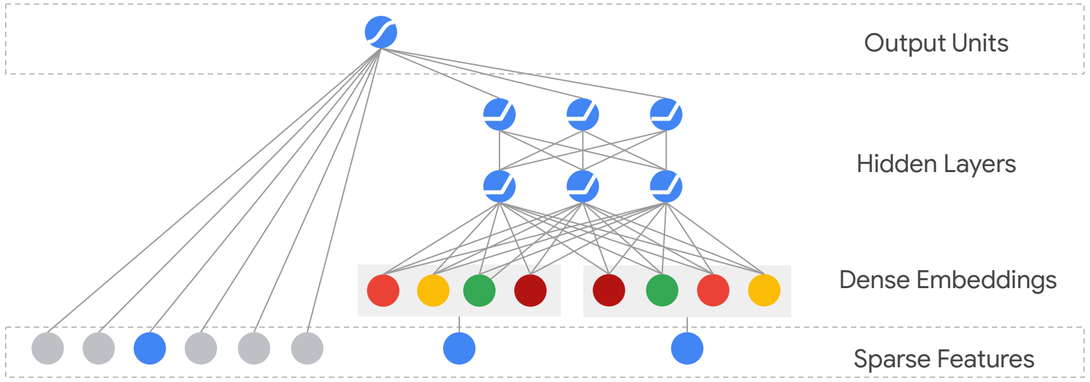
Wide & Deep Models

created in the previous slide

Creating a Wide and Deep model in Keras

```
# Create wide columns
wide_columns = [
    # One-hot encoded feature crosses
    fc.indicator_column(fc_crossed_dloc),
    fc.indicator_column(fc_crossed_ploc),
    fc.indicator_column(fc_crossed_pd_pair)
]

# Create the wide part of model
wide = layers.DenseFeatures(wide_columns,
name='wide_inputs')(inputs)
```

created in the previous slide

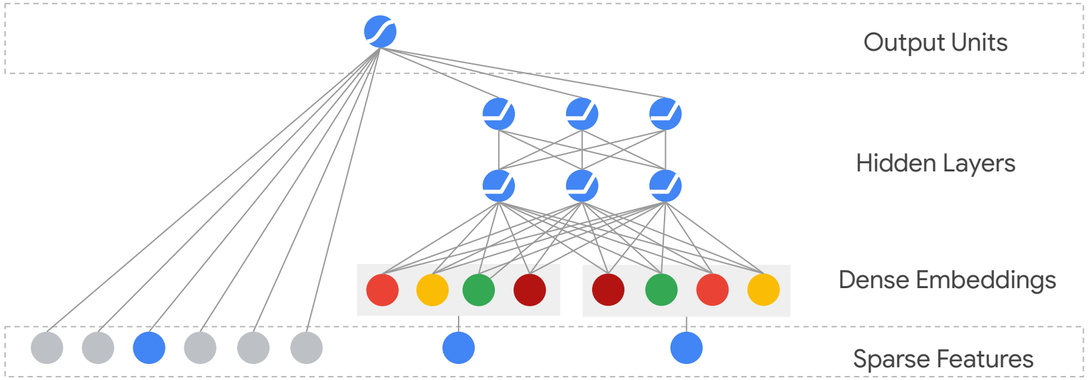


Wide & Deep Models

Creating a Wide and Deep model in Keras

```
# Combine outputs
combined = concatenate(inputs=[deep, wide],
                       name='combined')
output = layers.Dense(1,
                      activation=None,
                      name='prediction')(combined)

# Finalize model
model = keras.Model(inputs=list(inputs.values()),
                    outputs=output,
                    name='wide_and_deep')
model.compile(optimizer="adam",
              loss="mse",
              metrics=[rmse, "mse"])
```

To finalize the model, specify the inputs and outputs



Wide & Deep Models

Strengths and weaknesses of the Functional API

Strengths

- Less verbose than using keras.Model subclasses
- Validates your model while you're defining it
- Your model is plottable and inspectable
- Your model can be serialized or cloned

Weaknesses

- Doesn't support dynamic architectures
- Sometimes you have to write from scratch and you need to build subclasses, e.g. custom training or inference layers

Model building: From simple to arbitrarily flexible

Sequential API

+ Built-in layers

01

New users,
simple models

Functional API

+ Built-in layers

02

Engineers with
standard use cases

Functional API

+ Custom layers
+ Custom metrics
+ Custom losses

03

Engineers requiring
increasing control

Subclassing

Write everything
yourself from scratch

04

Researchers

Progressive disclosure of complexity

Model building: From simple to arbitrarily flexible

Sequential API

+ Built-in layers

01

New users,
simple models

Functional API

+ Built-in layers

02

Engineers with
standard use cases

Functional API

+ Custom layers
+ Custom metrics
+ Custom losses

03

Engineers requiring
increasing control

Subclassing

Write everything
yourself from scratch

04

Researchers

Progressive disclosure of complexity

Model building: From simple to arbitrarily flexible

Sequential API

+ Built-in layers

01

New users,
simple models

Functional API

+ Built-in layers

02

Engineers with
standard use cases

Functional API

+ Custom layers
+ Custom metrics
+ Custom losses

03

Engineers requiring
increasing control

Subclassing

Write everything
yourself from scratch

04

Researchers

Progressive disclosure of complexity

Define layers

```
import tensorflow as tf

class MyModel(tf.keras.Model):

  def_init_(self):
      super(myModel, self)._init_()
      self.dense1 =  tf.keras.layers.Dense(4, activation=tf.nn.relu)
      self.dense2 =  tf.keras.layers.Dense(5, activation=tf.nn.softmax)

  def call(self, inputs):
      x = self.dense1(inputs)
      return self.dense2(x)

  model =  MyModel()
```

Implement the forward pass

```
import tensorflow as tf

class MyModel(tf.keras.Model):

   def_init_(self):
      super(myModel, self)._init_()
      self.dense1 =  tf.keras.layers.Dense(4, activation=tf.nn.relu)
      self.dense2 =  tf.keras.layers.Dense(5, activation=tf.nn.softmax)

   def call(self, inputs):
      x = self.dense1(inputs)
      return self.dense2(x)

   model =  MyModel()
```

Constructor method

```
import tensorflow as tf

class MyModel(tf.keras.Model):

   def_init_(self):
      super(myModel, self)._init_()
      self.dense1 =  tf.keras.layers.Dense(4, activation=tf.nn.relu)
      self.dense2 =  tf.keras.layers.Dense(5, activation=tf.nn.softmax)
```

Constructor

Call method

```
def call(self, inputs):
    x = self.dense1(inputs)
    return self.dense2(x)

model =  MyModel()
```

Call method

The “constructor” and “call method”

```
import tensorflow as tf

class MyModel(tf.keras.Model):

    def_init_(self):
        super(myModel, self)._init_()
        self.dense1 =  tf.keras.layers.Dense(4, activation=tf.nn.relu)
        self.dense2 =  tf.keras.layers.Dense(5, activation=tf.nn.softmax)

    def call(self, inputs):
        x = self.dense1(inputs)
        return self.dense2(x)

    model =  MyModel()
```

Subclass the model directly

Constructor

Call method

The “constructor” and “call method”

```
class MyModel(tf.keras.Model):
    def_init_(self, num_classes=10):
        super(myModel, self)._nit_(name='my_model')
        # Define your layers here.
        self.dense_1 =  layers.Dense(32, activation='relu')
        self.dense_2 =  layers.Dense(num_classes,activation='softmax'

    def call(self, inputs);
        # Define your forward pass here
        # using layers you previously defined in '_init_'
        x = self.dense_1(inputs)
        return self.dense_2(x)
```

Subclass the model directly

Constructor

Call method

The “constructor” and “call method”

```
class MyModel(tf.keras.Model):
    def_init_(self, num_classes=10):
        super(myModel, self)._nit_(name='my_model')
        # Define your layers here.
        self.dense_1 =  layers.Dense(32, activation='relu')
        self.dense_2 =  layers.Dense(num_classes,activation='softmax'

    def call(self, inputs);
        # Define your forward pass here
        # using layers you previously defined in '_init_'
        x = self.dense_1(inputs)
        return self.dense_2(x)
```

Subclass the model directly

Constructor

Call method

The “constructor” and “call method”

```
class MyModel(tf.keras.Model):
    def_init_(self, num_classes=10):
        super(myModel, self)._nit_(name='my_model')
        # Define your layers here.
        self.dense_1 =  layers.Dense(32, activation='relu')
        self.dense_2 =  layers.Dense(num_classes,activation='softmax'

    def call(self, inputs);
        # Define your forward pass here
        # using layers you previously defined in '_init_'
        x = self.dense_1(inputs)
        return self.dense_2(x)
```

Subclass the model directly

Constructor

Call method

Custom training loop

```
model = MyModel()
with tf.GradientTape() as tape:
   logits = model(images, training=True)
   loss_value = loss(logits, labels)
grads =  tape.gradient(loss_value, model.variables)
optimizer.apply_graidents(zip(grads, model.variables))
```

Custom training loop

```
model = MyModel()
with tf.GradientTape() as tape:
   logits = model(images, training=True)
   loss_value = loss(logits, labels)
grads =  tape.gradient(loss_value, model.variables)
optimizer.apply_graidents(zip(grads, model.variables))
```

Model building: From simple to arbitrarily flexible

Sequential API

+ Built-in layers

01

New users,
simple models

Functional API

+ Built-in layers

02

Engineers with
standard use cases

Functional API

+ Custom layers
+ Custom metrics
+ Custom losses

03

Engineers requiring
increasing control

Subclassing

Write everything
yourself from scratch

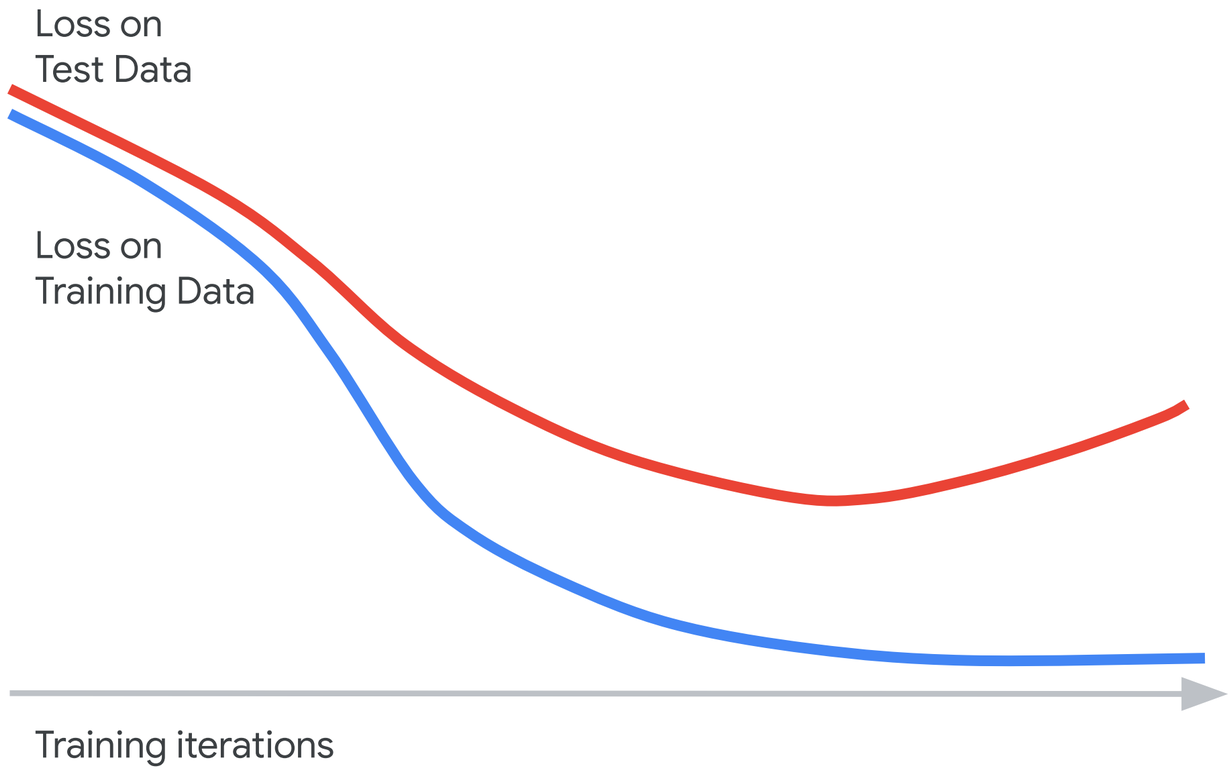
04

Researchers

Progressive disclosure of complexity

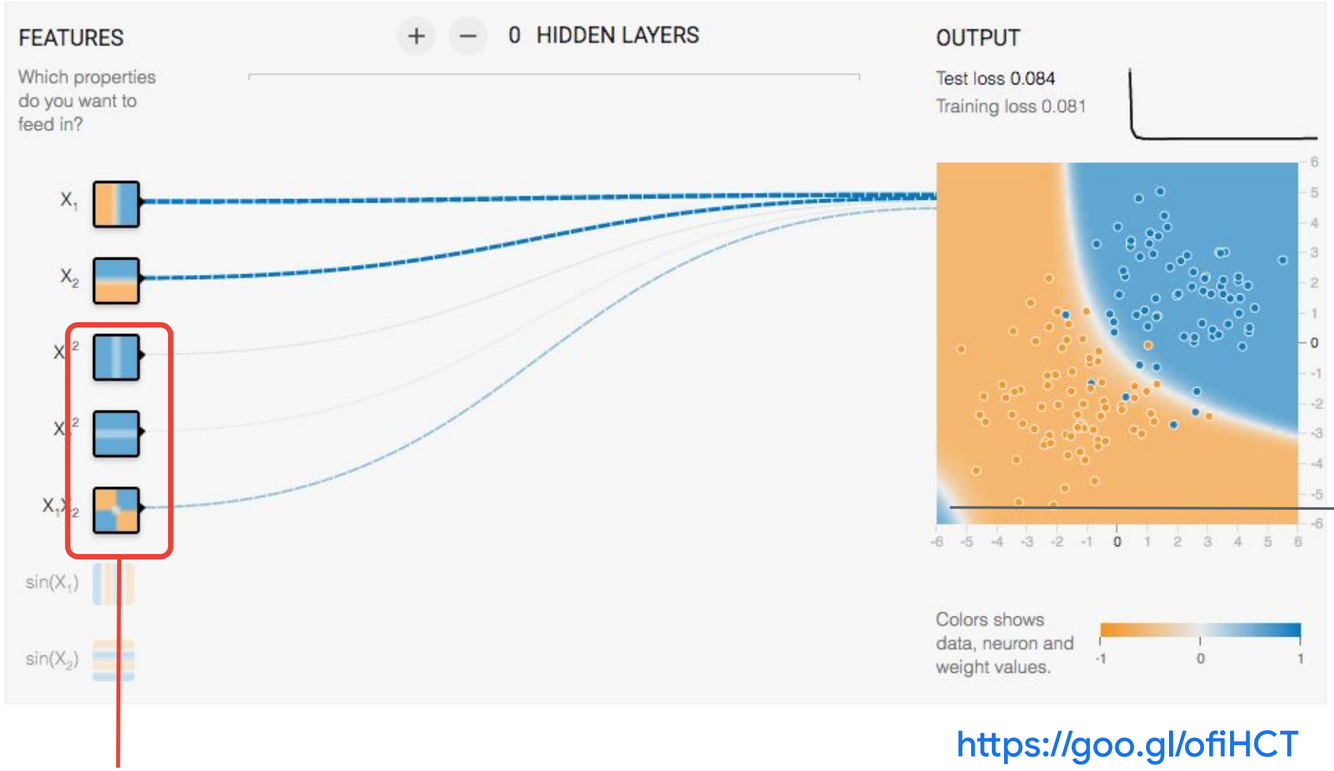
What is happening here?

How can we address this?



Remember the splotch of blue?

Why does it happen?



Why?

https://goo.gl/ofiHCT

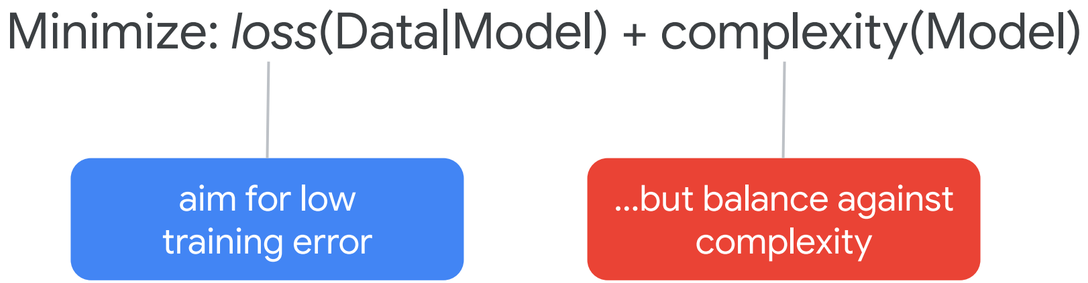
Is the model behavior surprising? What's the issue?
Try removing cross-product features. Does performance improve?

Occam's razor

When presented with competing hypothetical answers to a problem, one should select the one that makes the fewest assumptions.

The idea is attributed to William of Ockham (c. 1287–1347).

Factor in model complexity when calculating error



Optimal model complexity is data-dependent, so requires hyperparameter tuning.

Regularization is a major field of ML research

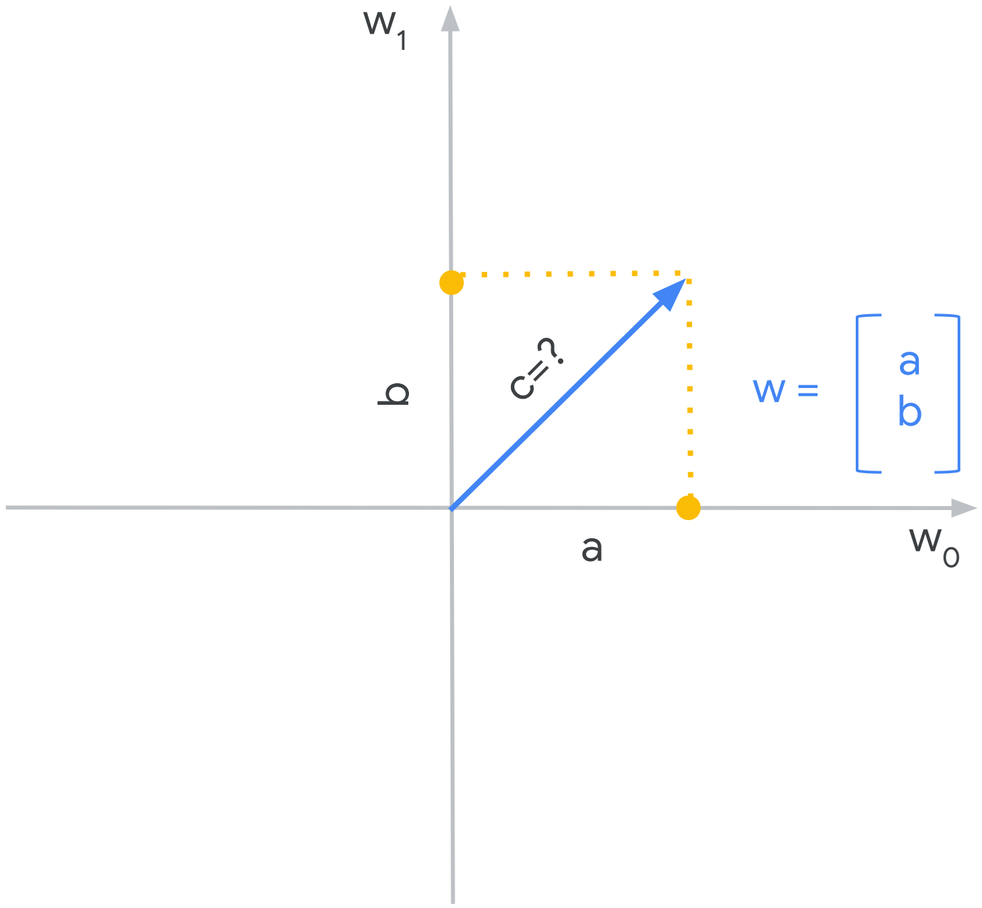
```
Early Stopping
Parameter Norm Penalties
        L1 regularization
        L2 regularization
        Max-norm regularization
Dataset Augmentation
Noise Robustness
Sparse Representations
. . .
```

We will look into these methods.

How can we measure
model complexity?

L2 vs. L1 Norm

$$w = [w_0, w_1, ..., w_n]^T$$

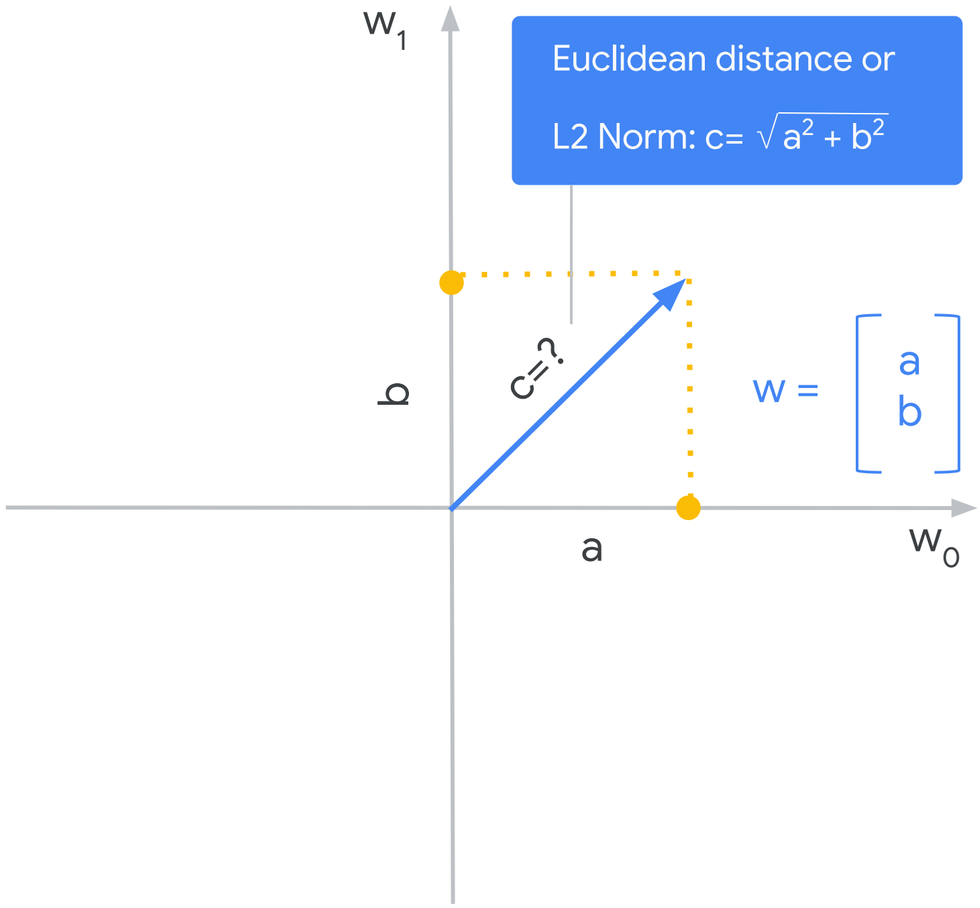


L2 vs. L1 Norm

$$w = [w_0, w_1, ..., w_n]^T$$

$$\|w\|_2 = [w_0^2 + w_1^2 + ... + w_n^2]^{1/2}$$

$$\|w\|_1 = (|w_0| + |w_1| + ... + |w_n|)$$



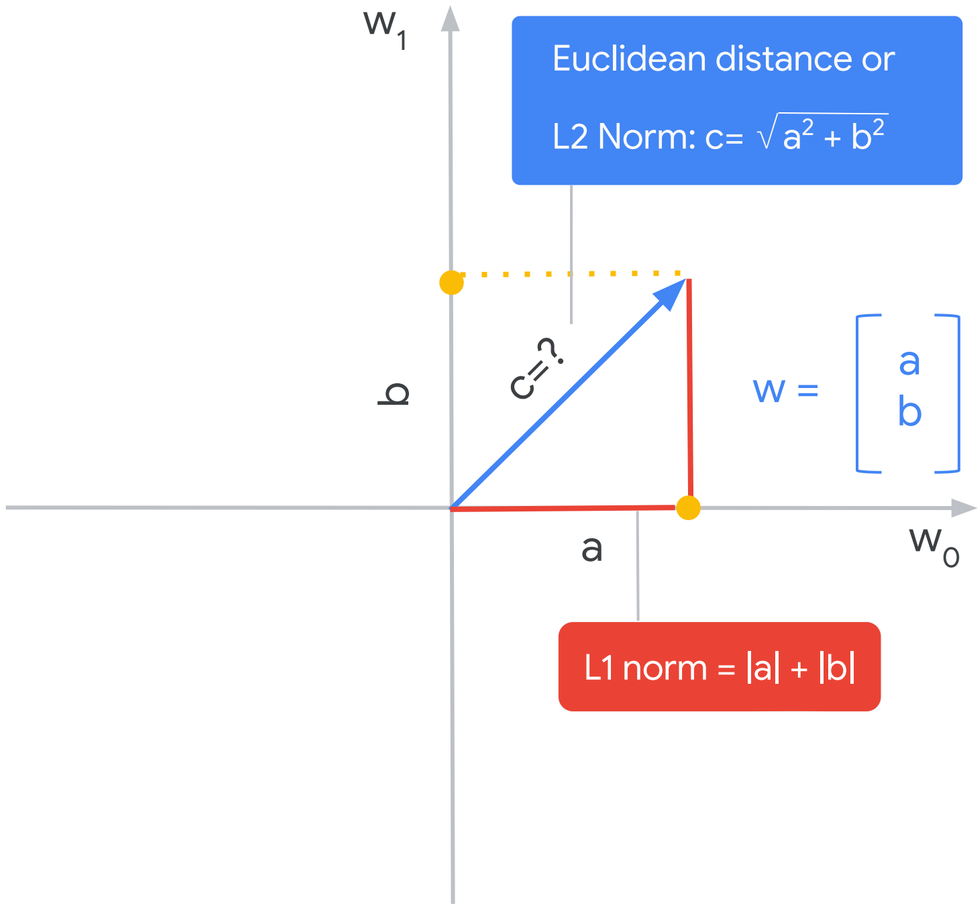
L2 vs. L1 Norm

$$w = [w_0, w_1, ..., w_n]^T$$

$$\|w\|_2 = [w_0^2 + w_1^2 + ... + w_n^2]^{1/2}$$

L2 norm

$$\|w\|_1 = (|w_0| + |w_1| + ... + |w_n|)$$



L2 vs. L1 Norm

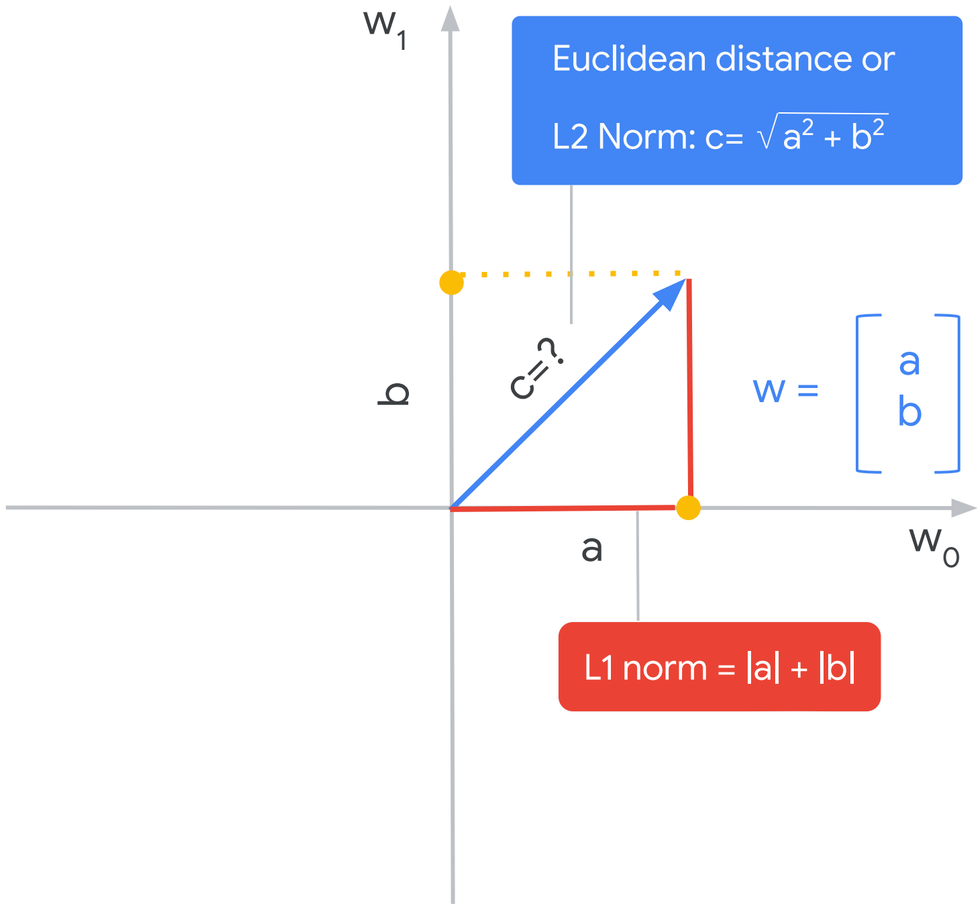
$$w = [w_0, w_1, ..., w_n]^T$$

$$\|w\|_2 = [w_0^2 + w_1^2 + ... + w_n^2]^{1/2}$$

L2 norm

$$\|w\|_1 = (|w_0| + |w_1| + ... + |w_n|)$$

L1 norm



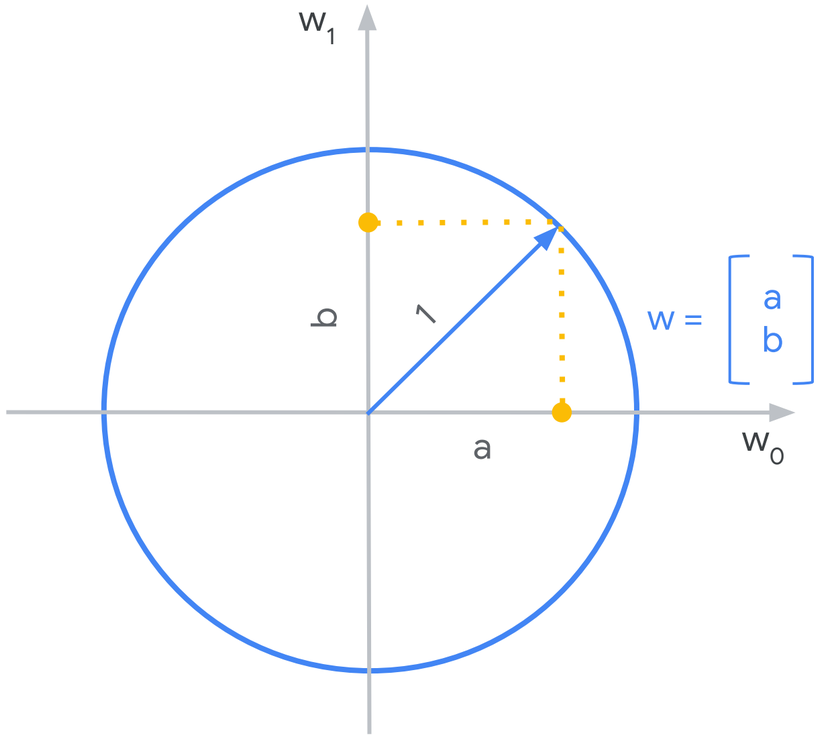
L2 vs. L1 Norm

$$w = [w_0, w_1, ..., w_n]^T$$

$$\|w\|_2 = [w_0^2 + w_1^2 + ... + w_n^2]^{1/2}$$

L2 norm

$$\|w\|_1 = (|w_0| + |w_1| + ... + |w_n|)$$



L2 vs. L1 Norm

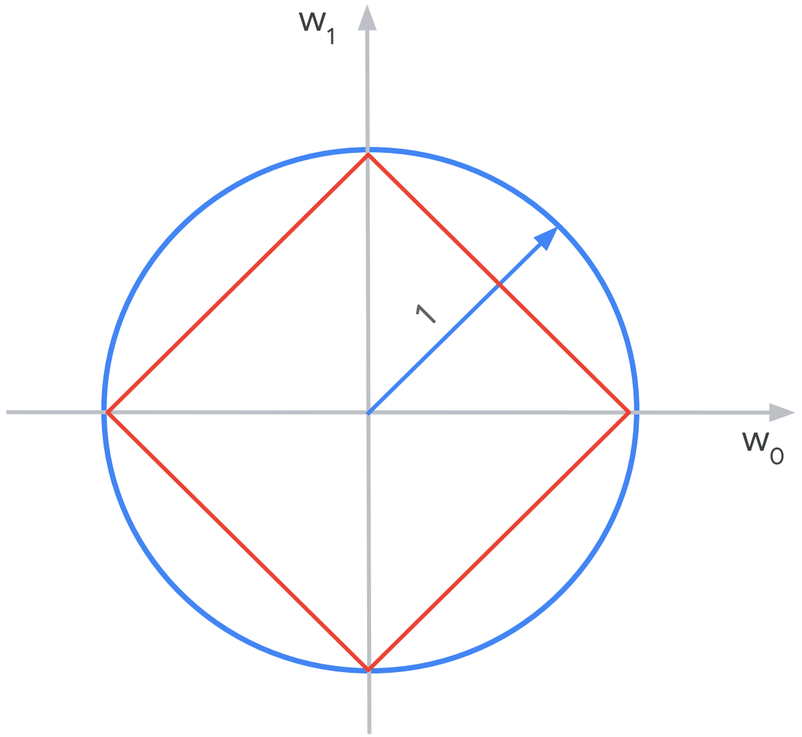
$$w = [w_0, w_1, ..., w_n]^T$$

$$\|w\|_2 = [w_0^2 + w_1^2 + ... + w_n^2]^{1/2}$$

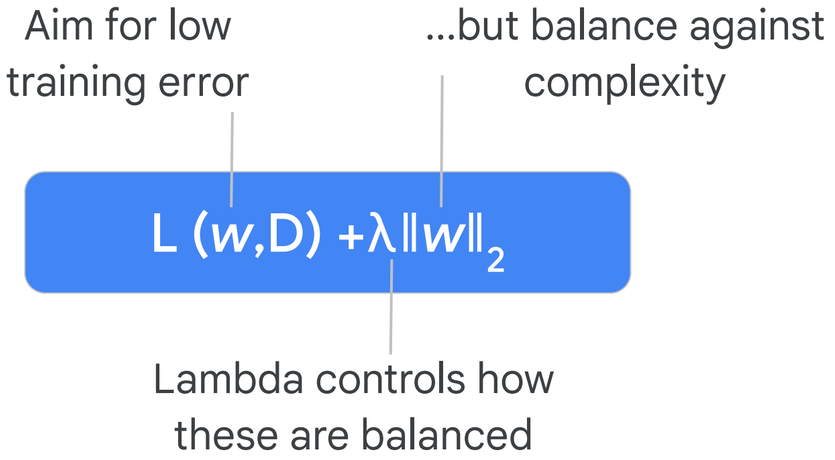
L2 norm

$$\|w\|_1 = (|w_0| + |w_1| + ... + |w_n|)$$

L1 norm



In L2 regularization, complexity of model is defined by the L2 norm of the weight vector



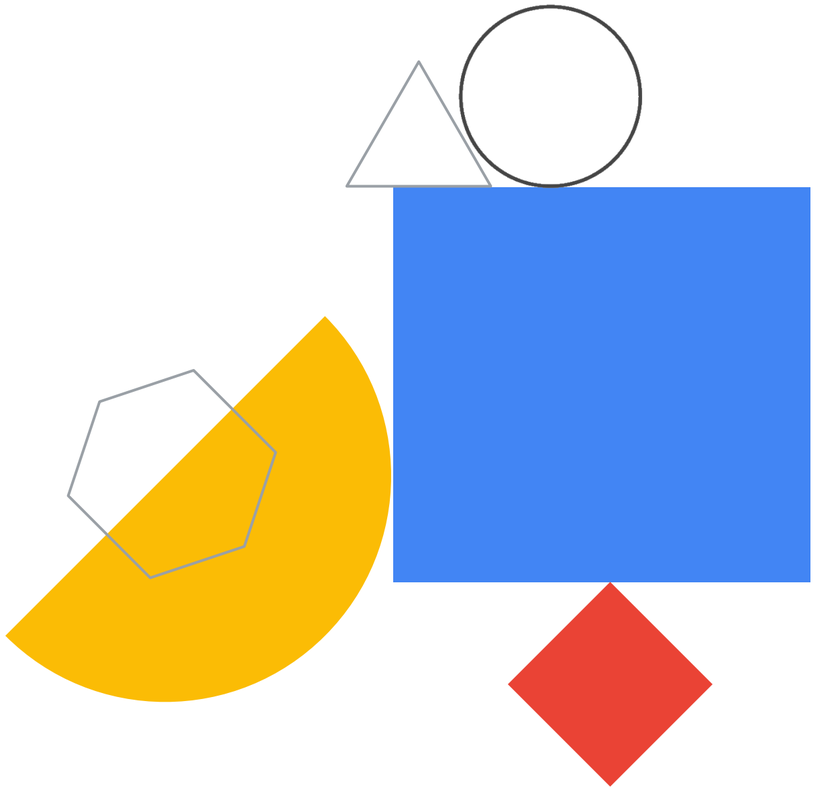
In L1 regularization, complexity of model is defined by the L1 norm of the weight vector

$$L(w,D) + \lambda \|w\|_2$$

L1 regularization can be used as a feature selection mechanism.

Google Cloud

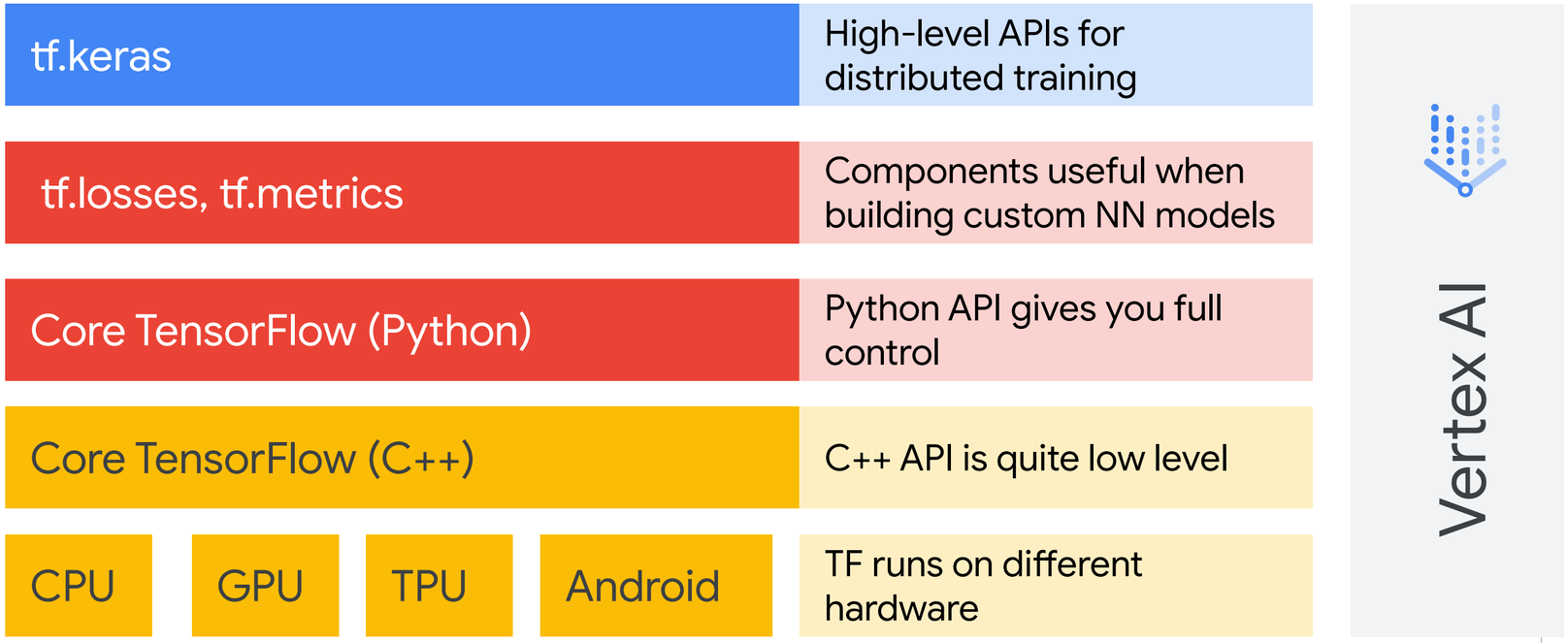
Training at Scale with Vertex AI



In this module, you learn to ...

01	Use TensorFlow to create a training job
02	Package up a TensorFlow model as a Python package
03	Configure, start, and monitor a Vertex AI training job

We will use distributed TensorFlow on Vertex AI



We will use distributed TensorFlow on Vertex AI

1. Use TensorFlow to create your Keras model
2. Package your trainer application
3. Configure and start a Vertex AI training job

Create task.py to parse command-line parameters and send along to train_and_evaluate

model.py

```
def train_and_evaluate(hparams):
    # [...] Obtain param values from hparam

    model = build_dnn_model(nbuckets, nnsize, lr)
    trainds = create_train_dataset(train_data_path, batch_size)
    evalds = create_eval_dataset(eval_data_path, batch_size)

    # [...] create the callbacks
    steps_per_epoch = num_examples_to_train_on // (batch_size * num_evals)

    history = model.fit(
        trainds,
        validation_data=evalds,
        epochs=num_evals,
        steps_per_epoch=max(1, steps_per_epoch),
        verbose=2,  # 0=silent, 1=progress bar, 2=one line per epoch
        callbacks=[checkpoint_cb, tensorboard_cb]
    )
    tf.saved_model.save(model, model_export_path)
    return history
```

task.py

```
parser.add_argument(
     '--train_data_paths', required=True)
parser.add_argument(
     '--train_steps', ...
```

Package up TensorFlow model as Python package

Python modules need to contain an __init__.py in every folder.

```
taxifare/trainer/__init__.py
taxifare/trainer/task.py
taxifare/trainer/model.py
```

Test your code

locally first

```
EVAL_DATA_PATH=./taxifare/tests/data/taxi-valid*
TRAIN_DATA_PATH=./taxifare/tests/data/taxi-train*
OUTPUT_DIR=./taxifare-model

test ${OUTPUT_DIR} && rm -rf ${OUTPUT_DIR}
export PYTHONPATH=${PYTHONPATH}:${PWD}/taxifare

python3 -m trainer.task \
--eval_data_path $EVAL_DATA_PATH \
--output_dir $OUTPUT_DIR \
--train_data_path $TRAIN_DATA_PATH \
--batch_size 5 \
--num_examples_to_train_on 100 \
--num_evals 1 \
--nbuckets 10 \
--lr 0.001 \
--nnsize "32 8"
```

To make our code compatible with Vertex AI

01

Upload data to Google Cloud Storage

02

Move code into a trainer Python package

03

Submit training job with gcloud to train on Vertex AI

02

Move code to trainer Python package

We package our code as a source distribution using setup.py and setuptools.

```
taxifare/trainer/__init__.py
taxifare/trainer/task.py
taxifare/trainer/model.py
taxifare/setup.py
```

02

Move code to trainer Python package

We package our code as a source distribution using setup.py and setuptools.

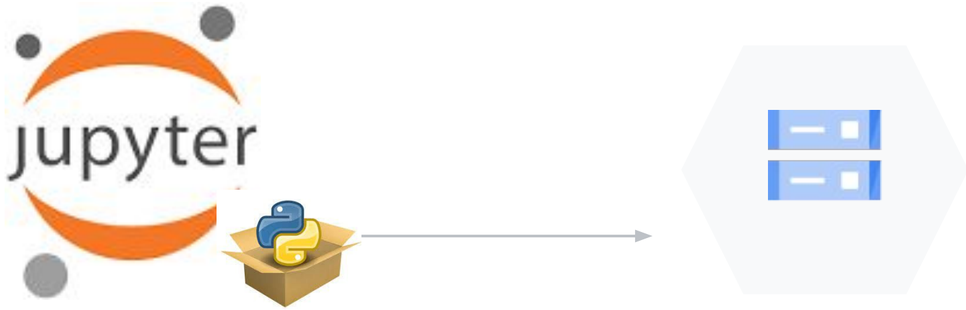
```
taxifare/trainer/__init__.py
taxifare/trainer/task.py
taxifare/trainer/model.py
taxifare/setup.py
```

```
python ./setup.py sdist --formats=gztar
```

02

Copy that Python package to your GCS bucket

```
gsutil cp taxifare/dist/taxifare_trainer-0.1.tar.gz
gs://${BUCKET}/taxifare/
```



03

Submitting a job with gcloud ai custom-jobs create

using a pre-built container....

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \
  --worker-pool-spec=machine-type=$MACHINE_TYPE,\
    replica-count=$REPLICA_COUNT,\
    executor-image-uri=$PYTHON_PACKAGE_EXECUTOR_IMAGE_URI,\
    python-module=$PYTHON_MODULE
```

using a custom container....

```
gcloud ai custom-jobs create \
  --region=$LOCATION \
  --display-name=$JOB_NAME \
  --worker-pool-spec=machine-type=$MACHINE_TYPE,\
    replica-count=$REPLICA_COUNT,\
    container-image-uri=$CUSTOM_CONTAINER_IMAGE_URI
```

03

For distributed training, specify multiple worker-pool-spec

using a pre-built container....

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \

  --worker-pool-spec=machine-type=$MACHINE_TYPE,\
    replica-count=$REPLICA_COUNT,\
    executor-image-uri=$PYTHON_PACKAGE_EXECUTOR_IMAGE_URI,\
    python-module=$PYTHON_MODULE \

  --worker-pool-spec=machine-type=$SECOND_MACHINE_TYPE,\
    replica-count=$SEOOND_REPLICA_COUNT,\
    executor-image-uri=$SEOOND_PYTHON_PACKAGE_EXECUTOR_IMAGE_URI,\
    python-module=$SEOOND_POOL_PYTHON_MODULE
```

03

For our taxifare model, we'll specify the following components...

using a pre-built container....

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \
  --worker-pool-spec=$WORKER_POOL_SPEC \
  --args="$ARGS"
```

```
TIMESTAMP=$(date -u +%Y%m%d_%H%M%S)
JOB_NAME=taxifare_$TIMESTAMP
```

03

For our taxifare model, we'll specify the following components...

using a pre-built container....

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \
  --worker-pool-spec=$WORKER_POOL_SPEC \
  --args="$ARGS"
```

```
PYTHON_PACKAGE_URIS=gs://${BUCKET}/taxifare/taxifare_trainer-0.1.tar.gz
```

03

For our taxifare model, we'll specify the following components...

using a pre-built container....

```
gcloud ai custom-jobs create \
  --region=$REGION \
  --display-name=$JOB_NAME \
  --python-package-uris=$PYTHON_PACKAGE_URIS \
  --worker-pool-spec=$WORKER_POOL_SPEC \
  --args="$ARGS"
```

```
GCS_PROJECT_PATH=gs://$BUCKET/taxifare
DATA_PATH=$GCS_PROJECT_PATH/data
TRAIN_DATA_PATH=$DATA_PATH/taxi-train*
EVAL_DATA_PATH=$DATA_PATH/taxi-valid*

BATCH_SIZE=50
NUM_EXAMPLES_TO_TRAIN_ON=5000
NUM_EVALS=100
NBUCKETS=10
LR=0.001
NNSIZE="32 8"

ARGS="--eval_data_path=$EVAL_DATA_PATH,\
--output_dir=$OUTDIR,\
--train_data_path=$TRAIN_DATA_PATH,\
--batch_size=$BATCH_SIZE,\
--num_examples_to_train_on=$NUM_EXAMPLES_TO_TRAIN_ON,\
--num_evals=$NUM_EVALS,\
--nbuckets=$NBUCKETS,\
--lr=$LR,\
--nnsize=$NNSIZE"
```

03

You can also use config.yaml to control training parameters

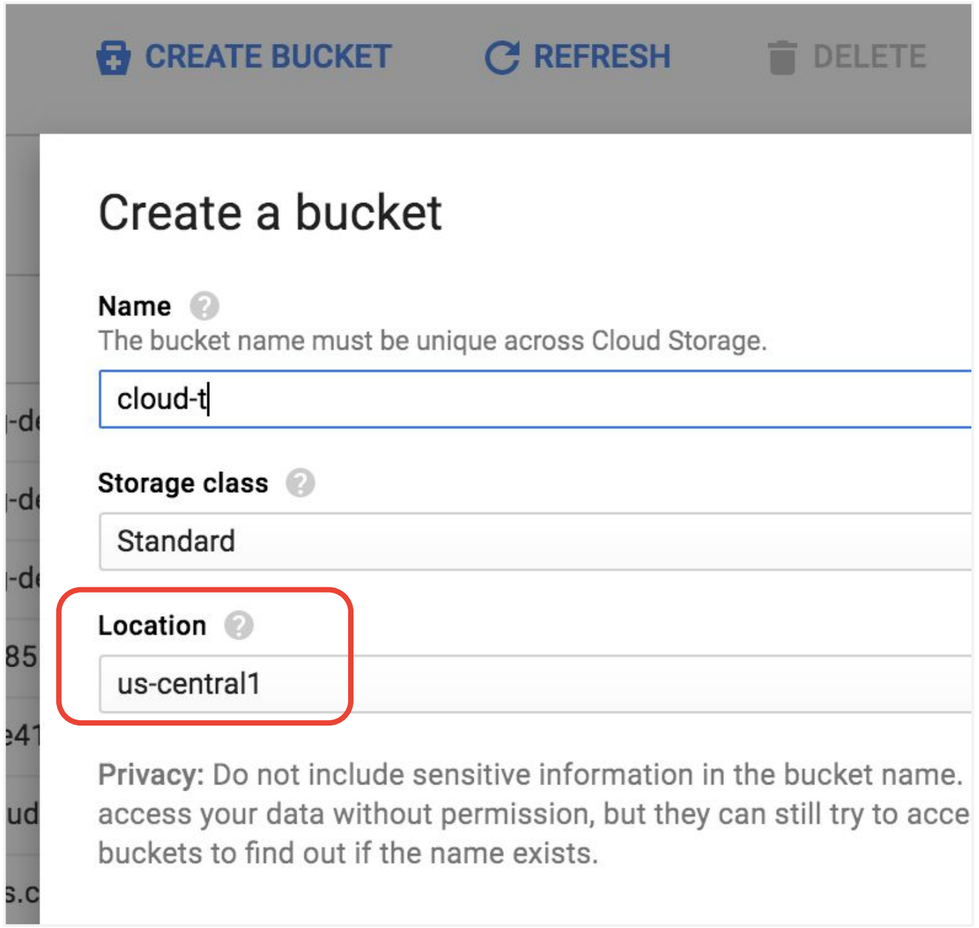
using a config.yaml file....

```
gcloud ai custom-jobs create \
    --region=$LOCATION \
    --display-name=$JOB_NAME \
    --config=config.yaml
```

config.yaml

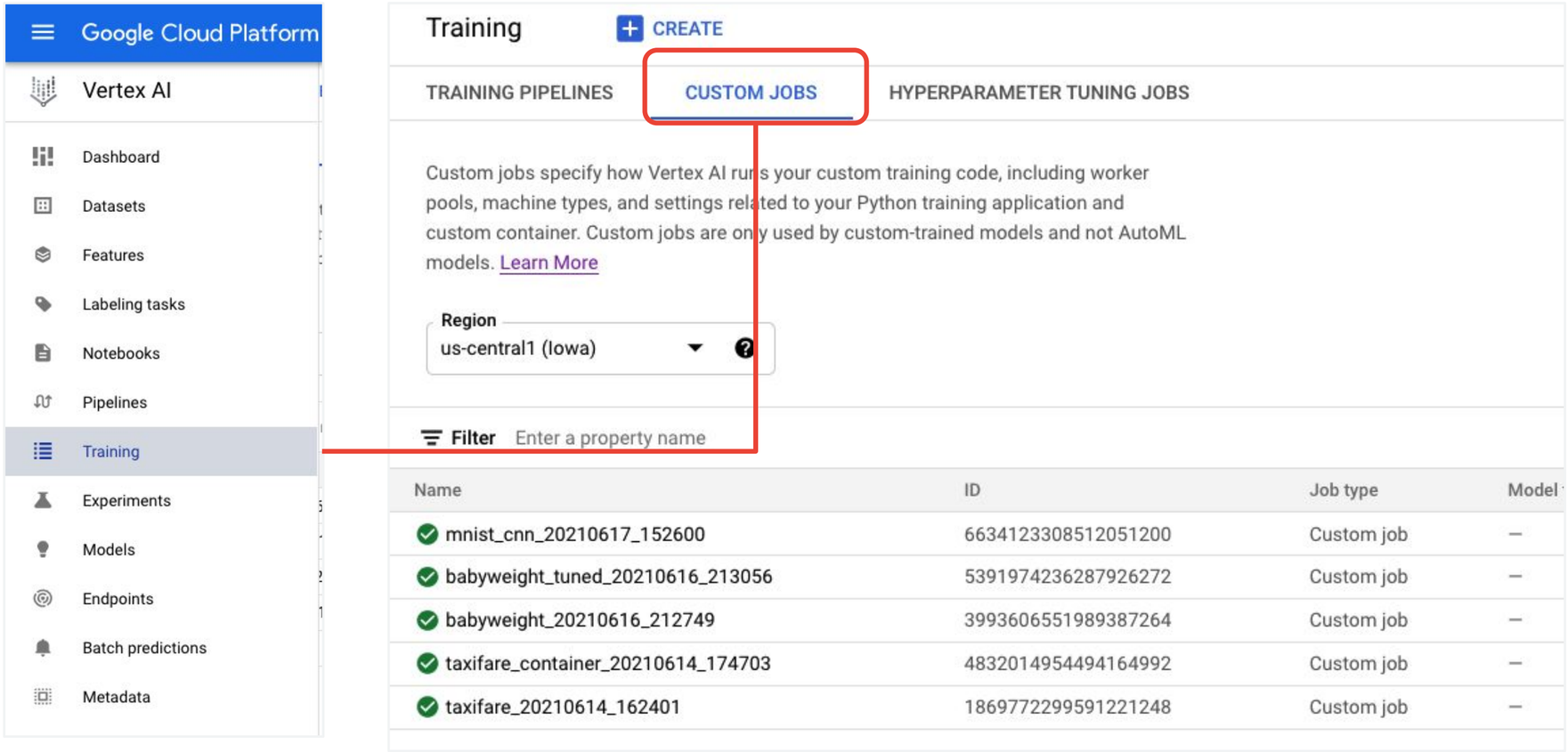
```
workerPoolSpecs:
  machineSpec:
    machineType: n1-highmem-2
  replicaCount: 1
  containerSpec:
    imageUri:
gcr.io/ucaip-test/ucaip-training-test
    args:
    - port=8500
    command:
    - start
```

Tip:
Use single-region bucket for ML



Monitor training jobs with GCP console

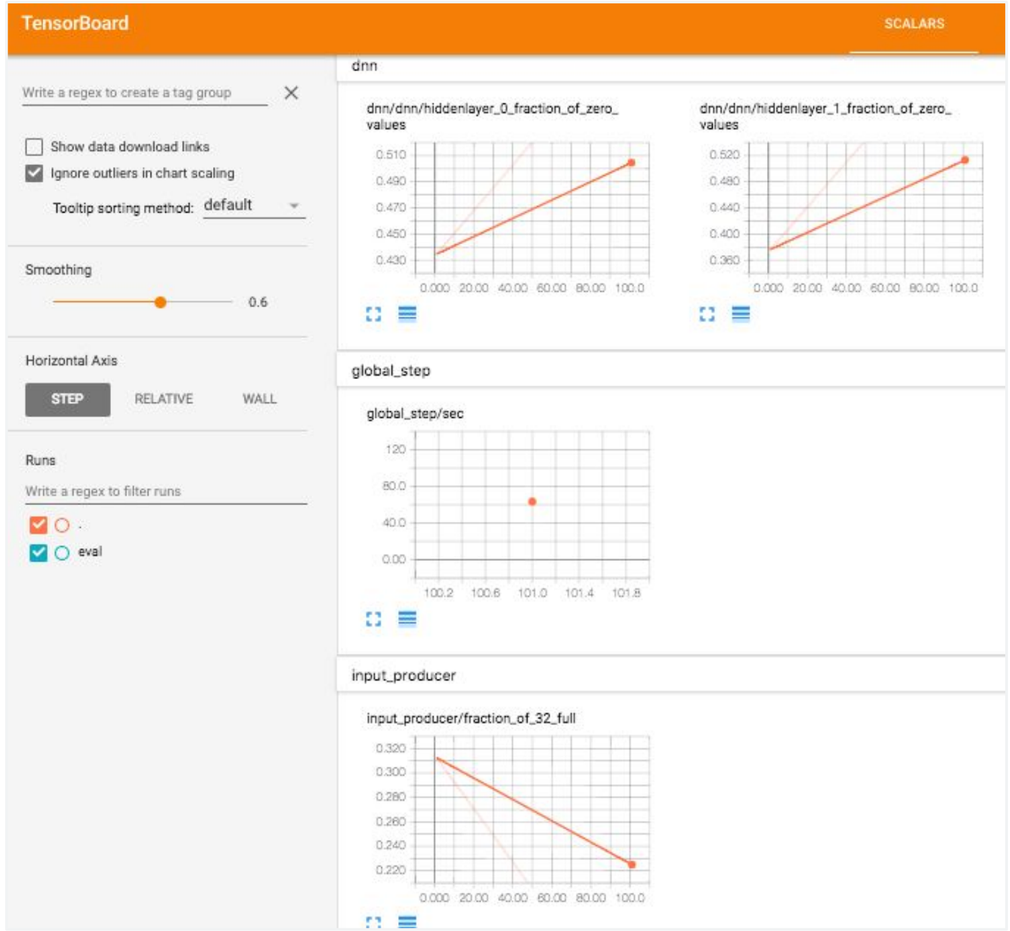
You can also view CPU and Memory utilization charts for this training job with StackDriver Monitoring.



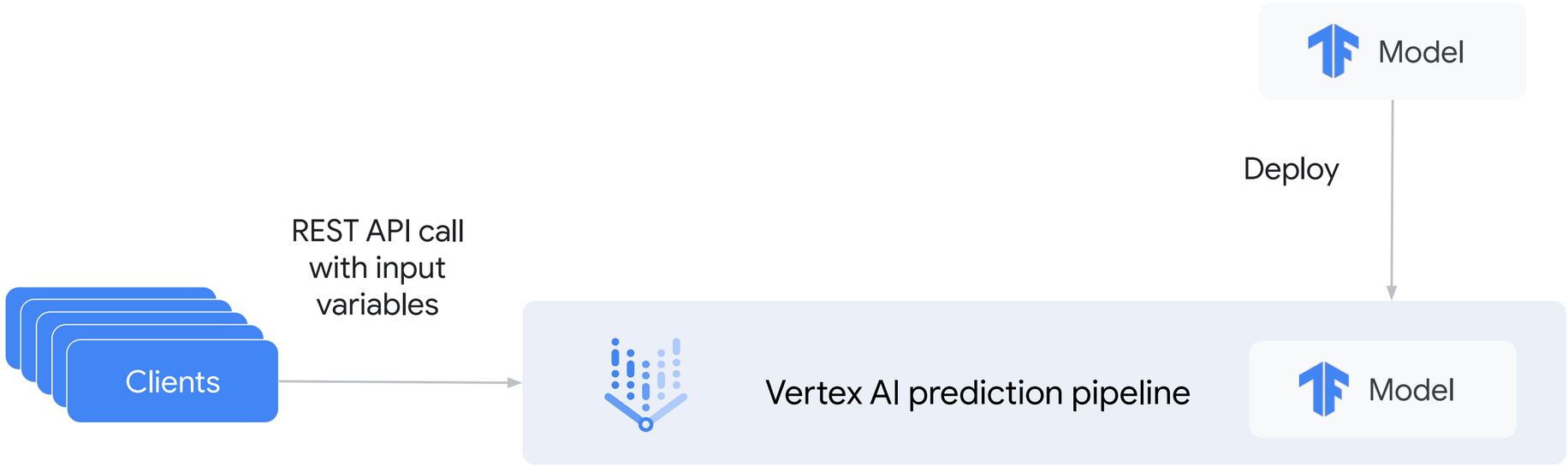
Monitor training jobs with TensorBoard

Configure your trainer to save summary data that you can examine and visualize using TensorBoard.

```
tensorboard_cb = callbacks.TensorBoard(tensorboard_path)

history = model.fit(
    trainds,
    validation_data=evalds,
    epochs=num_evals,
    steps_per_epoch=max(1, steps_per_epoch),
    verbose=2,  # 0=silent, 1=progress bar, 2=one line per epoch
    callbacks=[checkpoint_cb, tensorboard_cb]
)
```



Vertex AI Prediction service makes deploying models and scaling the infrastructure easy



We'll see how to deploy with models and endpoints and make predictions in the later labs