Google Cloud

Day 3

Introduction to Tensorflow

In this module, we will be going over some of the main concepts in Tensorflow 2. We will be looking at tensors, variables, and the current API hierarchy. We will then have a deep dive in the tf.data API and learn how to create input pipelines for models that train on data that is in-memory and also data that lives in multiple files. Lastly, we will learn how feature columns are used to manipulate and prepare data so it can be used to train neural network models.

Machine Learning with TensorFlow on GCP

How Google does Machine Learning

Launching into ML

Introduction to TensorFlow

Feature Engineering

The Art and Science of ML

This is the third chapter of the *Machine Learning with TensorFlow on GC* specialization.

Agenda

- Introduction to Tensorflow
- Creating Input Pipelines and Training on Large Datasets and
- Feature Columns
- Activation Functions
- DNNs with Tensorflow 2 and Keras
- Regularization
- Deploy models for scaled serving

In this course we will be talking about Tensorflow. We first start with an introduction to the framework and preview its main components as well as the overall API hierarchy. Tensorflow 2.x was launched with tight integration of Keras, eager execution by default, and Pythonic function execution, among other new features and improvements. By the way, when I say 2.X it means everything you'll learn here is for the second major release of Tensorflow (that's the 2) and for any minor releases (that's the X). As of the time of this recording the latest stable release candidate was 2.2 but you very well may be on a future minor release version. The principles and syntax still apply.

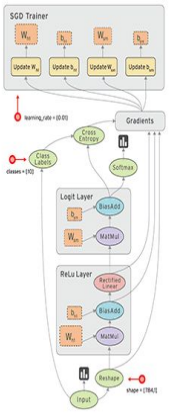
Next, we will discuss how to train on large datasets using the Dataset API; how to use feature columns to prepare the data for training; and how activation functions are needed in order for the model to be able to learn nonlinearities in the data.

We introduce the tf.keras API… tf.keras is TensorFlow's high-level API for building and training deep learning models. We will explore

the Sequential and Functional APIs and learn how to use them to create deep learning models.

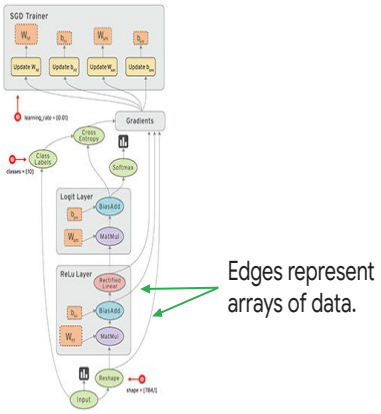
Finally, we discuss how to deploy models for scaled serving using the Cloud AI Platform managed service.
Let's get started.

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



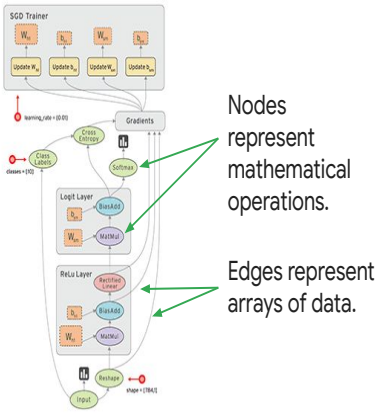
TensorFlow is an open-source, high-performance library for numerical computation. ANY numeric computation -- Not just about machine learning. In fact, people have used TensorFlow for all kinds of GPU computing; for example, you can use TensorFlow to solve partial differential equations which is super useful in fields like fluid dynamics. TensorFlow as a numeric programming library is appealing because you can write your computation code in a high-level language like Python and have it be executed in a fast way at run time.

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



The way TensorFlow works is that you create a directed graph (a DAG) to represent the computation you want to do. In this schematic, *the nodes, like those light green circles, represent mathematical operations* -- things like adding, subtracting, and multiplying. You'll also see some more complex math functions like softmax and matrix-multiplication which are great for machine learning.

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

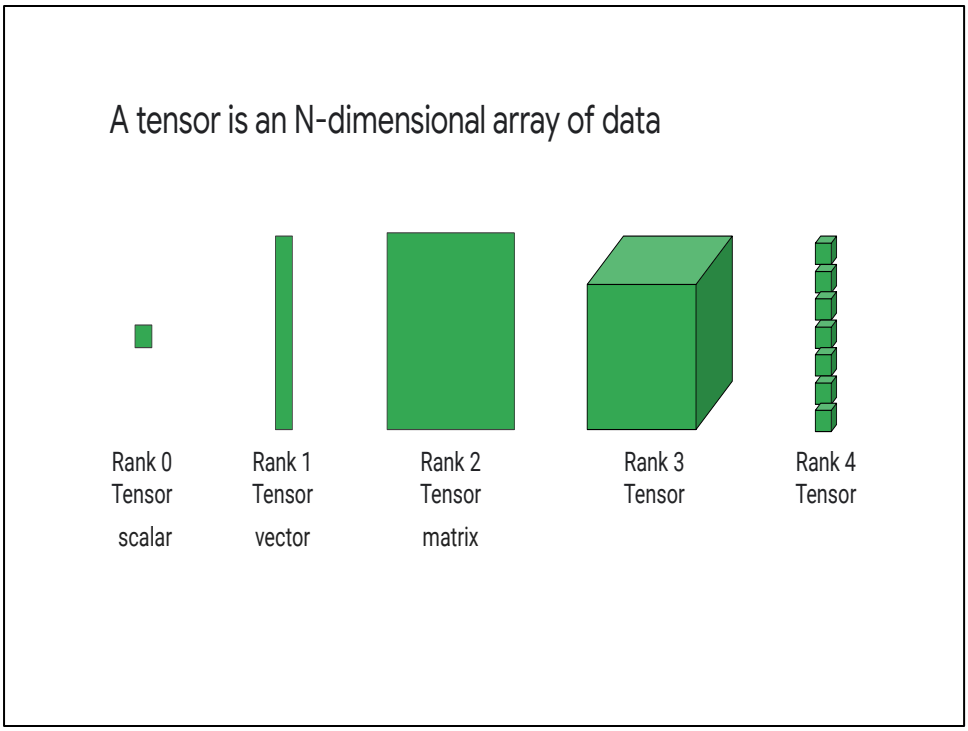


Connecting the nodes are the edges which are the input and output of the mathematical operations. The *edges represent arrays of data flowing towards the output*.

Starting from the bottom are arrays of raw input data. Sometimes we will need to reshape it before feeding it into a layers of a neural network (like the ReLu layer here). More on ReLu later. Once inside that ReLu layer the weight is multiplied across the array of data in a MatMul or matrix multiplication. Then a bias term is added and the data flows through to the activation function.

Wow -- ReLu, activation functions? Don't worry, let's start with the basics. I kept mentioning an array of data flowing around -- what exactly does that mean?

GIF: https://www.tensorflow.org/images/tensors_flowing.gif



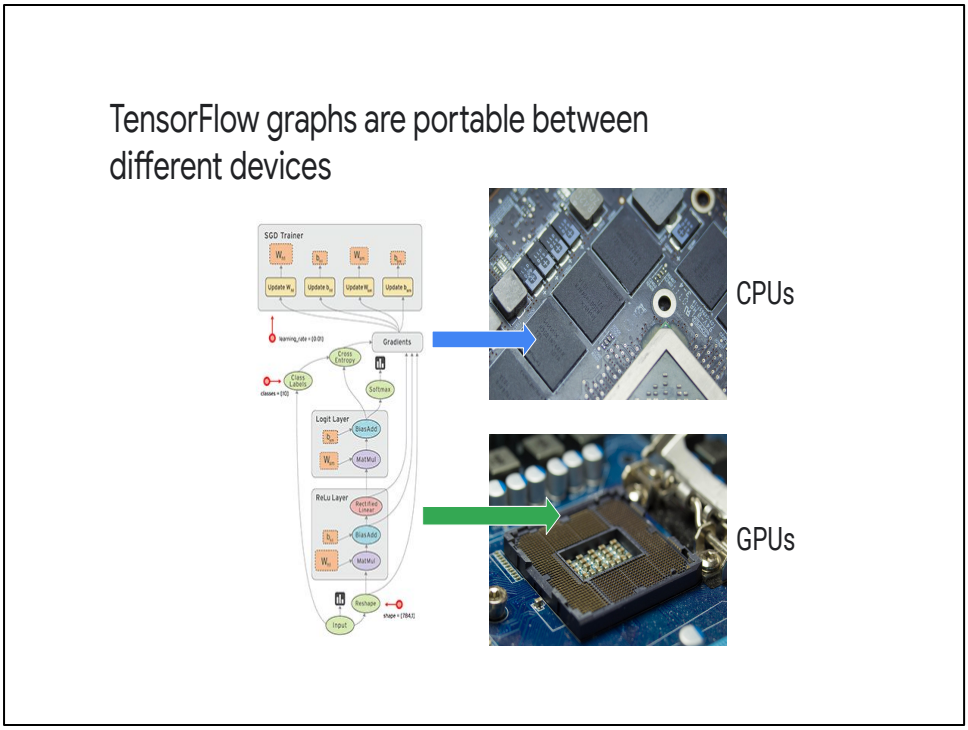
Well that's actually where TensorFlow gets it's name from!

Starting on the far left -- the simplest piece of data we can have is a scalar. That's a number like "3" or "5". It's what we call zero dimensional or Rank 0. We're not going to get very far passing around single numbers in our flow so let's upgrade.

Rank 1 or 1 dimensional array is a vector. In physics, a vector is something with magnitude and direction.
But in Computer Science, you use vector to mean 1D arrays like a series of numbers in a list.

Let's keep going. A two-dimensional array is a matrix. A three-dimensional array? We just call it a 3D tensor. So scalar, vector, matrix, 3D Tensor, 4D Tensor, etc.

A tensor is an n-dimensional array of data. So, your data in TensorFlow are tensors. They flow through the graph. Hence the name TensorFlow.

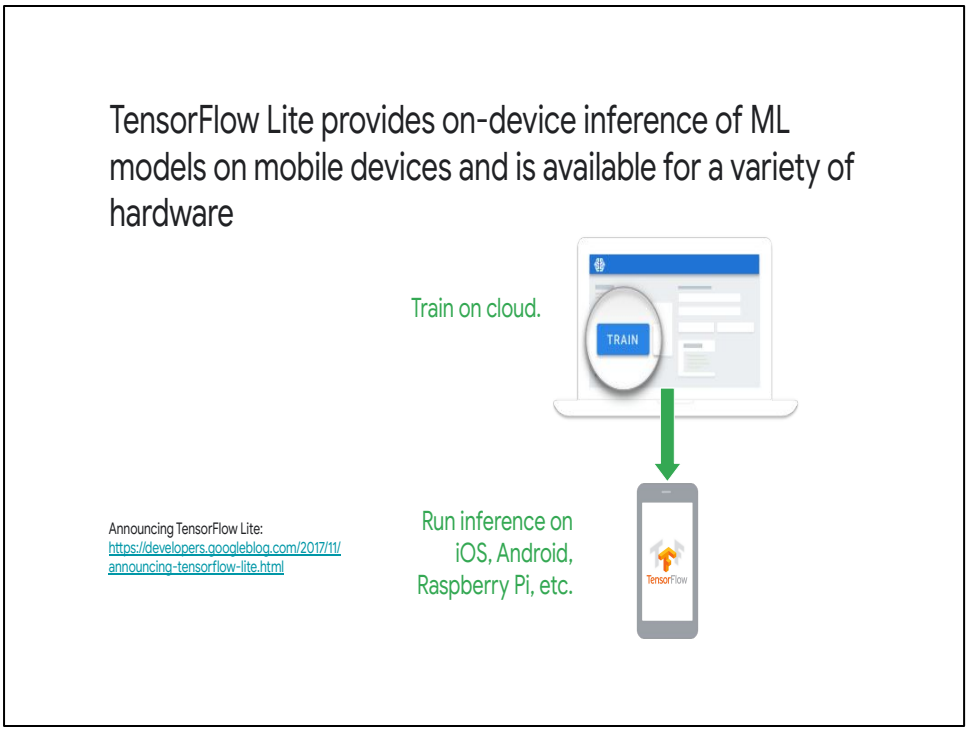


So, why does TensorFlow use directed graphs to represent computation? The answer is portability.

The directed graph is a language-independent representation of the code in your model. You can build a DAG in Python, store it in a SavedModel, and restore it in a C++ program for low-latency predictions. You can use the same Python code and execute it on both CPUs, GPUs, and TPUs. This provides language-and-hardware portability.

In a lot of ways, this is similar to how the Java Virtual Machine (JVM) and its bytecode representation helps the portability of Java code. As a developer, you get to write code in a high-level language (Java) and have it be executed in different platforms by the JVM. The JVM itself is very efficient and targeted toward the exact OS and hardware and written in C or C++.

It's a similar thing with TensorFlow. As a developer, you get to write code in a high-level language (Python) and have it be executed in different platforms by the TensorFlow execution engine. The TensorFlow execution engine is very efficient and targeted toward the exact hardware chip and its capabilities, and is written in C++.



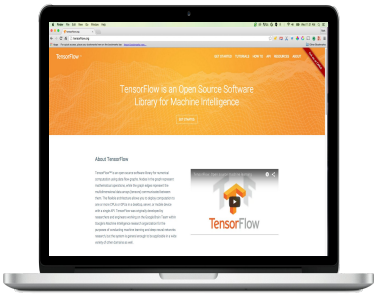
Portability between devices enables a lot of power and flexibility. For example, this is a common pattern. You can train a TensorFlow model on the cloud, on lots and lots of powerful hardware. Then, take the trained model and put it on a device out at the edge -- perhaps a mobile phone or even an embedded chip. You can then do predictions with the model right on that device itself.

Have you had a chance to use the Google Translate app on an Android phone? That app can work completely offline because the trained translation model is stored on the phone and is available for offline translation. Due to the limitations of the processing power available on phones, the edge model tends to be a smaller, therefore less powerful than what's on the cloud. However, the fact that TensorFlow allows for models to run on the edge propitiate a much faster response during predictions.

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

Image (train on cloud) cc0: https://cloud.google.com/products/machine-learning/

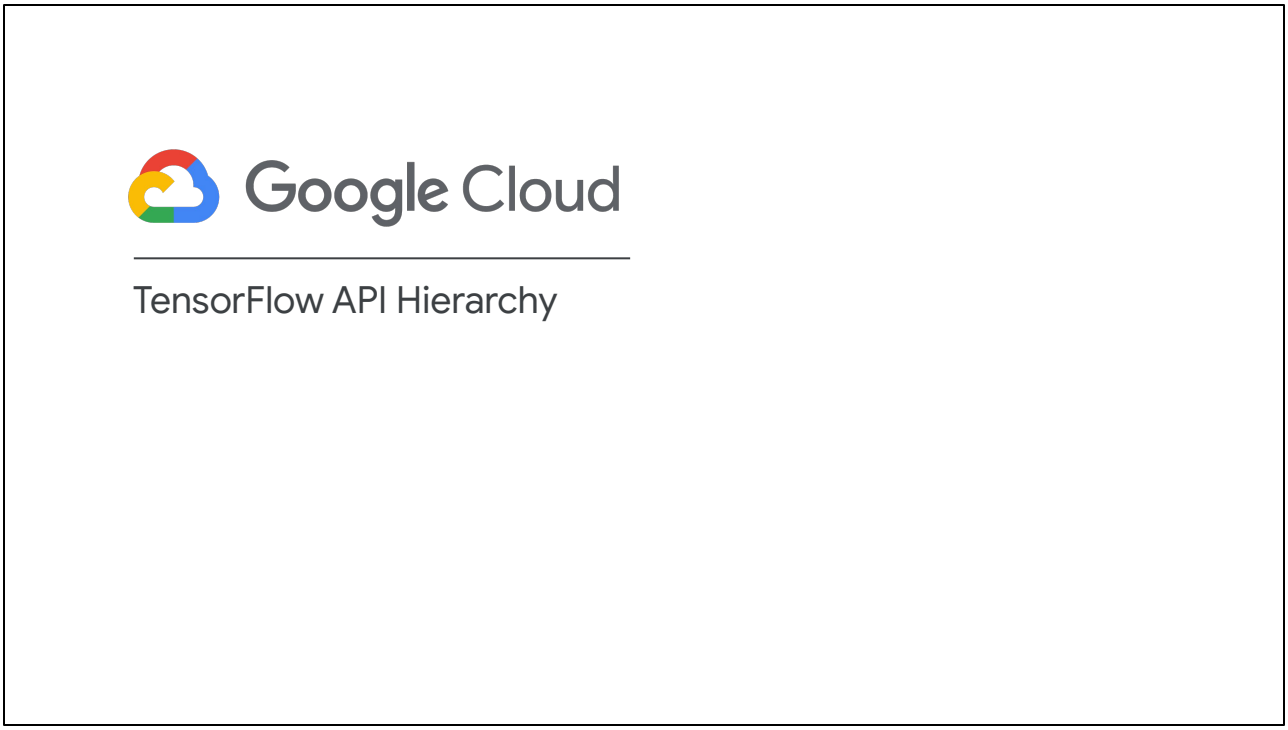
TensorFlow is popular among both deep learning researchers and machine learning engineers



#1 repository
for "machine learning"
category on GitHub

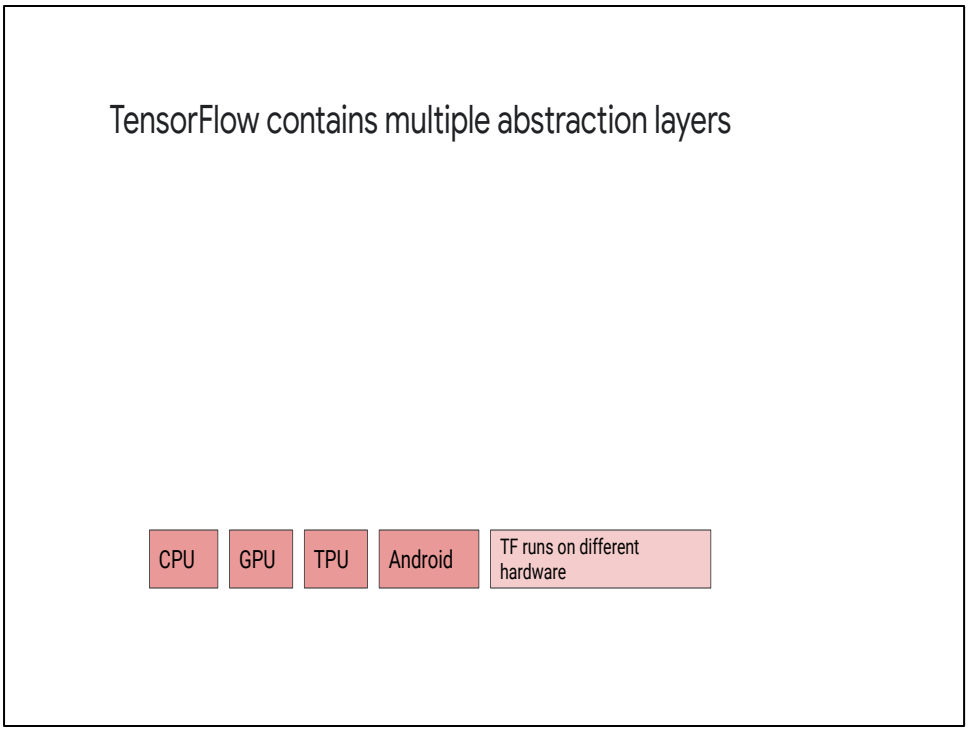
So, TensorFlow is this portable, powerful, production-ready software to do numerical computing. It's particularly popular for machine learning. The #1 repository for machine learning on GitHub. Why is it so popular?

It's popular among Deep Learning researchers because of the community around it, and the ability to extend it and do new, cool things.
It's popular among Machine Learning engineers because of the ability to productionize models, do things at scale.

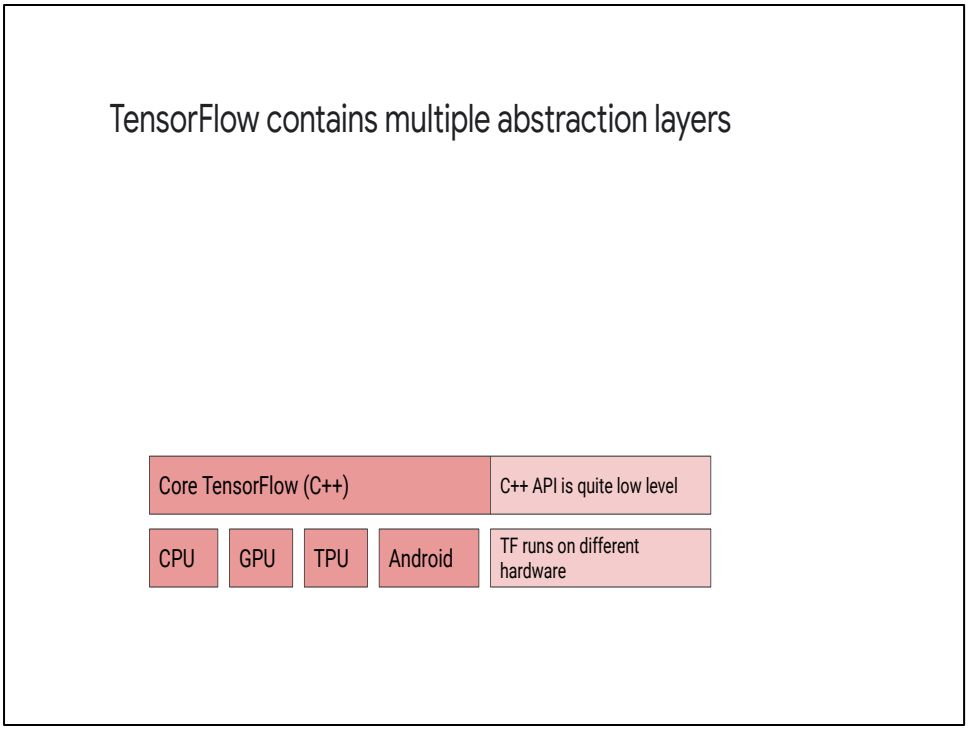


Let's take a look at the API hierarchy which will consist of a spectrum of low-level APIs for hardware all the way up to very abstract high-level APIs for super powerful tasks like creating a 128 layer neural network with just a few lines of code written with the Keras API.

Let’s start at the bottom.

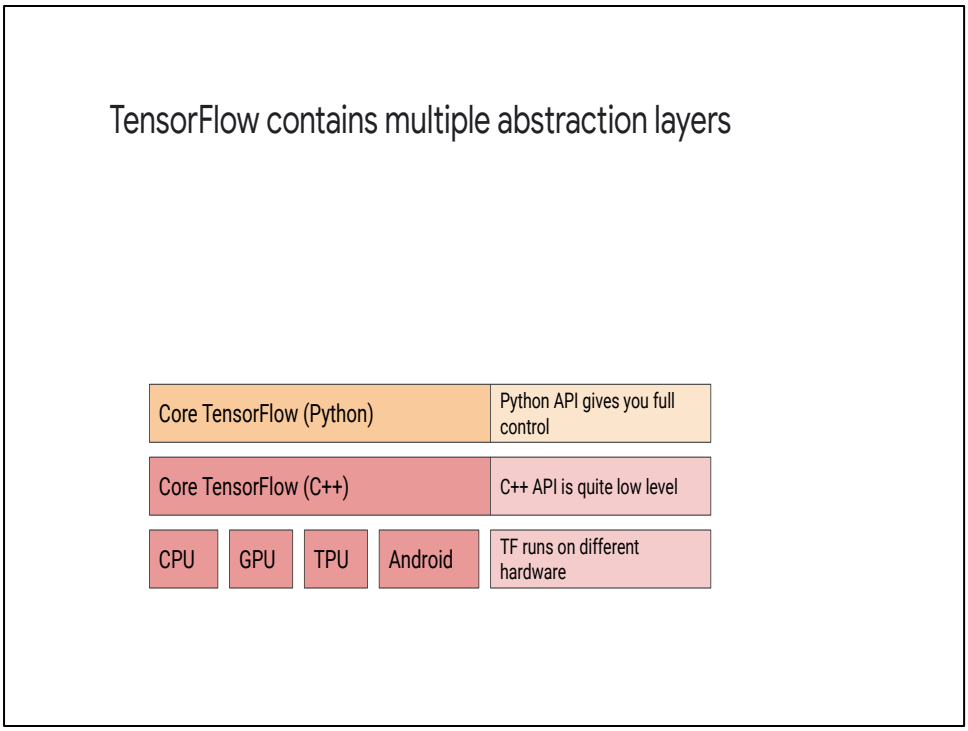


The lowest level of abstraction is the layer that is implemented to target the different hardware platforms. Unless your company makes hardware, it's unlikely that you'll do much at this level but it does exist.

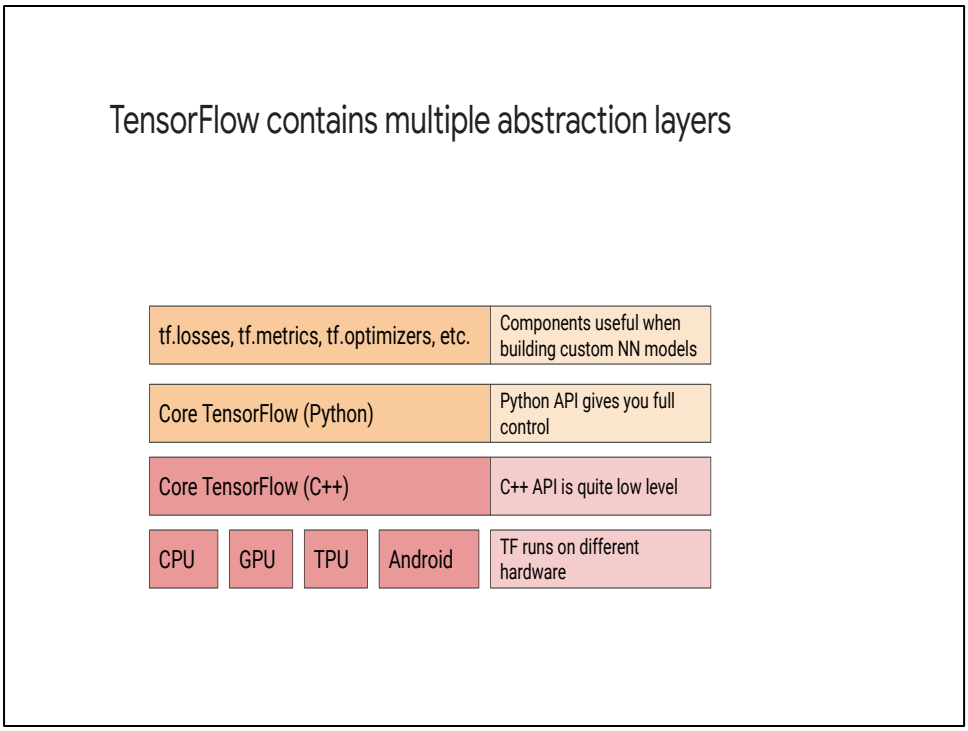


The next level is the TensorFlow C++ API. This is how you can write a custom TensorFlow operation. You would implement the function you want in C++ and register it as a TensorFlow operation. (You can find more details on the Tensorflow documentation on "extending an op" https://www.tensorflow.org/extend/adding_an_op).

TensorFlow will then give you a Python wrapper that you can use just like you would use an existing function. Assuming you're not an ML researcher, you don't have to do this. But if you ever need to implement your own custom op, you would do it in C++ and it's not too hard. TensorFlow is extensible that way.



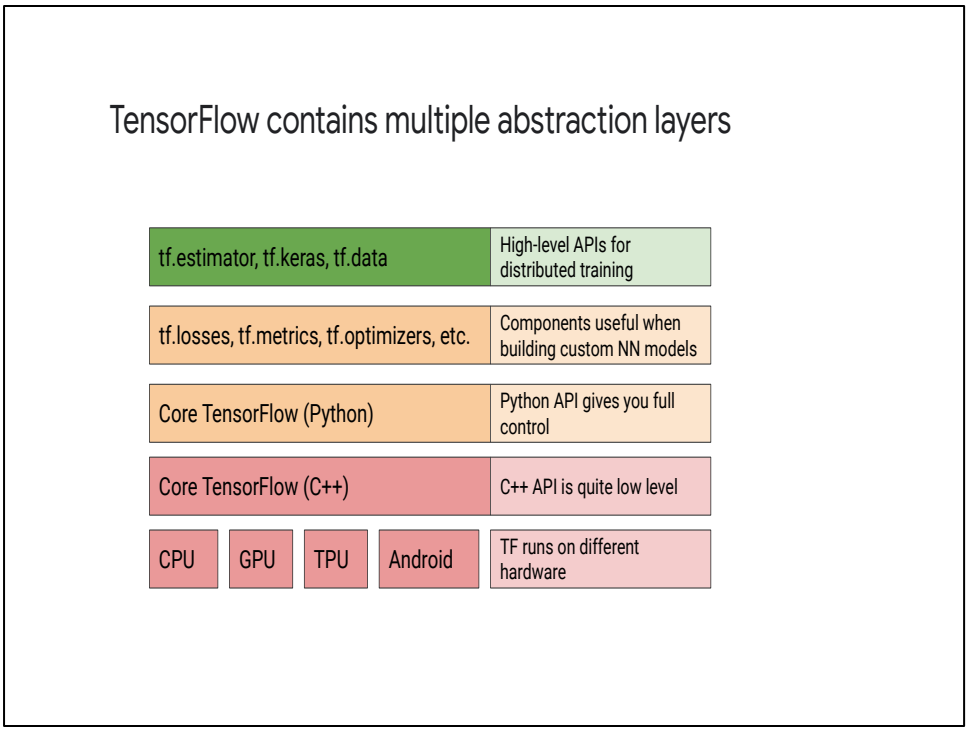
The Core Python API is what contains much of the numeric processing code. Add, subtract, divide, matrix multiply, etc. Creating variables, creating tensors, getting the shape or dimension of a tensor. All that core, basic, numeric processing stuff is in the Python API.



Then, there are sets of Python modules that have high-level representation of useful neural network components.

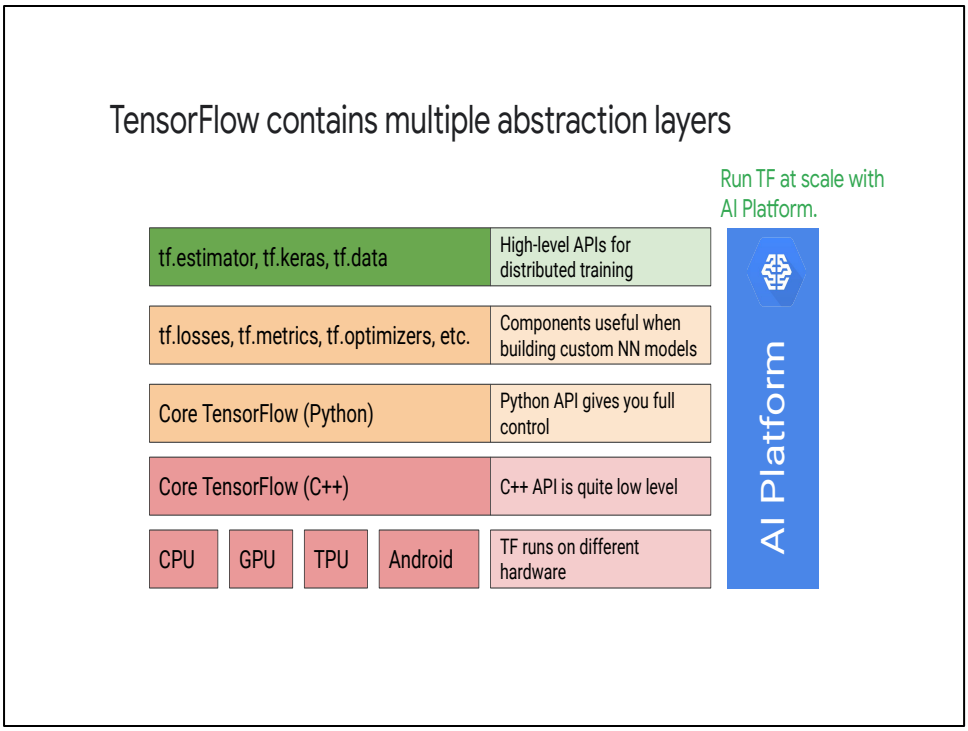
Let's say you are interested in creating a new layer of hidden neurons with a ReLU activation function. You can do that by using tf.layers.
If you want to compute the RMSE on data as it comes in, you can use tf.metrics.
To compute cross-entropy-with-logits, for example, which is a common loss measure in classification problems, you can use tf.losses.
These modules provide components that are useful when building 'custom' Neural Network models.

Why are ‘custom’ NN models emphasized? Because you often don’t need a custom NN model. Many times, you’re quite happy to go with a relatively standard way of training, evaluating and serving models. You don’t need to customize the way you train. You’re going to use one of the family of gradient-descent-based optimizers and you’re going to backpropagate the weights and do this iteratively. In that case, don’t write the low-level session loop. Just use an estimator or a high-level API such as Keras!

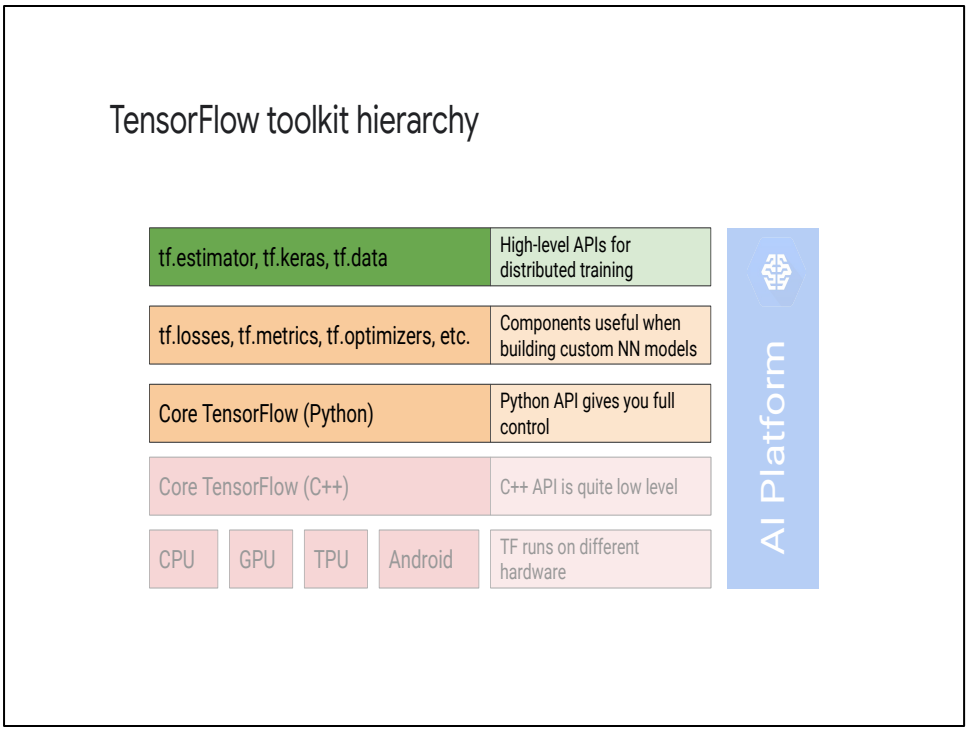


The high-level APIs allow you to easily do distributed training, data preprocessing and model definition, compilation and training. It knows how to evaluate, how to create a checkpoint, how to save a model, how to set it up for TensorFlow serving. It comes with everything done in a sensible way that fits most ML models in production.

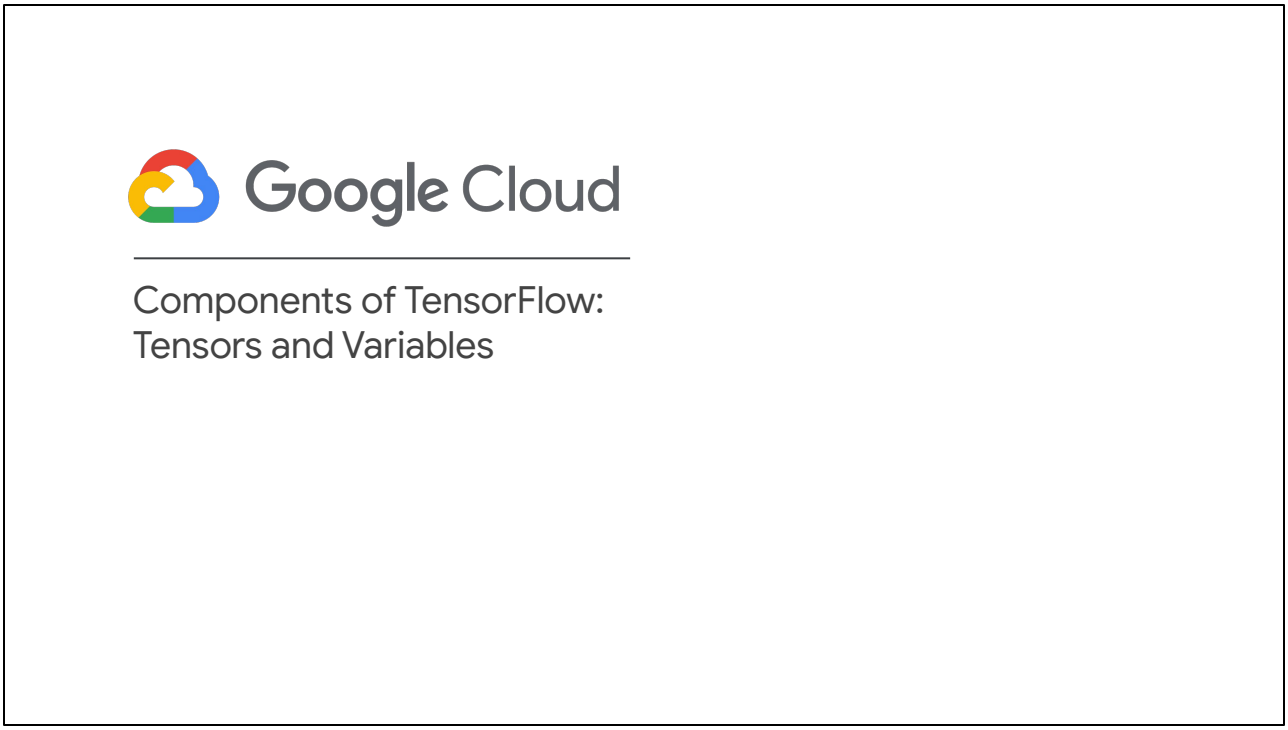
If you see example TensorFlow code out on the internet that does not use the Estimator API, ignore that code. Walk away. It's not worth it. You'll have to write a lot of code to do device placement, and memory management, and distribution. Let the high-level API do it for you.



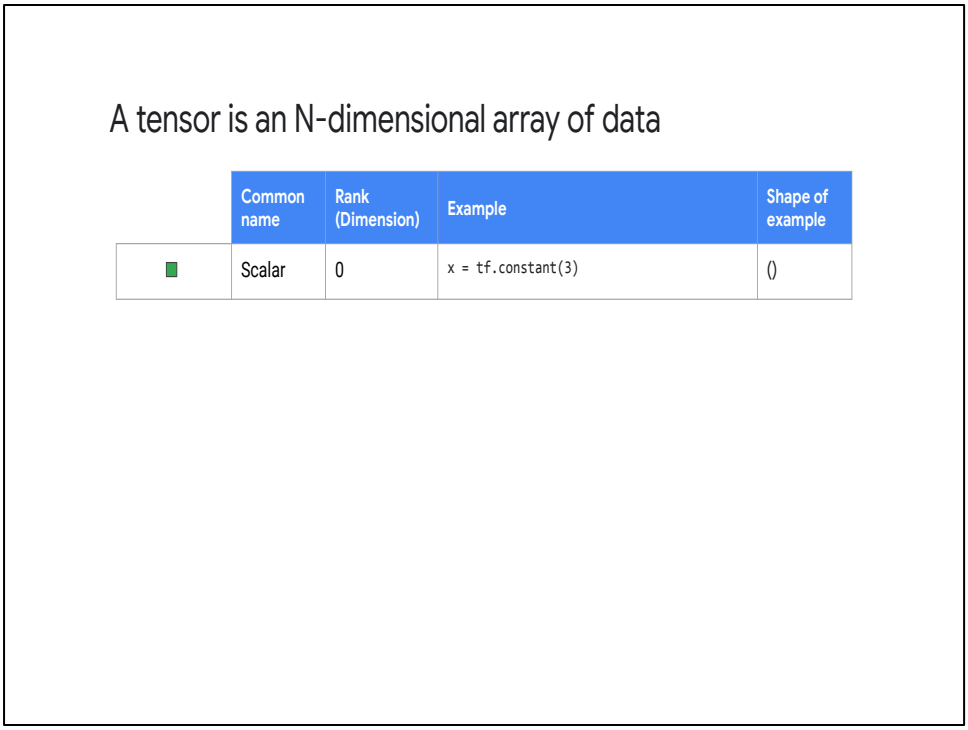
So, those are the TensorFlow levels of abstraction. Cloud AI Platform is orthogonal to this hierarchy -- meaning it cuts across all low-level and high-level APIs. Regardless of which abstraction level you are writing your TensorFlow code at, CAIP (Cloud AI Platform) gives you a managed service. It's fully hosted TensorFlow. So you can run TF on the cloud on a cluster of machines without having to install any software or manage any servers.



For the rest of this module we will be largely working with these top 3 level APIs here. Before we start writing API code and showing you the syntax for building machine learning models however -- we first need to really understand the pieces of data that we're working with. Much like in regular computer science classes where you start with variables and their definitions before moving on to advanced topics like classes, methods, and functions -- that is exactly how we're going to start learning TF components next.



Now let's go over tensors and variables in Tensorflow. It’s time to see some code!
How can we bring life to each dimension of a tensor that we learned about earlier?



A tensor is an N-dimensional array of data. When you create a tensor, you will specify its SHAPE. Occasionally, you'll not specify the shape completely. For example, the first element of the shape could be variable, but that special case will be ignored for now. Understanding the SHAPE of your data -- or often times the shape it SHOULD be -- is the first essential part of your machine learning flow.

Here, you're going to create a tf.constant(3). This is 0-rank tensor. Just a number. A scalar. The shape when you look at the Tensor debug output will be simply open-parenthesis close-parenthesis. It's zero-rank. To better understand why there isn't a number in those parentheses let's upgrade to the next level.

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,)

If you passed in a bracketed list like 3, 5, 7 to tf.constant instead -- you would now be the proud owner of a one-dimensional tensor (otherwise known as a vector)

Now, you have a one-dimensional tensor. A vector.

It's grow horizontally (think like on the X-Axis) by three units. Nothing on the Y Axis yet since we're still in 1 dimension here. That's why the shape is (3, nothing).

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)

Level up!

Now we have a matrix of numbers or a two dimensional array. Take a look at the shape (2,3) that means we have two rows and three columns of data. The first row being that original vector of [3,5,7] which also has three elements in length (that's where the three columns of data comes from).

You can think of a matrix as essentially a stack of 1 D tensors. The first tensor is the vector [3,5,7], and the second 1D tensor that is being stacked is the vector [4,6,8].

Okay so we've got height and width. Let's get more complex!

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)

What does 3D look like?

It's a 2D tensor with another 2D tensor on top of it. Here you can see we're stacking the 3,5,7 matrix on top of the 1,2,3 matrix. We started with two 2x3 marices so our resulting shape of the 3D tensor is now 2 comma 2 comma 3.

A tensor is an N-dimensional array of data

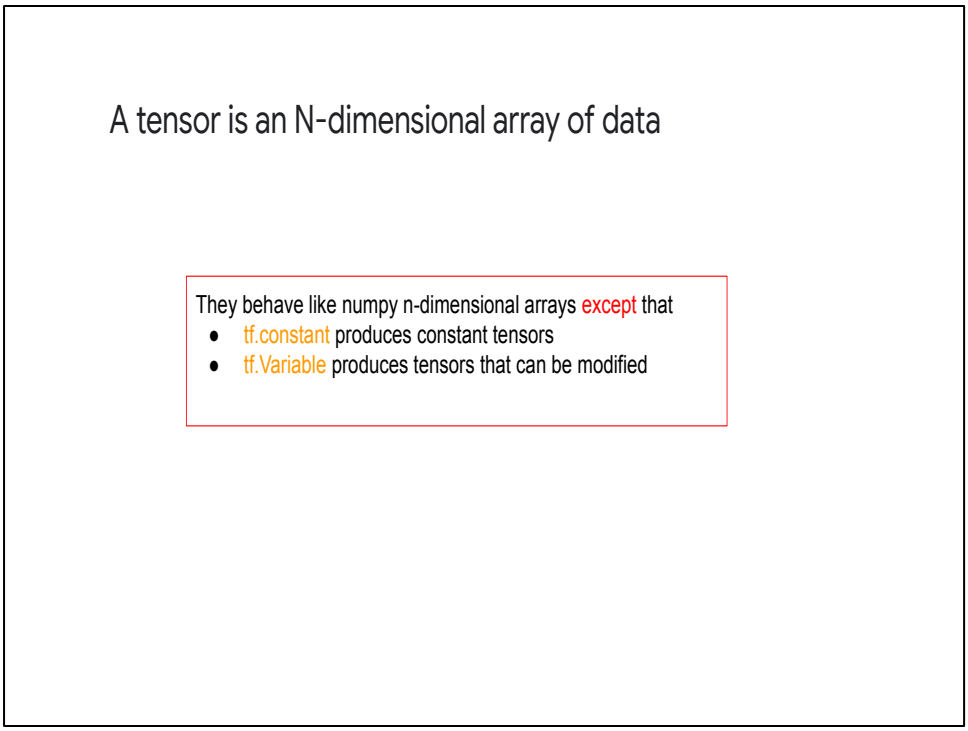
	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

	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)

Of course, you can do the stacking in code itself instead of counting parentheses.

Take the example here. Our x1 variable is a tf constant constructed from a simple list [2,3,4]. That makes it a vector of length 3.
X2 is constructed by stacking x1 on top of x1. So, that makes it a 2x3 matrix.
X3 is constructed by stacking 4 x2s on top of each other. Since each x2 was a 2x3 matrix, this makes X3 a 3D tensor of shape 4x2x3.
X4 is constructed by stacking x3 on top of x3. That makes it two 4x2x3 tensors, or the final result which is a 4D tensor of shape 2 comma 4 comma 2 comma 3.



If you've worked with arrays of data before like with Numpy they're similar expect for these two points.

tf.constant will produce tensors with constant values and tf.variable produces tensors with variable values (or ones that can be modified). This will prove super useful later when we need to adjust the model weights during the training phase of our ML project. Those weights can simply be a modifiable tensor array.

Let's take a look at the syntax for each as you will become a ninja in combining, slicing, and reshaping tensors as you see fit.

tf.constant produces constant tensors...

```
import tensorflow as tf

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

Here's a constant tensor produced by --- well tf.constant of course!

Remember that 3,5,7 is our 1D vector and we've just stacked it here to be a 2D matrix.

Pop quiz -- what is the shape of X? Hint: How many rows of data (or stacks) and then how many columns do you see?

Answer: This shape is 2x3 or 2 rows and 3 columns. When you're coding you can also invoke tf.shape() which is quite handy when debugging.

Okay -- much like you can stack tensors to get to higher dimensions you can also SLICE them down too.

Tensors can be sliced

```
import tensorflow as tf

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

y = x[:, 1]

[5, 6]
```

Now, take a look at this code for y. It's 'slicing' x. Is it slicing rows, columns, or both?

The syntax is let Y be the result of taking X and take ALL rows (that's the COLON) and just the first column. Keeping in mind that Python is zero indexed when it comes to arrays what would the result be? Remember we're going down from 3D to 2D here so your answer should only be a single bracketed list of numbers.

Answer? [5, 6]. Again -- take all rows and only the first indexed column. 3,4 is the 0 index and 5,6 is the first index which is what gets printed out. Make sense? Don't worry you'll get plenty of practice.

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]
]
```

So we've seen stacking and slicing -- next let's talk about reshaping with tf.reshape.

Let's use the same 2D tensor or matrix of values that is X. What's the shape again? Think rows then columns. If you said 2x3 you're right.

NOW what if I reshaped X as [3,2] or three rows and two columns, what would happen?

Essentially Python will read the input tensor row-by-row and put numbers into the output tensor. It will pick the first two values and put it into the first row. So, you get 3 and 5.
The next two values, 7 and 4, go into the second row.
And the last two values, 6 and 8, go into the third row.
That's what reshaping does.

Well that's it for constants -- not too bad right? Next up are variable tensors.

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')
```

The Variable() constructor requires an initial value for the variable, which can be a tensor of any type and shape.
This initial value defines the type and shape of the variable. After construction, the type and shape of the variable are fixed.

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')

# x <- 48.5
x.assign(45.8)

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

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

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

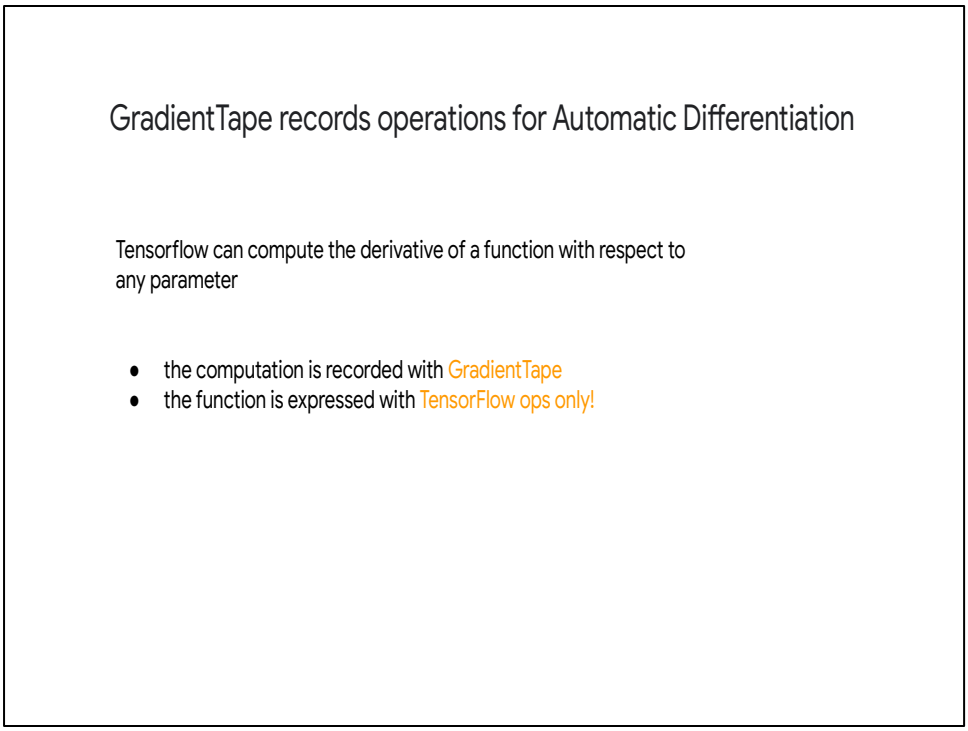
The value can be changed using one of the assign methods (assign, assign_add, assign_sub).

As we mentioned before, tf.variables are generally used for values that are modified during training (such as model weights).

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

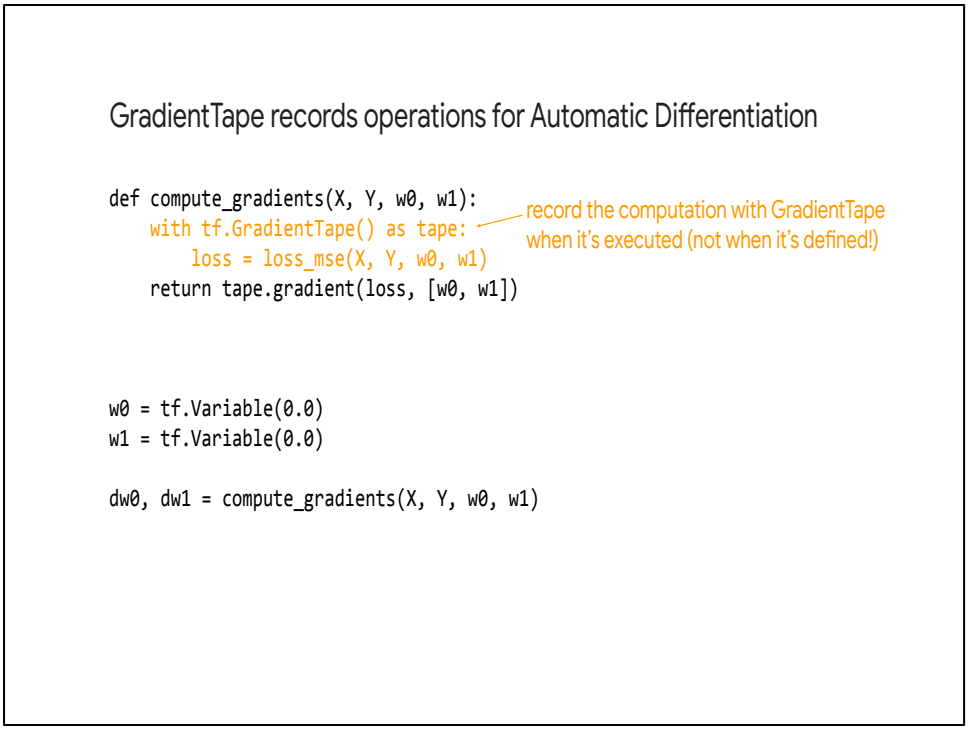
```
import tensorflow as tf

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

Just like any tensor, variables created with Variable() can be used as inputs to operations.
Additionally, all the operators overloaded for the Tensor class are carried over to variables.

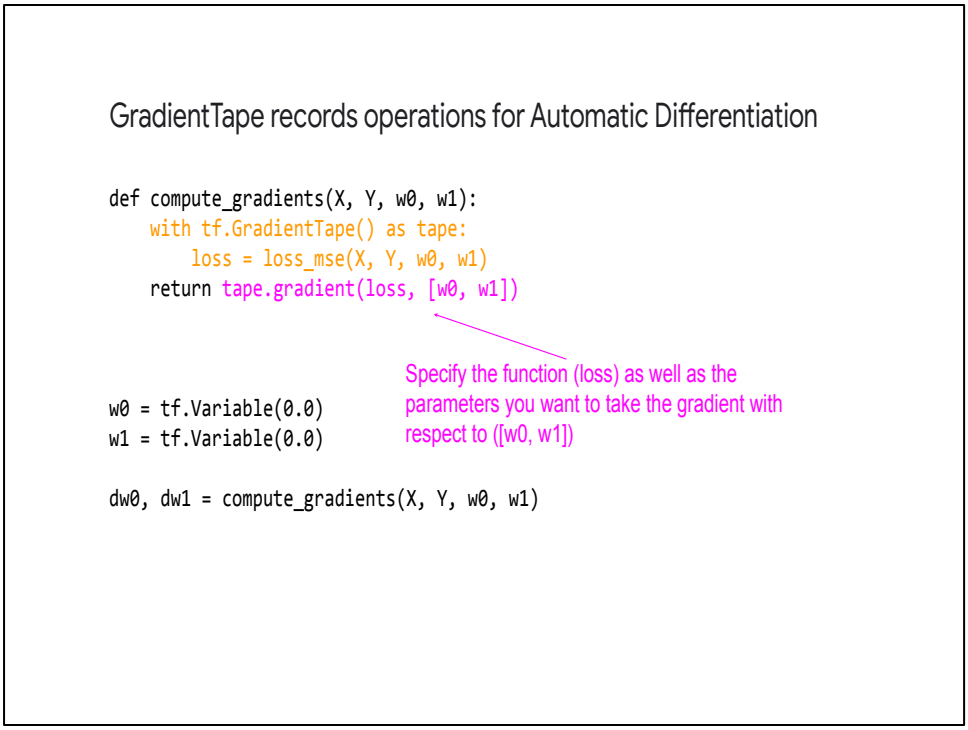


Tensorflow has the ability to calculate the partial derivative of any function with respect to any variable.
We know that, during training, weights are updated by using the partial derivative of the loss with respect to each individual weight.
To differentiate automatically, TensorFlow needs to remember what operations happen in what order during the forward pass. Then, during the backward pass, TensorFlow traverses this list of operations in reverse order to compute gradients.
GradientTape is a context manager in which these partial differentiations are calculated.

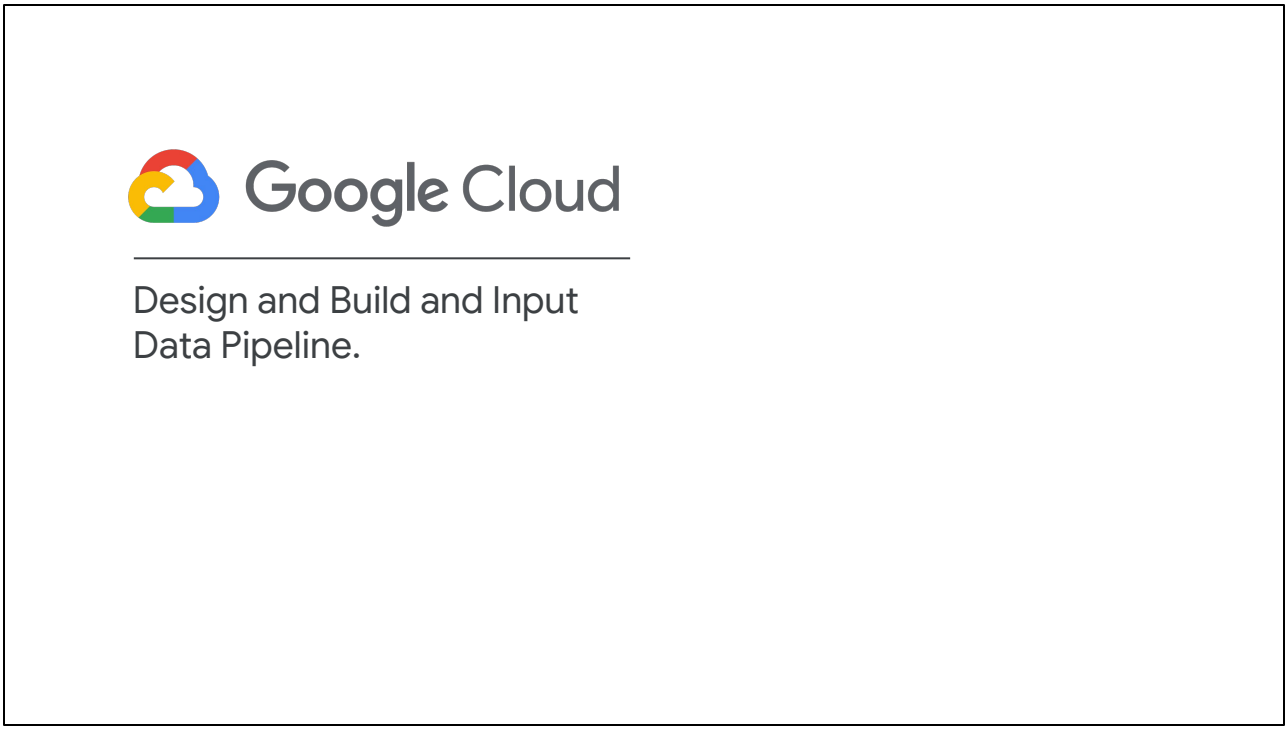
The functions have to be expressed within Tensorflow operations only. But, since most basic operations like addition, multiplication and subtraction, are overloaded by Tensorflow ops, these happen seamlessly.



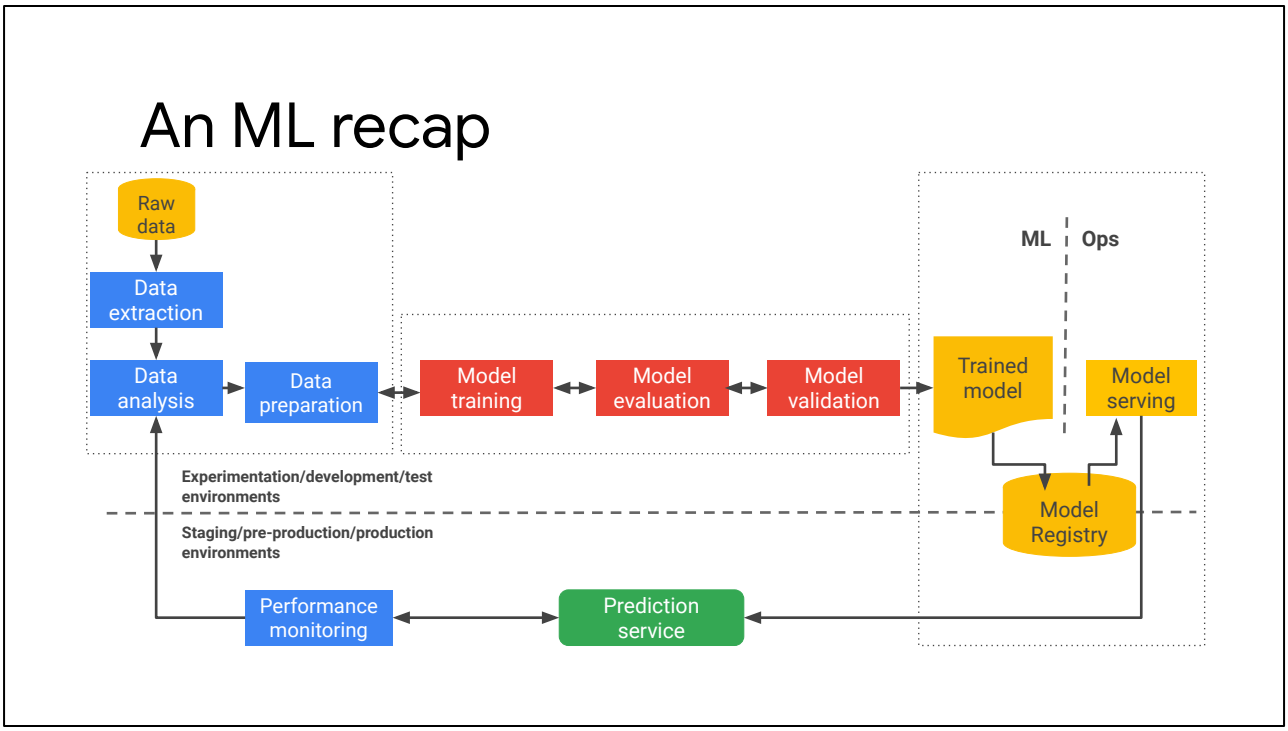
Let's say that we want to compute a loss gradient. TensorFlow "records" all operations executed inside the context of a tf.GradientTape onto a "tape". Then, it uses that tape and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation using reverse mode differentiation.



There are cases where you may want to control exactly how gradients are calculated rather than using the default. These cases can be when the default calculations are numerically unstable or you wish to cache an expensive computation from the forward pass, among others. For such scenarios, you can use Custom Gradient functions to write a new operation or to modify the calculation of the differentiation.



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



Let's start with a recap.

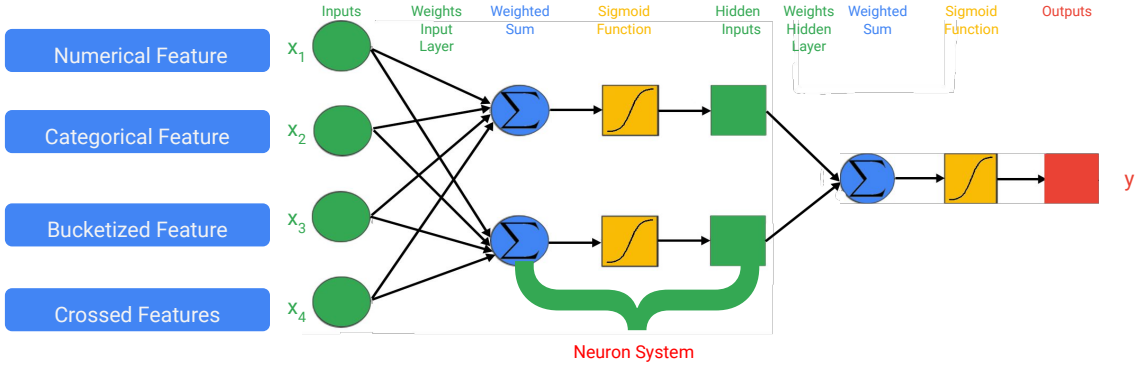
In any ML project, after you define the business use case and establish the success criteria, the process of delivering an ML model to production involves the following steps. These steps can be completed manually or can be completed by an automated pipeline:

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

An ML recap

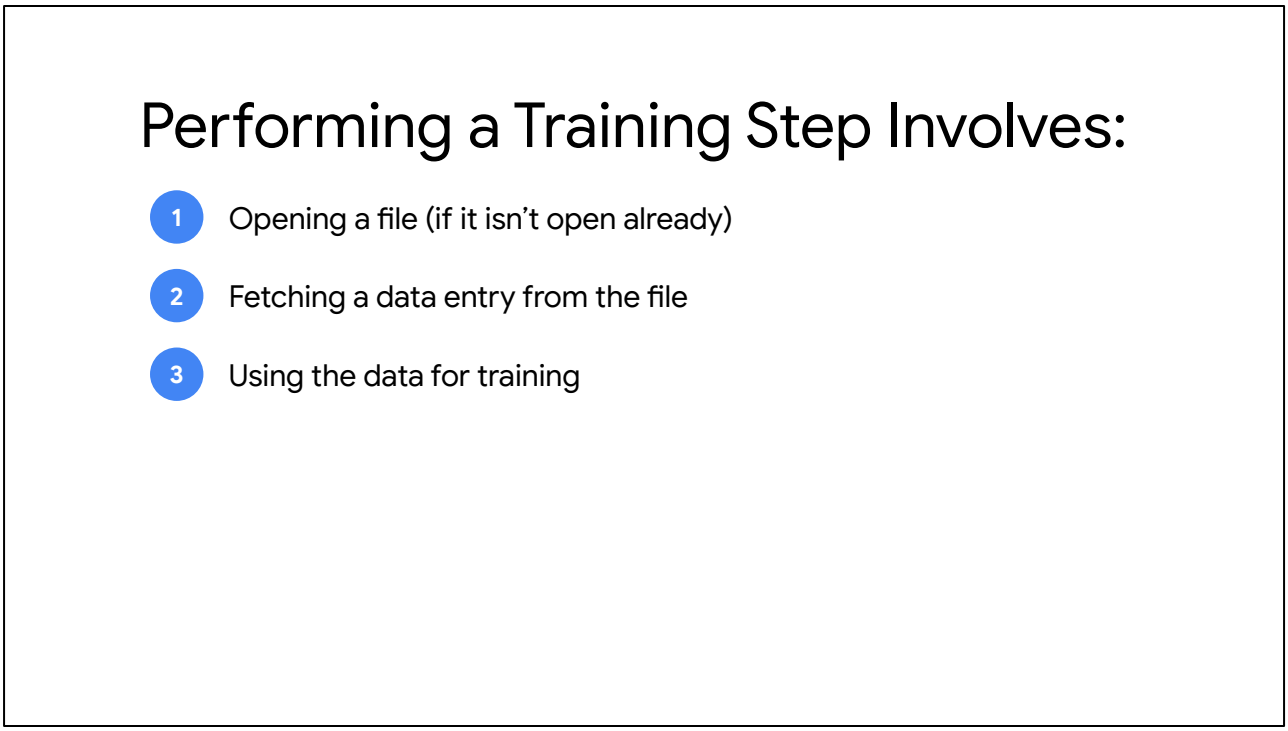
We saw that there are two phases in machine learning: a training phase and an inference phase. We learned that an ML problem can be thought of as being all about data.

An ML recap



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

Features, the columns of your dataframe, are key in assisting machine learning models to learn. Better features result in faster training and more accurate predictions. As the diagram shows, feature columns are input into the model—not as raw data, but as feature columns.



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

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

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

tf.data API

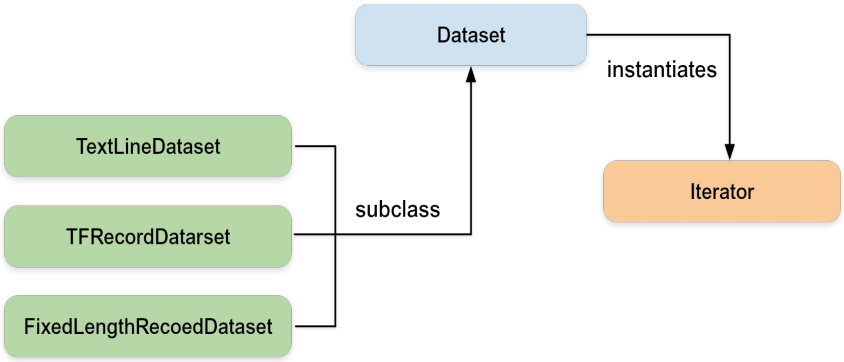
1. Build complex input pipelines from simple, reusable pieces.
2. Build pipelines for multiple data types.
3. Handle large amounts of data; perform complex transformations.

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

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

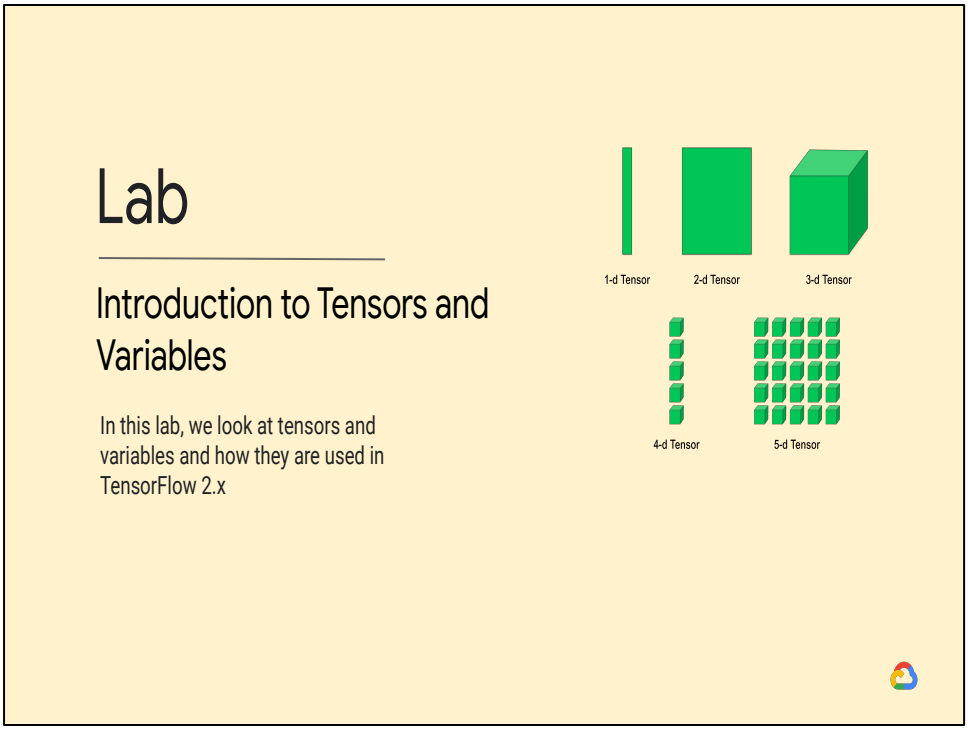
We'll use the tf.data API quite a bit in this lesson!

Multiple ways to feed TensorFlow models with data



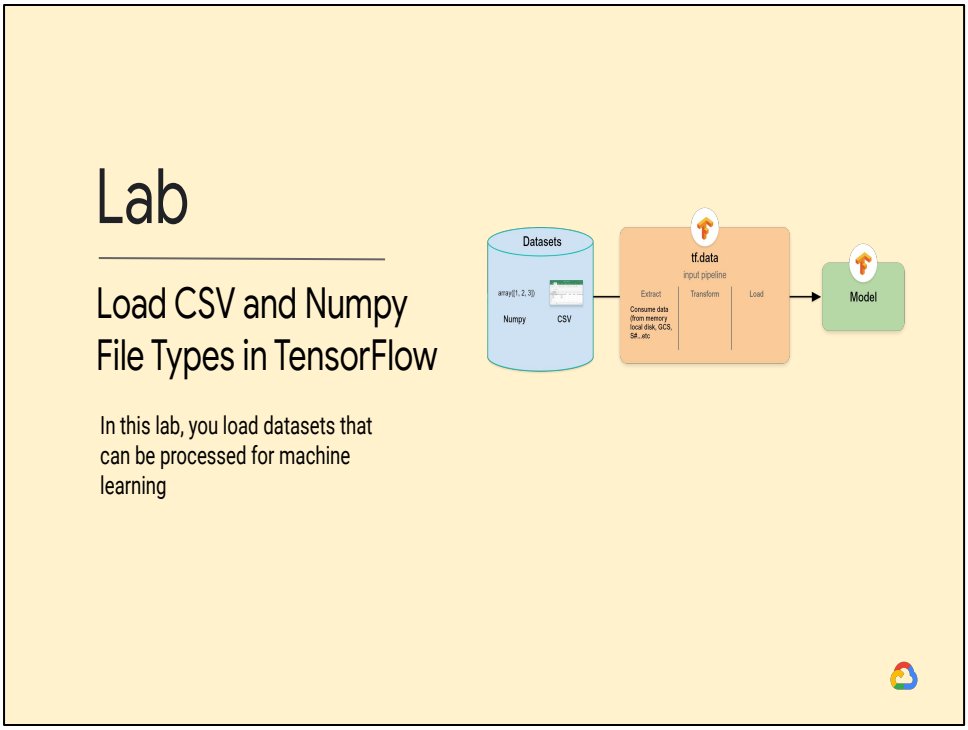
There are multiple ways to feed TensorFlow models with data, and you will see those in the next videos.

So, let's get started!



In this lab, you will write low-level TensorFlow code.

Link to lab: [ML on GCP C3] Introduction to Tensors and Variables (qwiklabs) CBL248 link to github repo here.



In this lab you will learn how to load data sets from a source file, from a folder or an application so that it can be processed for machine learning. To achieve that, you will learn how to load CSV data from a file into a tf.data.Dataset object...And you will also see an example of loading data from NumPy arrays into a tf.data.Dataset.

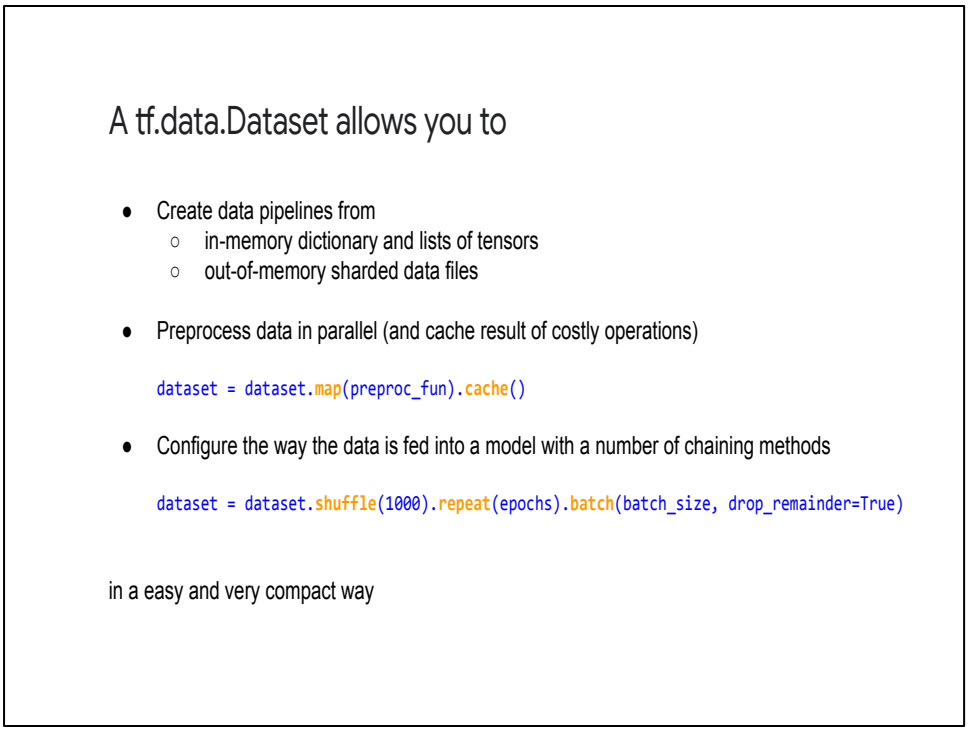
Link to lab: [ML on GCP C3] Load CSV, Numpy, and Text data in TensorFlow (qwiklabs) github repo link here. CBL250

Training on Large Datasets with tf.data

Models which are deployed in production, specially, require lots and lots of data. This is data that likely won't fit in memory and can possibly be spread across multiple files or may be coming from an input pipeline.

The tf.data API enables you to build complex input pipelines from simple, reusable pieces. For example, the pipeline for structured data might require normalization, feature crosses or bucketization. An image model might aggregate data from files in a distributed file system, apply random perturbations to each image, and merge randomly selected images into a batch for training. The pipeline for a text model might involve extracting symbols from raw text data, converting them to embedding identifiers with a lookup table, and batching together sequences of different lengths.

The tf.data API makes it possible to handle large amounts of data, read from different data formats, and perform complex transformations.

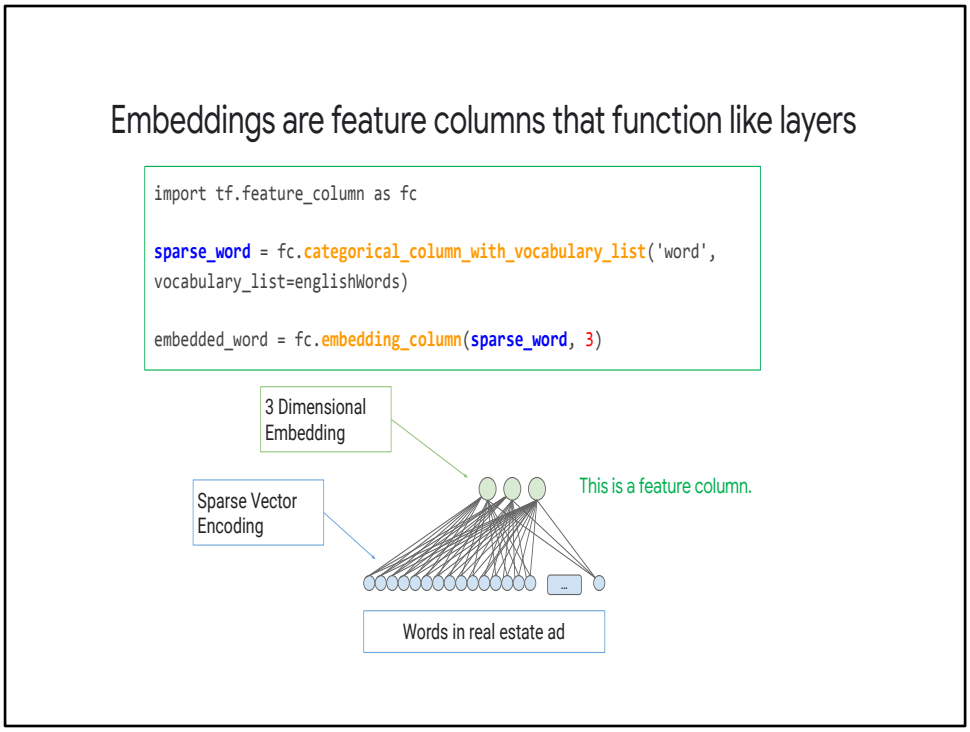


It's time to look at some specifics. The tf.data API introduces a tf.data.Dataset abstraction that represents a sequence of elements, in which each element consists of one or more components. For example, in an image pipeline, an element might be a single training example, with a pair of tensor components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

There are two distinct ways to create a dataset:

A data source constructs a Dataset from data stored in memory or in one or more files.

A data transformation constructs a dataset from one or more tf.data.Dataset objects.

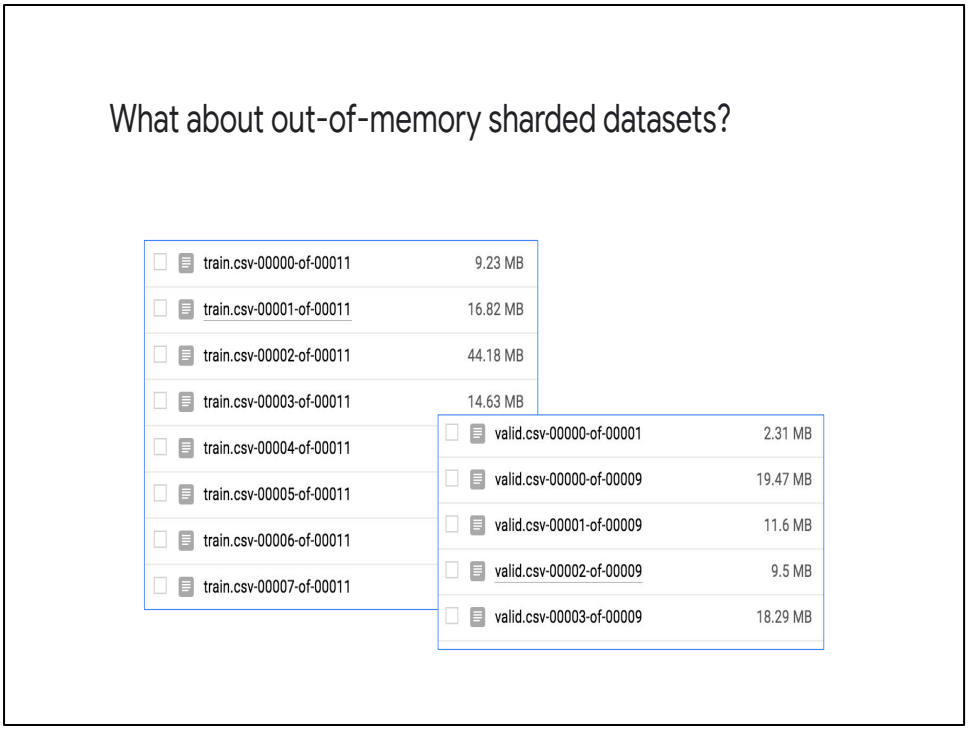


Imagine that you are creating an embedding to represent the key word in a real-estate ad. Let's ignore, for now, how you choose this important word.
Now, words in an ad are natural language and so the potential dictionary is vast. We call this a dataset with high dimensionality. In this case, it could be list of all english words. Tens of thousands of words even if we ignore rare words and scientific jargon.

So, obviously, even though the first layer here takes the word in the real-estate ad and one-hot encodes it, the representation of this in memory will be as a sparse vector. That way, TensorFlow can be efficient in its use of memory.

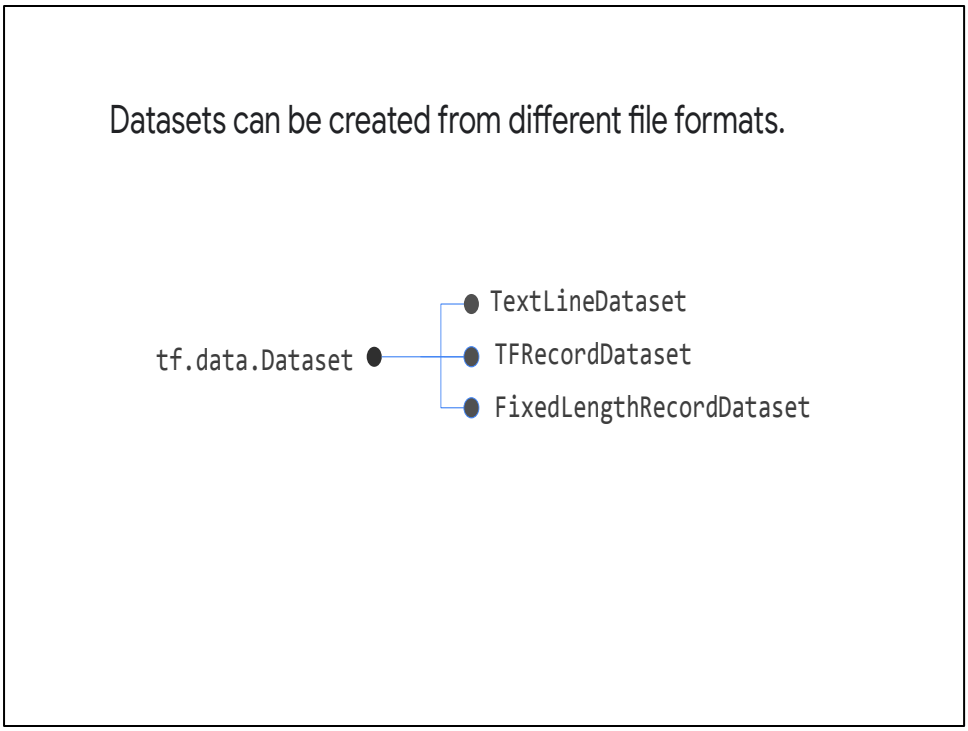
Once we have the one-hot encoded representation, we pass it through a 3-node layer as you see here. This is our **embedding**, and because we use 3 nodes in that layer, it is a 3-dimensional embedding. Note that even though sparse_word and embedded_word are really feature columns, I am showing them here as neural network layers.
That is because, mathematically, they *are* just like neural network layers.

Mathematically, an embedding in this case isn't really different from any other hidden layer in a network. You can view it as a handy adapter that allows the network to incorporate sparse or categorical data well.



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

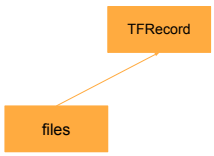
Large datasets tend to be sharded into multiple files which can be loaded progressively. Remember that you train on mini-batches of data. You do not need to have the entire dataset in memory. One mini-batch is all you need for one training step.



The Dataset API will help you create input functions for your model that will load data *progressively*. There are specialized Dataset classes that can read data from text files like CSVs, Tensorflow records or fixed length record files.
Datasets can be created from different file formats:
Use TextLineDataset to instantiate a Dataset object which is comprised of lines from one or more text files.
TFRecordDataset comprises records from one or more TFRecord files.
And FixedLengthRecordDataset is a dataset object from fixed-length records from one or more binary files.

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

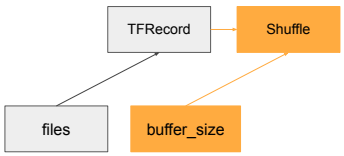
```
dataset = tf.data.TFRecordDataset(files)
```



Let's walk through an example of TFRecordDataset.

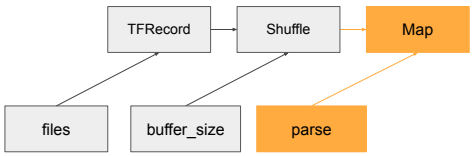
At the beginning, the TFRecord op is created and executed, producing a variant tensor representing a dataset which is stored in the corresponding Python object.

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

dataset = dataset.shuffle(buffer_size=X)
```



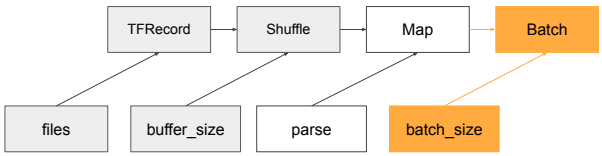
Next, the Shuffle op is executed, using the output of the TFRecord op as its input, connecting the two stages of the input pipeline.

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

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))
```



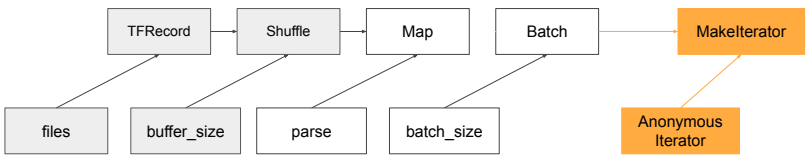
Next, the user-defined function is traced and passed as attribute to the Map operation, along with the Shuffle dataset variant input.

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

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))

dataset = dataset.batch(batch_size=Y)
```



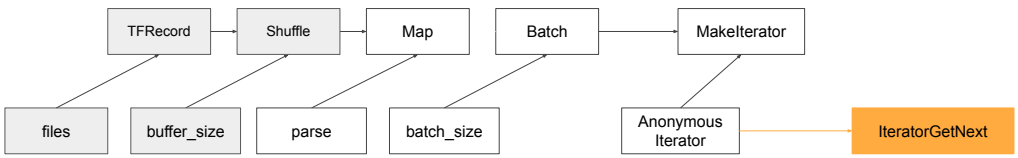
Finally, the Batch op is created and executed, creating the final stage of the input pipeline.

```
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

  ...
```



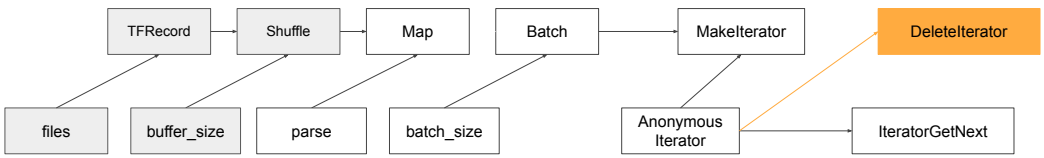
When the for loop mechanism is used for enumerating the elements of dataset, the `iter` method is invoked on the dataset, which triggers creation and execution of two ops. First an anonymous iterator op is created and executed, which results in creation of an iterator resource. Subsequently, this resource along with the Batch dataset variant is passed into the MakeIterator op, initializing the state of the iterator resource with the dataset.

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

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))

dataset = dataset.batch(batch_size=Y)


for element in dataset:

  ...  #  next() is called
```



When the `next` method is called, it triggers creation and execution of the IteratorGetNext op, passing in the iterator resource as the input. Note that the iterator op is created only once but executed as many times as there are elements in the input pipeline.

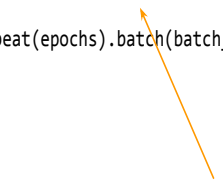
```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

dataset = dataset.map(lambda record: parse(record))

dataset = dataset.batch(batch_size=Y)


for element in dataset:

  ... #  iterator goes out of scope
```



Finally, when the Python iterator object goes out of scope, the DeleteIterator op is executed to make sure the iterator resource is properly disposed of. To state the obvious, properly disposing of the iterator resource is essential as it is not uncommon for the iterator resource to allocate 100MBs to 1GBs of memory because of internal buffering.

Creating a dataset from in-memory tensors

```
def create_dataset(X, Y, epochs, batch_size):
    dataset = tf.data.Dataset.from_tensor_slices((X, Y))
    dataset = dataset.repeat(epochs).batch(batch_size, drop_remainder=True)
    return dataset
```



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

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

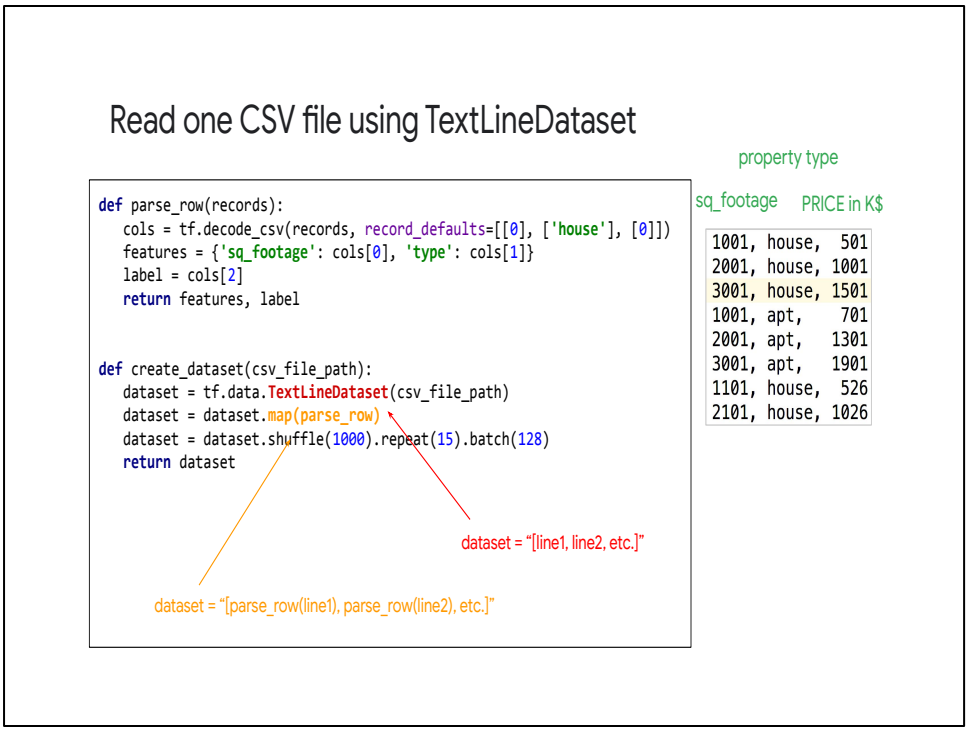
When data used to train a model sits in memory, we can create an input pipeline by constructing a Dataset using tf.data.Dataset.from_tensors() or tf.data.Dataset.from_tensor_slices().

Use from_tensors() or from_tensor_slices()

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

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

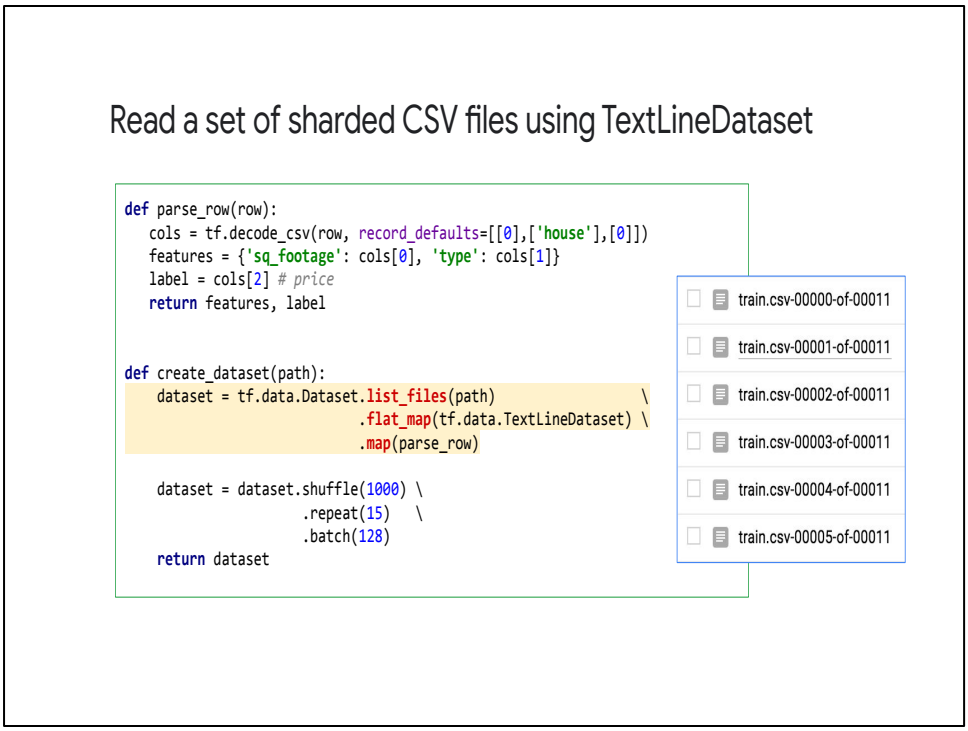
from_tensors() combines the input and returns a dataset with a single element while from_tensor_slices() creates a dataset with a separate element for each row of the input tensor



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

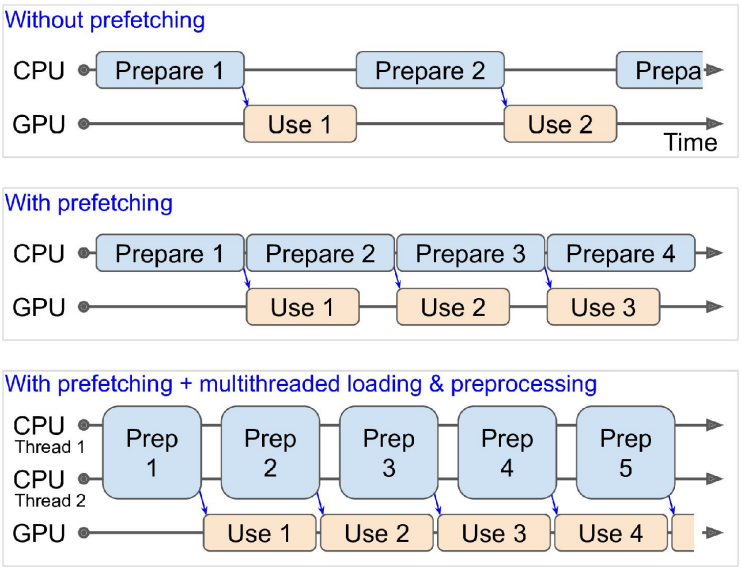
The TextLineDataset instantiation expects a file name and it has optional arguments such as, for example, the type of compression of the files or the number of parallel reads. The map function is responsible for parsing each row of the CSV file. It returns a dictionary from the file content. Once that is done, shuffling, batching and prefetching are steps that can be applied to the dataset to allow for the data to be fed into the training loop iteratively.

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



Finally, we have to address our initial concern: loading large datasets from a set of sharded files. An extra line of code will do.
We first scan the disk and load a dataset of file names using the Dataset.list_files functions. It supports a glob-like syntax with stars to match filenames with a common pattern. Then we use TextLineDataset to load the files and turn each file name into a dataset of text lines. We flat_map all of them together into a single dataset. And then, for each line of text we use map to apply the CSV parsing algorithm and obtain a dataset of features and labels.

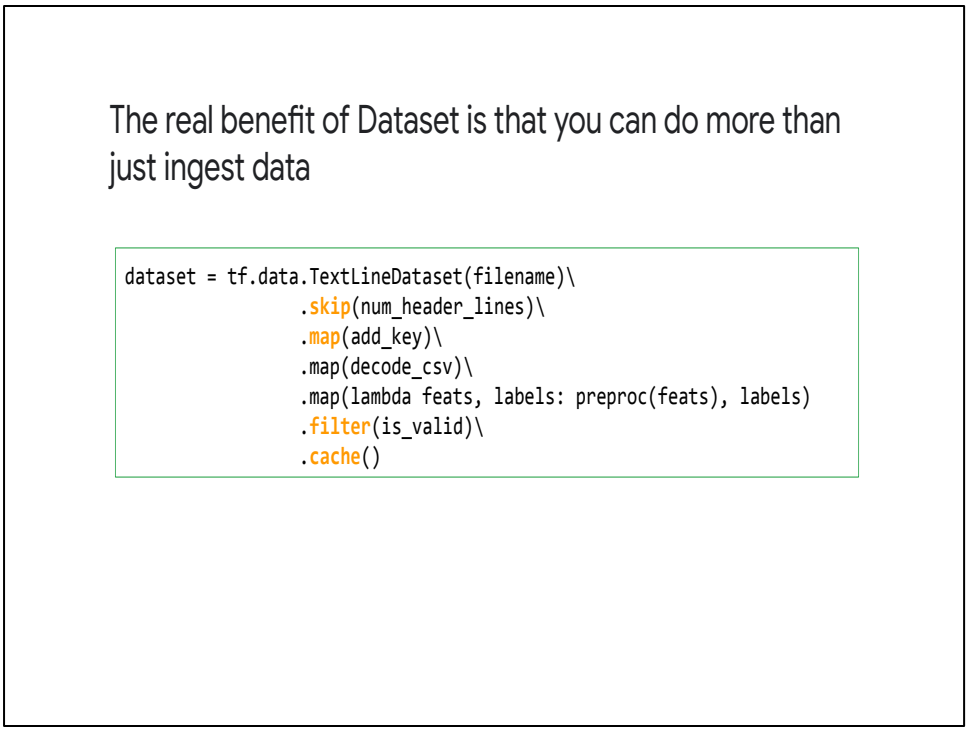
Why are there two mapping functions: map and flat_map ? One of them is simply for one to one transformations and the other one for one-to-many transformations. Parsing a line of text is a one to one transformation so we apply it with map. When loading a file with TextLineDataset, one file name becomes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



Dataset allows for data to be prefetched. Let's say we have a cluster with a GPU on it. Without prefetching, the CPU will be preparing the first batch while the GPU is waiting. Once that is done, the GPU can then run the computations on that batch. Once that is finished, the CPU will start preparing the next batch and so forth. We can easily see that this is not very efficient.

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

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



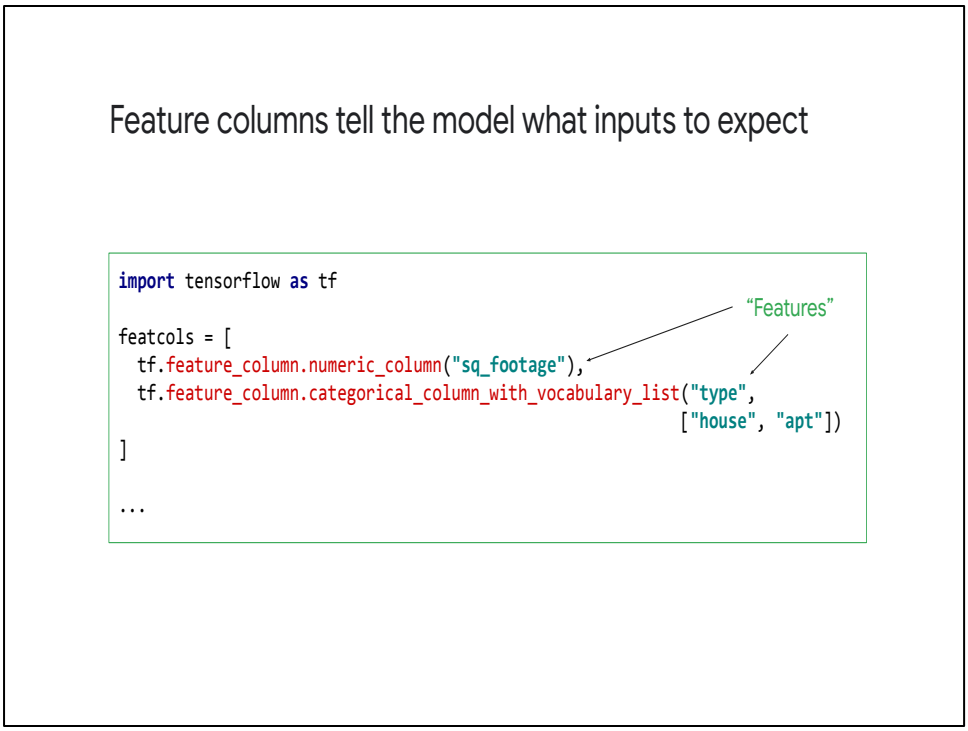
Now you know how to use Datasets to generate proper input functions for your models and get them training on large out of memory datasets. But Datasets also offer a rich API for working on and transforming your data. Take advantage of it!

As we think about modeling a real problem with machine learning, we first need to think about what input signals we can use to train the model.
In this next section, let's use a common example. How about real estate? Can you predict the price of a property? As you think about that problem, you must first choose your "features", that is, the data you will be basing your predictions on. Why not try and build a model that predicts the price based on the area of a house or apartment. Your features will be 1) the square footage, 2) the category: "house" or "apartment".

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

Feature columns bridge the gap between columns in a CSV file to the features used to train a model

Remember, a feature column describes how the model should use raw input data from the features dictionary. In other words, feature column provides methods for the input data to be properly transformed being sending it to a model for training.



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

Other variations of it are:

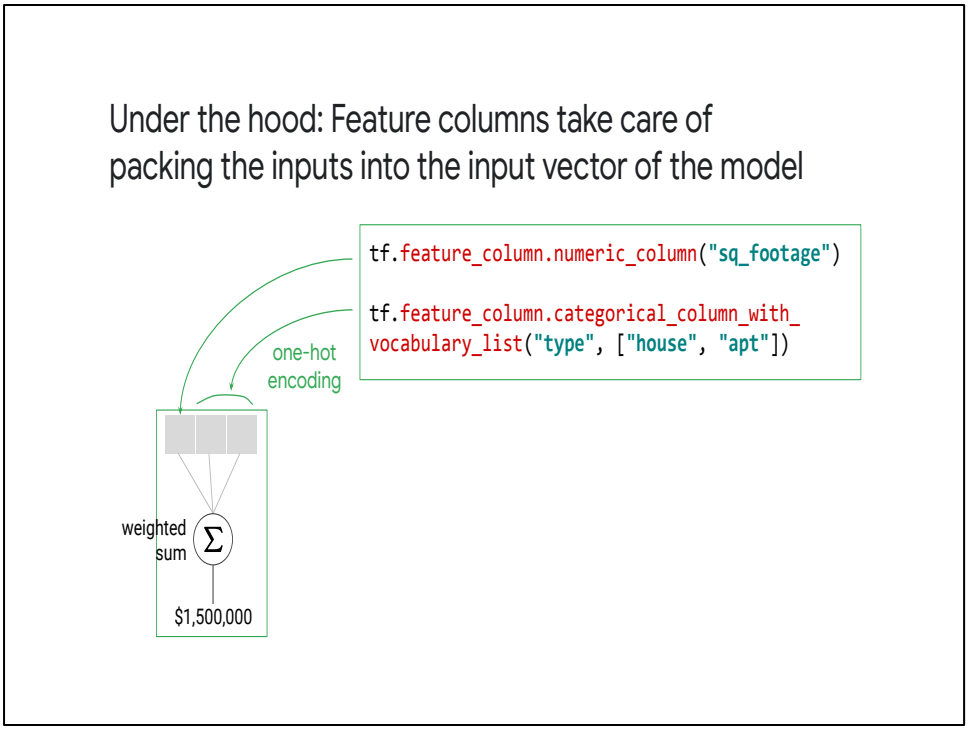
- categorical_column_with_vocabulary_file (used when the inputs are in string or integer format, and there is a vocabulary file that maps each value to an integer ID.);
- categorical_column_with_identity (used when the inputs are integers in the range [0, num_buckets), and you want to use the input value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

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

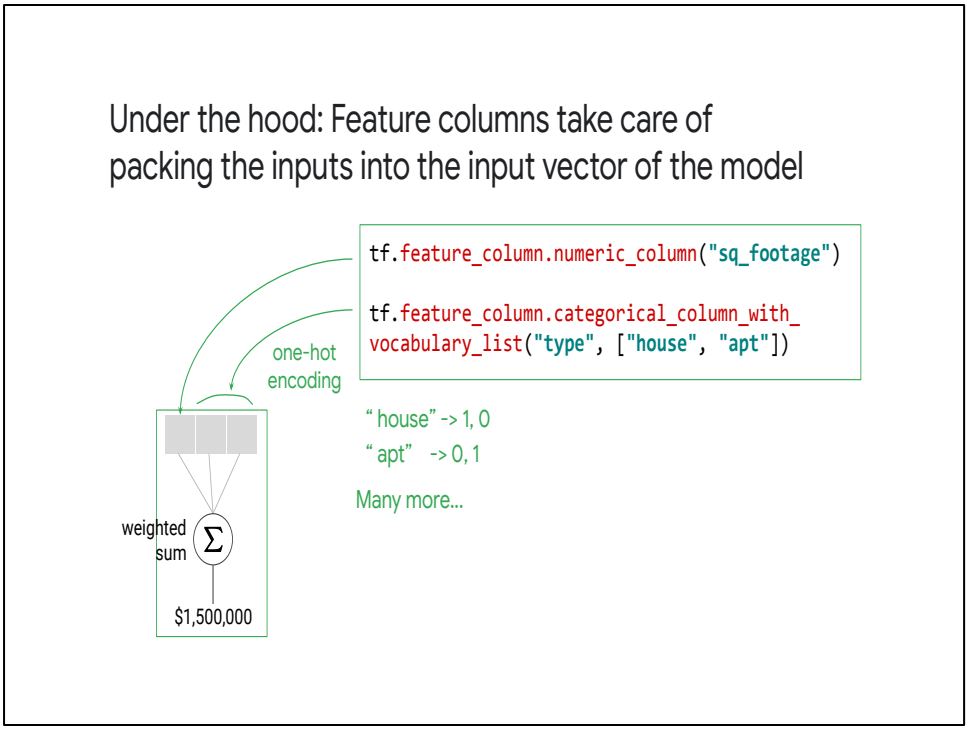
Under the hood: Feature columns take care of packing the inputs into the input vector of the model

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

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

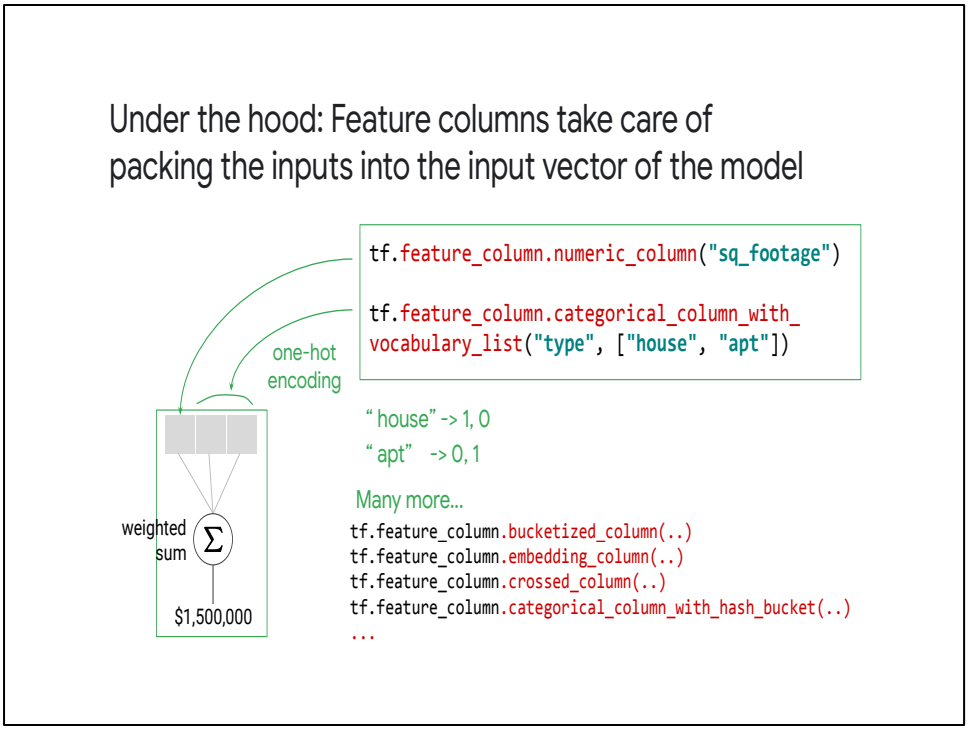
But why do you need feature columns in the content of model building? Do you remember how they get used? Let's break it down



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



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



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

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

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

set up numeric ranges

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

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

...
```

create bucketized columns for pickup latitude and pickup longitude

Bucketized column helps with discretizing continuous feature values. In this example, if we were to consider the latitude and longitude of the house or apartment we are training/predicting on, we would not want to feed the raw latitude/longitude values. Instead, we would create buckets that could group the range of values for latitude and longitude.

Representing feature columns as sparse vectors

These are all different ways to create a categorical column.

If you know the keys beforehand:

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

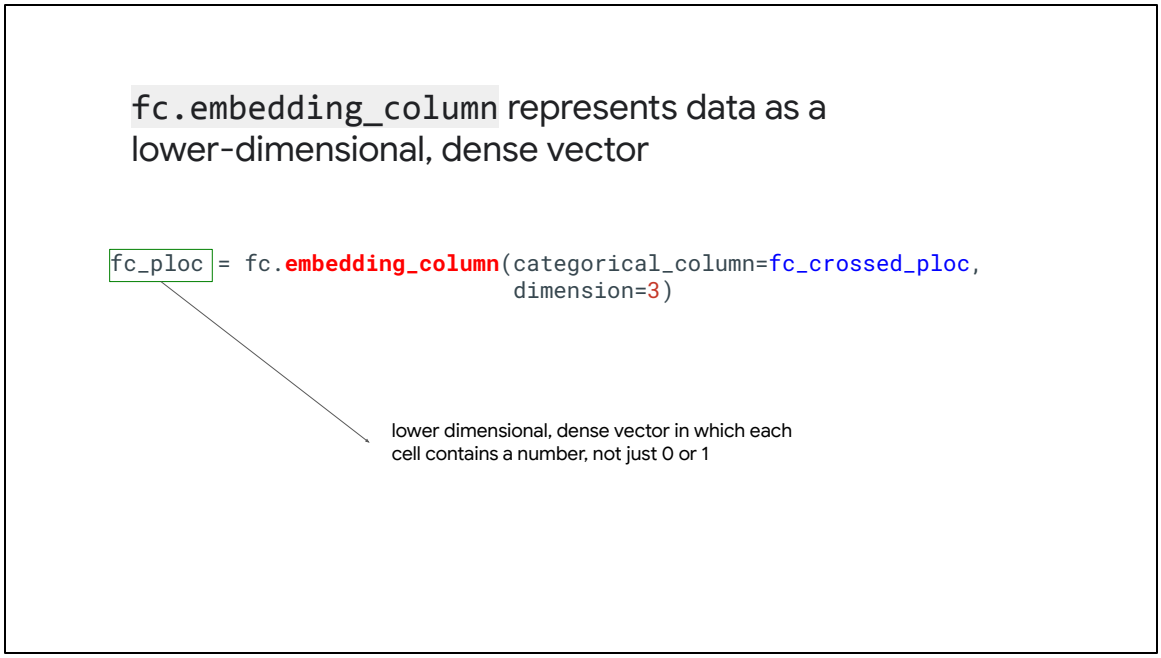
If your data is already indexed; i.e., has integers in [0-N):

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

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

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

If you are thinking this seems familiar, and just like vocabulary building for categorical columns, you are absolutely correct.
Categorical columns are represented by TensorFlow as sparse tensors.
So, categorical columns are an example of something that is sparse.
TensorFlow can do math operations on sparse tensors without having to convert them into dense.
This saves memory, and optimizes compute.



As the number of categories of a feature grow large, it becomes infeasible to train a neural network using one-hot encodings. Recall that we can use an embedding column to overcome this limitation. Instead of representing the data as a one-hot vector of many dimensions, an embedding column represents that data as a lower-dimensional, dense vector in which each cell can contain ANY number, not just 0 or 1.

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

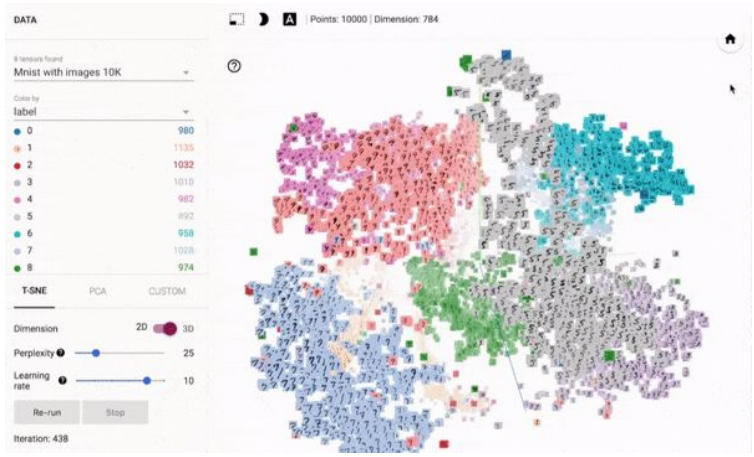
How can we visually cluster 10,000 variations of handwritten digits to look for similarities? Embeddings!

Generally, Neural network embeddings have 3 primary purposes:

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

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

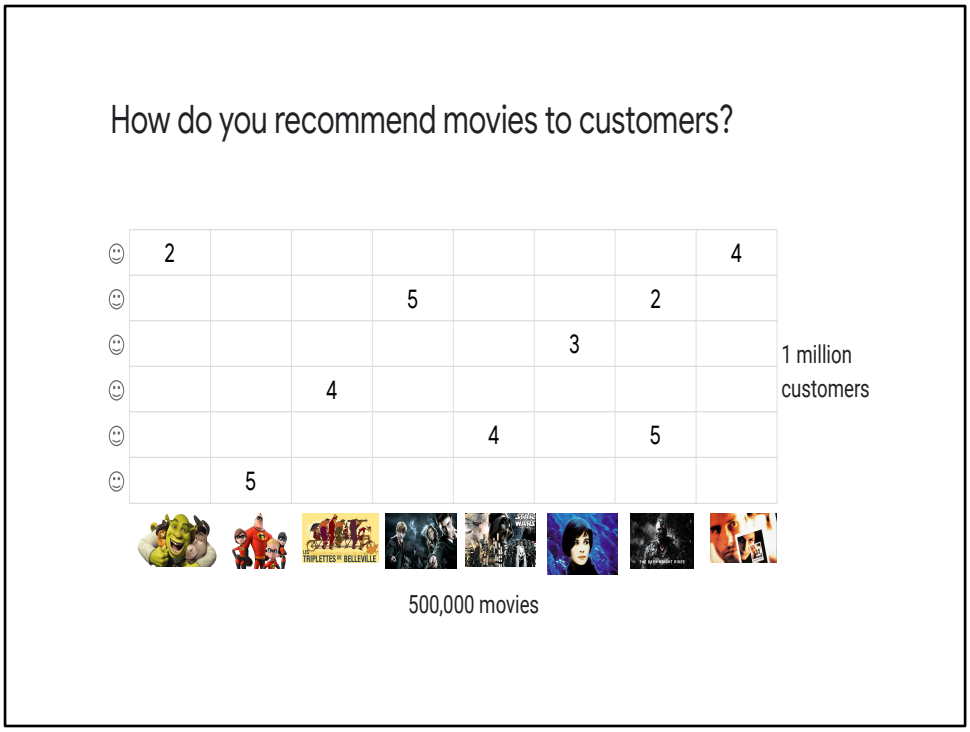
Embeddings are everywhere in modern machine learning



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

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

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



Next, let's look at embeddings in the context of movie recommendations.

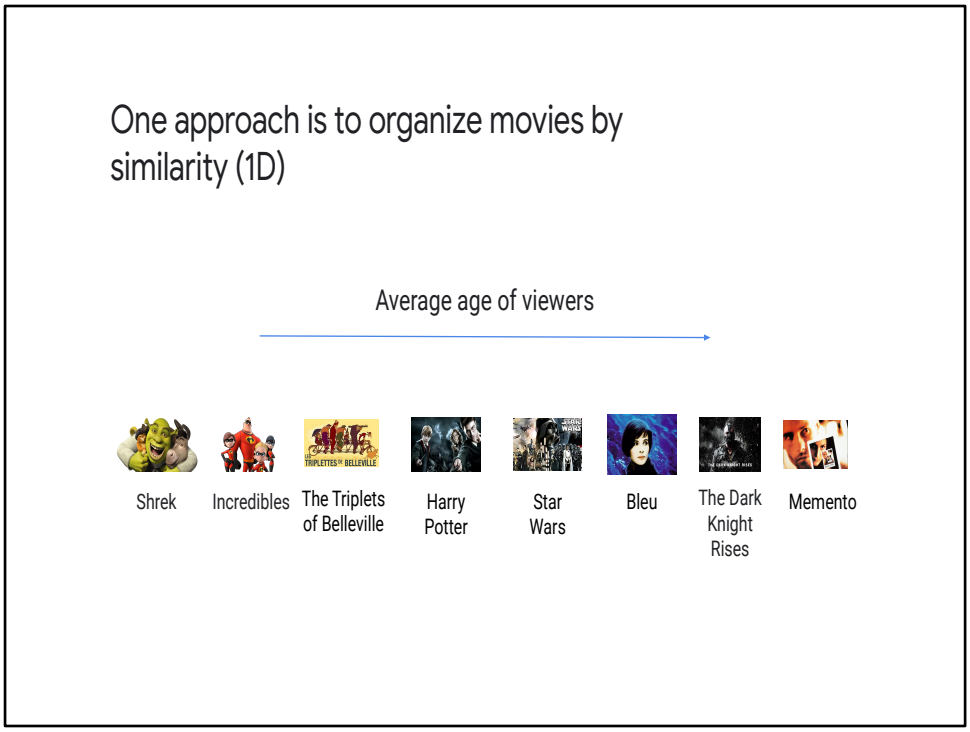
Let's say that we want to recommend movies to customers.
Let's say that our business has a million users and 500,000 movies. Remember that number.
That is quite small, by the way. YouTube and eight other Google properties have a billion users!

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

So, what's our input dataset? Our input dataset, if we represent it as a matrix, is 1 million rows and 500,000 columns.
The numbers in the diagram denote movies that customers have watched and rated.

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

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



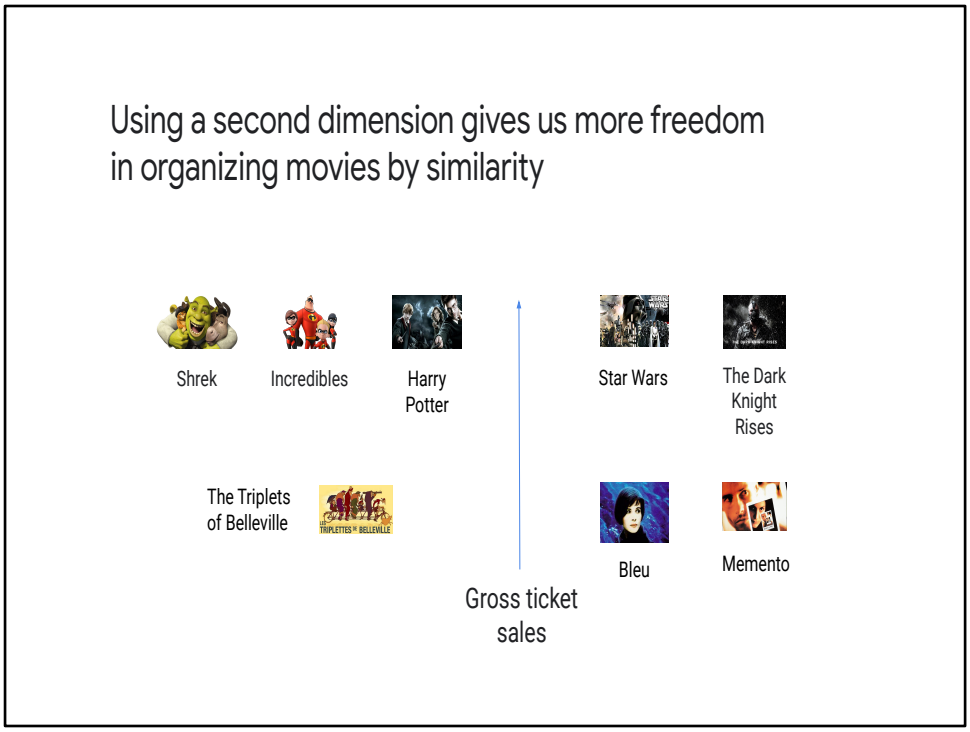
One approach is to organize movies by similarity using some attribute of the movies.

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

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

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

How do we solve this problem?



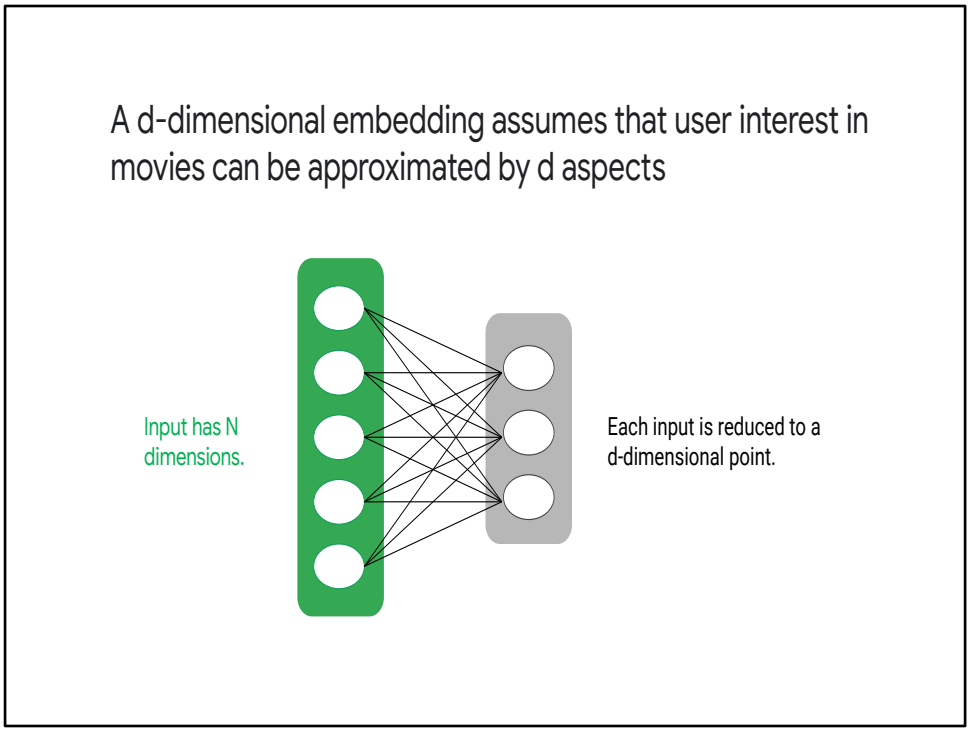
What if we add a second dimension? Perhaps the second dimension is the total number of tickets sold for that movie when it was released in theaters.

Now, we see that Star Wars and The Dark Knight Rises are close to each other.
Bleu and Memento are close to each other.
Shrek, Incredibles are close to each other as well.
Harry Potter is in between the cartoons and Star Wars in that kids watch it, some adults watch it, and it's a blockbuster.

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

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

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



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

What is N in the case of the movies we looked at?
[pause]
500,000-right? Remember that the movieId is a categorical feature and we'd normally be one-hot encoding it.
So N = 500,000

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

The key point is that d is much, much less than N.
And the assumption is that user interest in movies can be represented by some d aspects.

A good starting point for number of embedding dimensions

Lower dimensions -> less accuracy, more lossy compression

Higher dimensions -> overfitting, slow training.

$$dimensions \approx \sqrt[4]{possible\ values}$$

Empirical tradeoff.

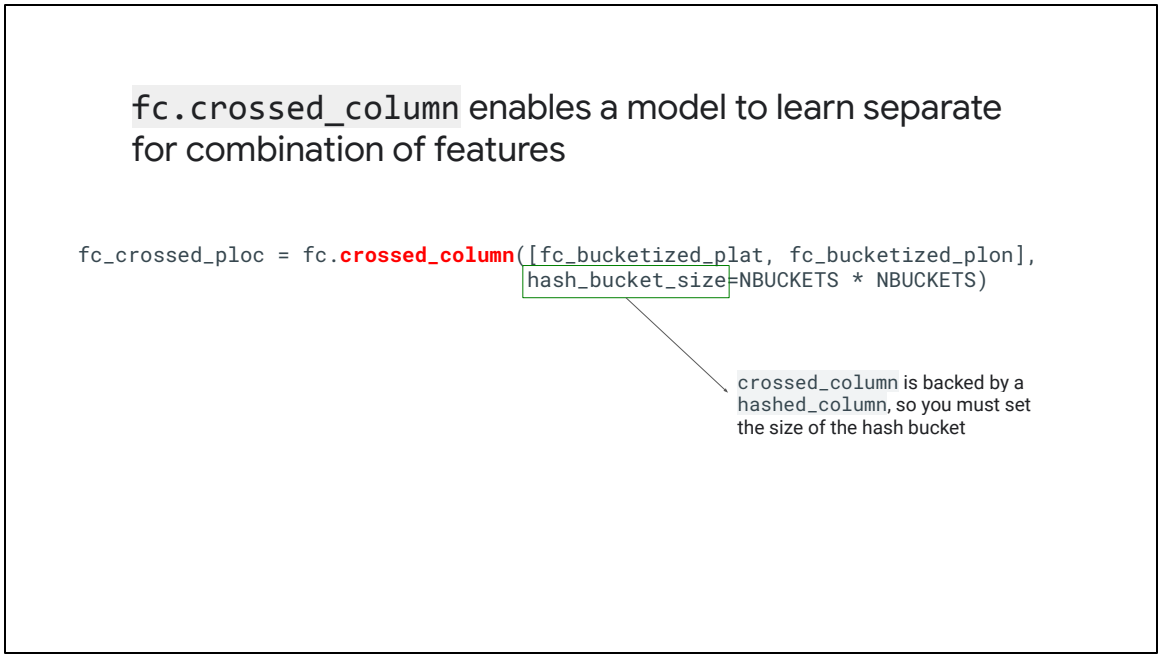
In all our examples, we used 3 for the number of embeddings. You can use different numbers, of course.
But what number should you use before you train?

This is a hyperparameter to your machine learning model. Hyperparameter means you set it before training. You will have to try different numbers of embedding dimensions because there is a tradeoff.
Higher-dimensional embeddings can more accurately represent the relationships between input values.
However, the more dimensions you have, the greater the chance of overfitting.
Also, the model gets larger and leads to slower training

A good starting point is to go with the fourth root of the total number of possible values.

For example, if you are embedding movieId and you have 500,000 movies in your catalog, a good starting point might be the fourth root of 500,000.
The square root of 500,00 is about 700. And the square root of 700 is about 26. So, I'd probably start at around 25.
In the hyperparameter tuning, I would specify a search space of perhaps 15 to 35.

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



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

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

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

Training input data requires dictionary of features and a label

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

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

Back to our real estate example. To train the model, you simply need to write an input function that returns the features named as in the feature columns. Since you are training, you also need the correct answers called “labels”. And now you can call the train function from the Keras API or from a custom model build, which will train the model by repeating this dataset 100 times, for example.

Create input pipeline using tf.data

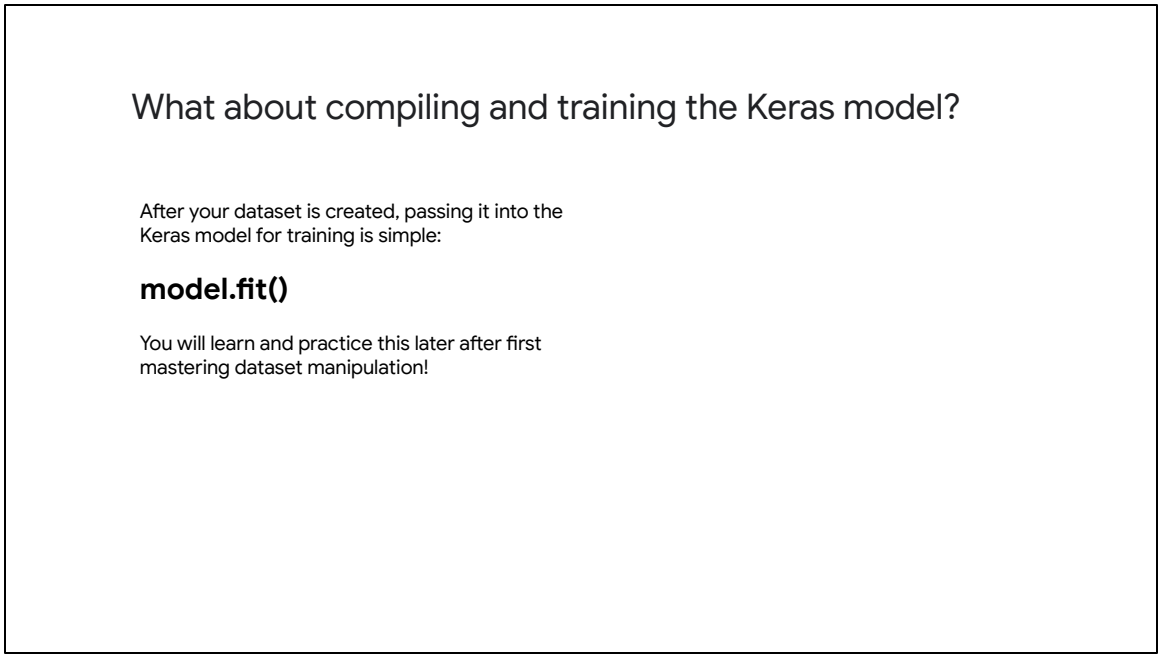
```
def create_dataset(pattern, batch_size=1, mode=tf.estimator.ModeKeys.EVAL):
    dataset = tf.data.experimental.make_csv_dataset(
        pattern, batch_size, CSV_COLUMNS, DEFAULTS)
    dataset = dataset.map(features_and_labels)
    if mode == tf.estimator.ModeKeys.TRAIN:
        dataset = dataset.shuffle(buffer_size=1000).repeat()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

    return dataset
```

You will see how batching works later but for those of you who already know about the concept of batching, the code as written here trains on a single batch of data at each step and this batch contains the entire dataset.
When passing data to the built-in training loops of a model, you should either use **Numpy arrays** (if your data is small and fits in memory) or **tf.data Dataset** objects.

Use DenseFeatures layer to input feature columns to the Keras model

```
feature_columns = [...]


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


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

...
```

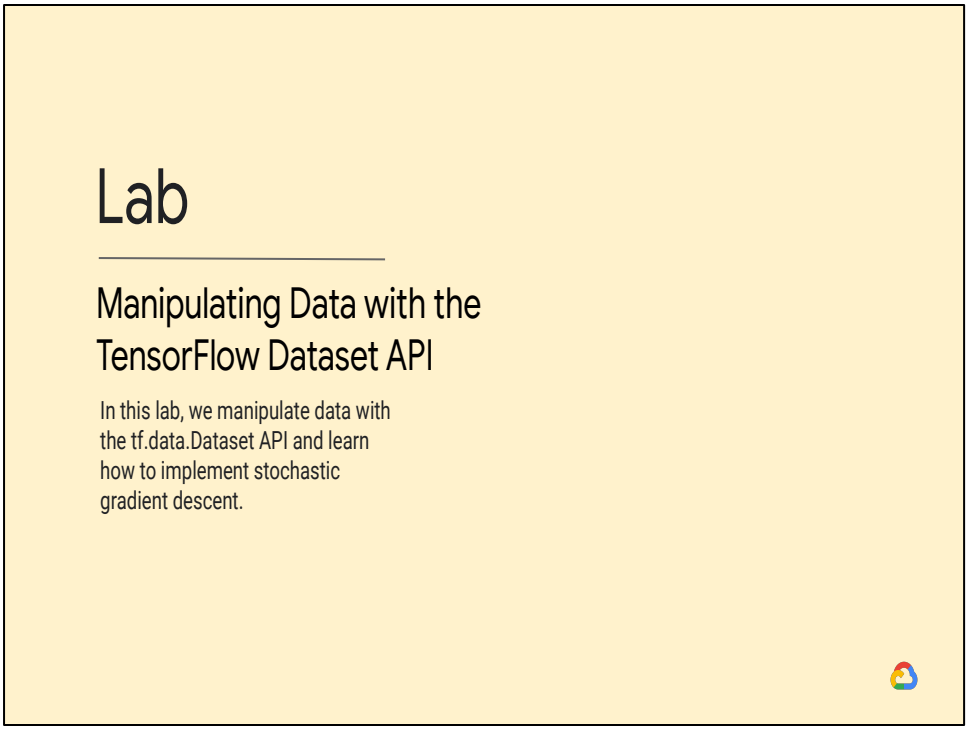
Once you have defined the feature columns, you can use a DenseFeatures layer to input them to a Keras model. This layer is simply a layer that produces a dense Tensor based on the given feature columns.



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

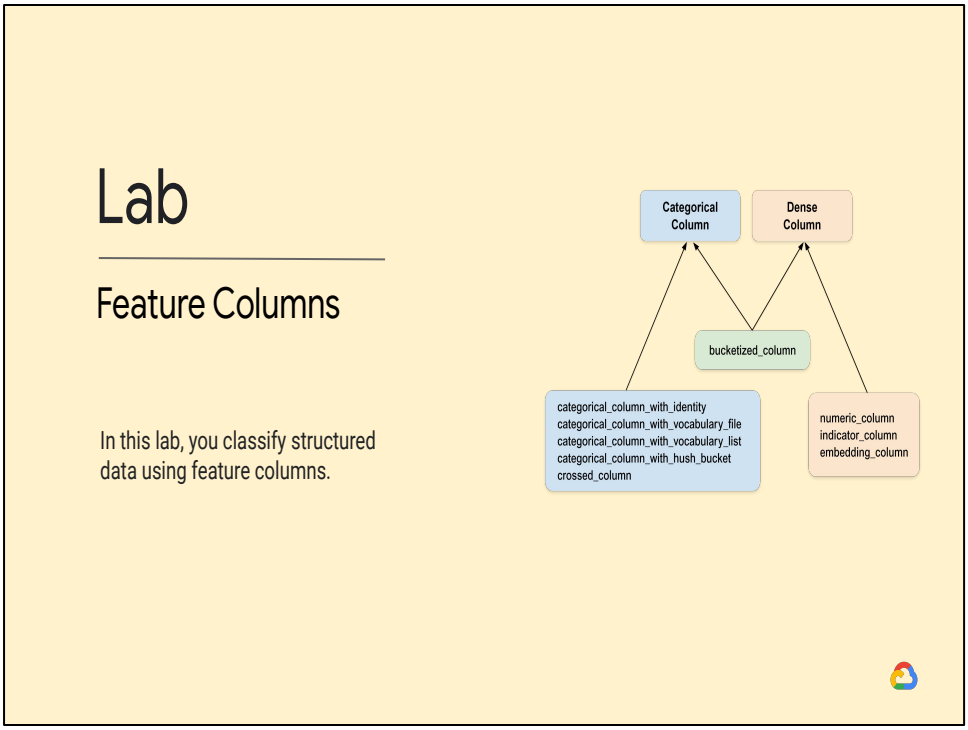
model.fit()

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



In this lab, we will start by refactoring the linear regression we implemented in a previous lab so that is takes its data from tf.data.Dataset, and we will learn how to implement **stochastic gradient descent** with it. In this case, the original dataset will be synthetic and read by the tf.data API directly from memory.

Link to lab: [ML on GCP C3] TensorFlow Dataset API (qwiklabs). Github repo link here. CBL122



In the upcoming lab, you classify structured data (such as tabular data in a CSV file) using feature columns. Feature columns serve as a bridge to map from columns in a CSV file to features used to train a model.

Link to lab: [ML on GCP C3] Introduction to Feature Columns (qwiklabs). github repo link here. CBL252

Agenda

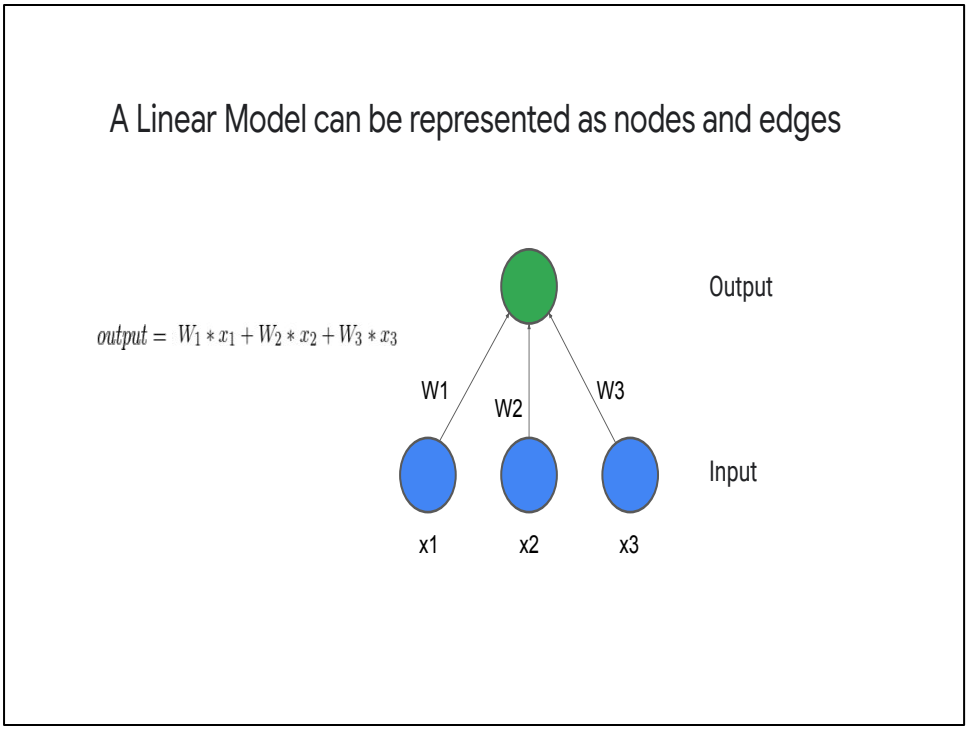
- Activation Functions
- Neural Networks with TF 2 and Keras
- Regularization

In this module, we will talk about activation functions and how they are needed to allow deep neural networks to capture nonlinearities of the data. Then we will learn how the Sequential and the Functional APIs of Keras allow us to simply write neural network models and how we can deploy models to Cloud AI Platform to serve them in a scaled fashion. We will finally discuss how to avoid overfitting by applying regularization to model training. So, let's jump right into it.

Agenda

- **Activation Functions**
- Neural Networks with TF 2 and Keras
- Regularization

Let's take a look at activation functions and how they help training deep neural network models.



Here is a graphical representation of a linear model. We have three inputs on the bottom, x1, x2, and x3, shown by the blue circles. They're combined with some weight (w), given to them on each edge (those are the arrows pointing up), to produce an output (which is the green circle). There often is an extra bias term, but for simplicity it isn't shown here. This is a linear model since it is of the form y = w1 * x1 + w2 * x2 + w3 * x3.

Add Complexity: Non-Linear?

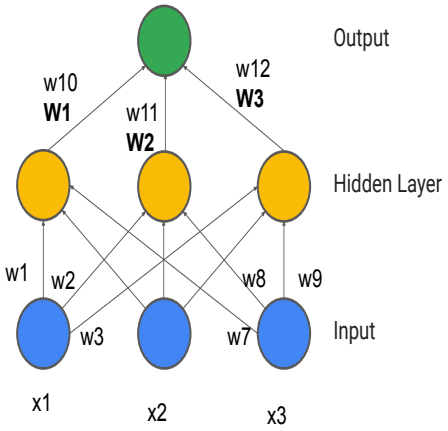
$$output = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

((w10 * w1) + (w10 * w4) + (w10 * w7)) * x1

+ ((w11 * w1) + (w11 * w4) + (w11 * w7)) * x2

+ ((w12 * w1) + (w12 * w4) + (w12 * w7)) * x3

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



We can substitute each group of weights for a new weight. Look familiar? This is exactly the same linear model as before despite adding a hidden layer of neurons.

So what happened?

Add Complexity: Non-Linear?

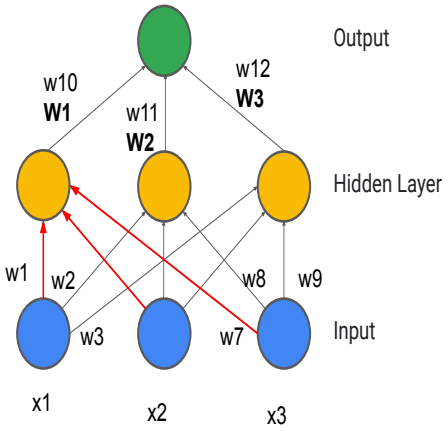
$$output = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

((w10 * w1) + (w10 * w4) + (w10 * w7)) * x1

+ ((w11 * w1) + (w11 * w4) + (w11 * w7)) * x2

+ ((w12 * w1) + (w12 * w4) + (w12 * w7)) * x3

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



The first neuron of the Hidden Layer on the left takes the weights from all three Input nodes (those are the red arrows)

Add Complexity: Non-Linear?

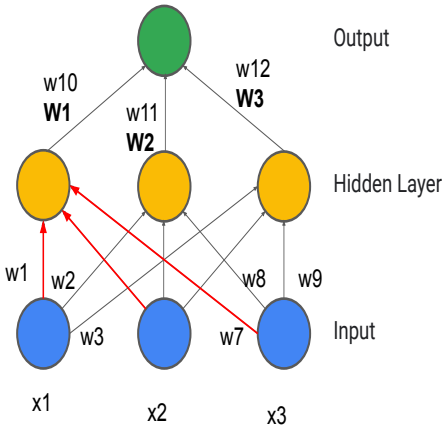
$$output = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

((w10 * **w1**) + (w10 * **w4**) + (w10 * **w7**)) * x1

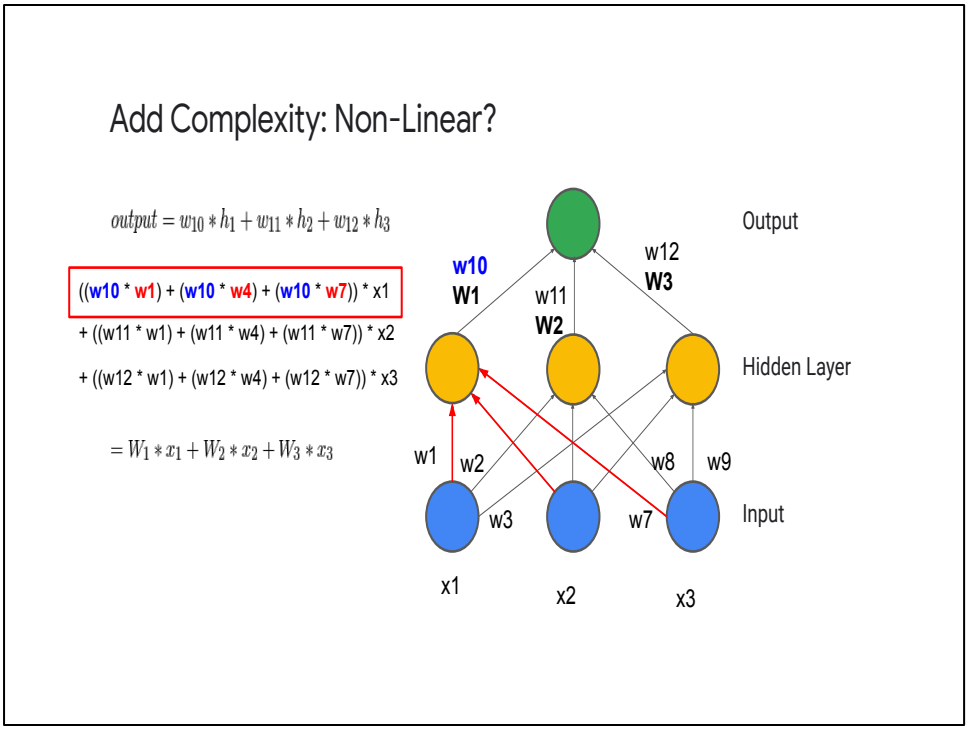
+ ((w11 * w1) + (w11 * w4) + (w11 * w7)) * x2

+ ((w12 * w1) + (w12 * w4) + (w12 * w7)) * x3

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



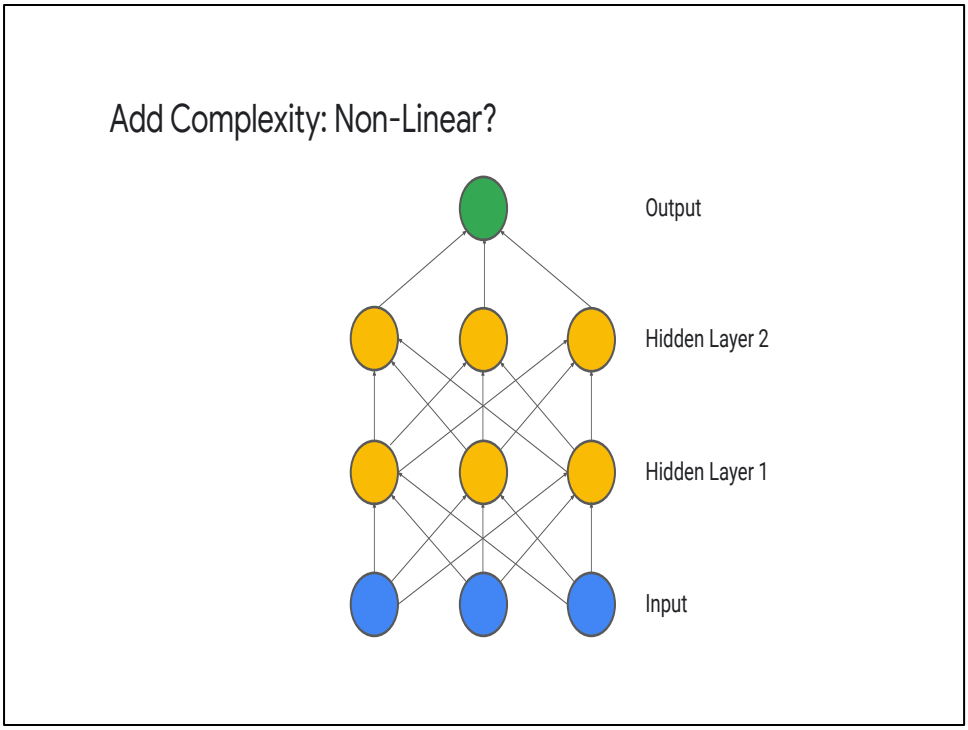
And those are little w1 w4 and w7 respectively as you see highlighted here



And then you take the new weight that is the output of the first neuron which is our little w10 now and one of three weights into the final output. You can see that we'll do this two more times for the other two yellow neurons and their inputs from x1 x2 and x3 respectively.

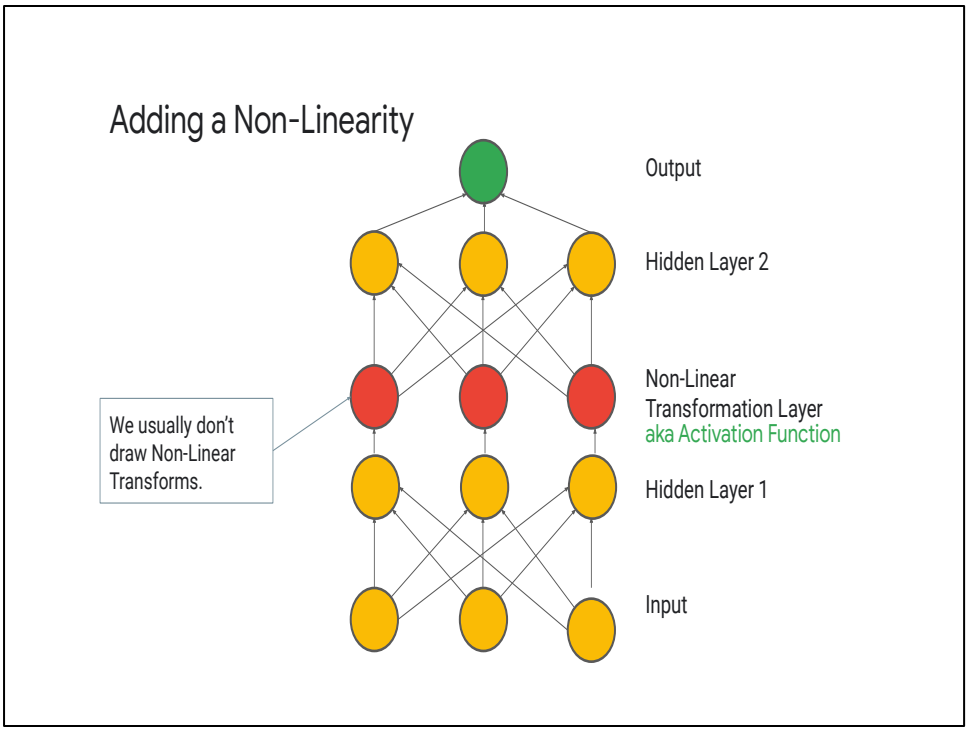
You can see that there is a ton of matrix multiplication going on behind the scenes. Honestly in my experience machine learning is basically taking arrays of various dimensionality and multiplying them against eachother (where one array aka "Tensor" could be the randomized starting weights of the model and the other is the input dataset and yet a third is the output of a hidden layer). You see it's all just simple math but a lot of it done really quickly.

Here still though we have a Linear model. Let's go deeper!



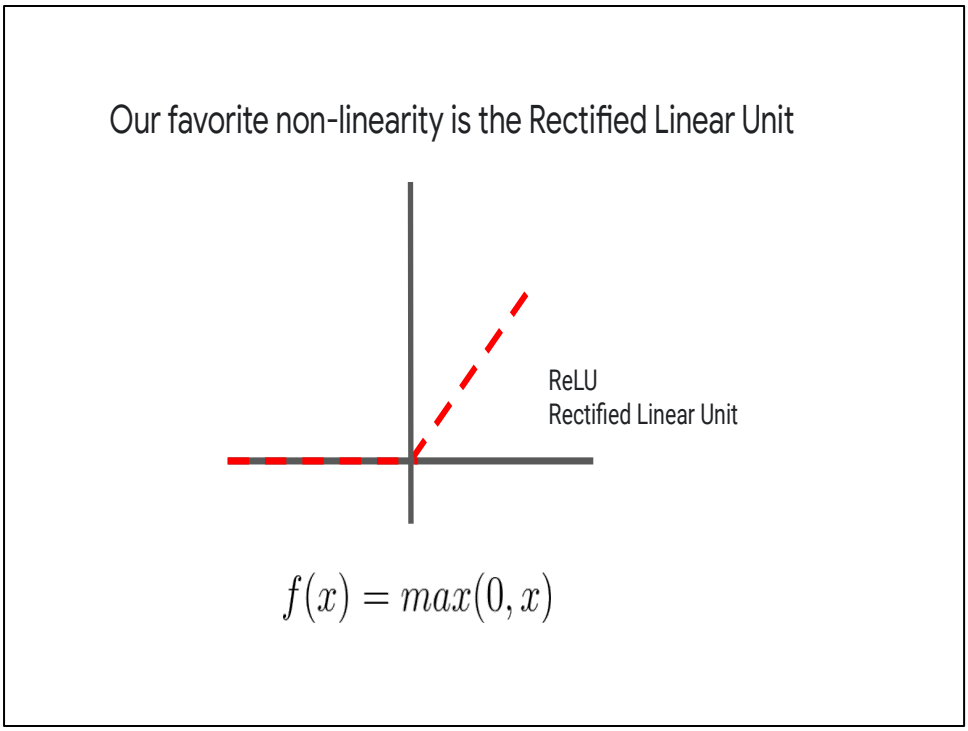
I know what you're thinking … What if we added another hidden layer? Unfortunately, this once again collapses all of the way back down into a single weight matrix multiplied by each of the three inputs. It is the same linear model! We could continue this process ad infinitum and it would be the same result, albeit a lot more costly computationally for training or prediction from a much, much more complicated architecture than needed.

So how can you escape having a linear model? By adding non linearity of course :)



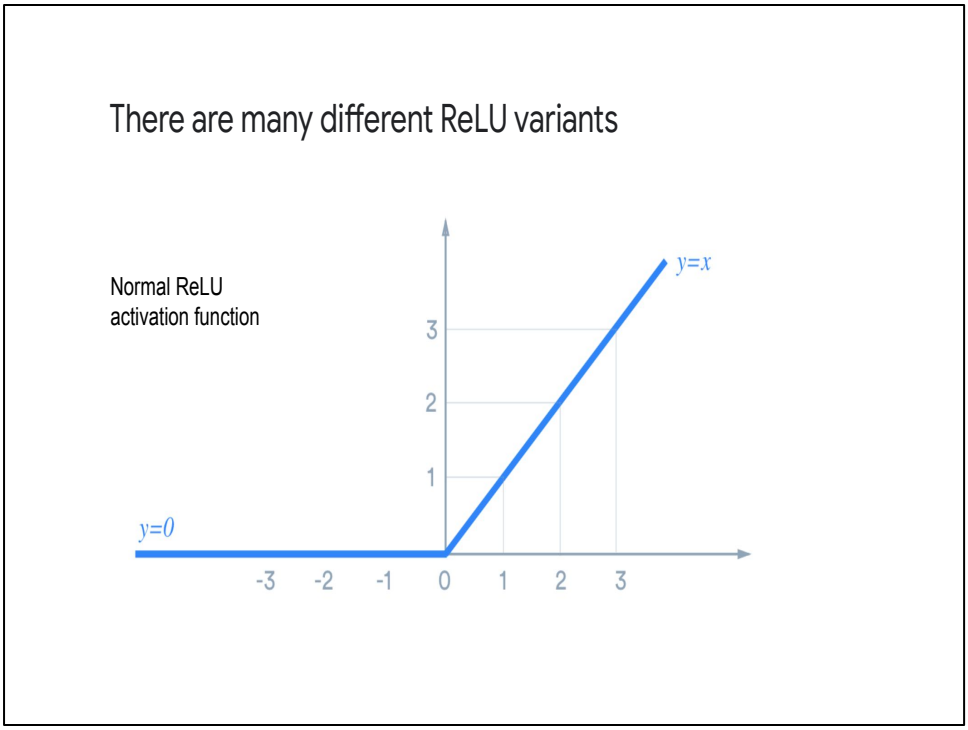
The solution is adding a nonlinear transformation layer which is facilitated by a nonlinear activation function such as Sigmoid, Tan-h, or ReLU. In thinking of terms of the graph that is created by Tensorflow, you can imagine each neuron actually having two nodes. The first node being the result of the weighted sum W * x + b and the second node being the result of that being passed through the activation function. In other words, there are the inputs to the activation function followed by the outputs of the activation function so the activation function acts as the transition point between layers.

Adding in this nonlinear transformation is the only way to stop the neural network from condensing back into a shallow network. Even if you have a layer with nonlinear activation functions in your network, if elsewhere in the network you have 2 or more layers with linear activation functions, those can still be collapsed into just one network. Usually, neural networks have all layers nonlinear for the first n minus 1 layers and then have the final layer transformation be linear (for regression) or sigmoid or softmax (for classification). It all depends on what you want the output to be.



There are many nonlinear activation functions with sigmoid, and the scaled and shifted sigmoid, hyberbolic tangent being some of the earliest. However, as I mentioned, these can have saturation which leads to the vanishing gradient problem where, with zero gradients, the model's weights don't update and training halts.

The Rectified Linear Unit, or ReLU for short, is one of our favorites because it's simple and works well. In the positive domain it is linear - so we don't have saturation - whereas in the negative domain the function is 0. Networks with ReLU hidden activations often have 10 times the speed of training than networks with sigmoid hidden activations. However, due to the negative domain's function always being zero, we can end up with ReLU layers dying. What I mean by this is that when you start getting inputs in the negative domain then the output of the activation will be zero which doesn't help in the next layer in getting the inputs into the positive domain. This compounds and creates a lot of zero activations. During backpropagation when updating the weights, since we have to multiply our error's derivative by the activation, we end up with a gradient of zero, thus a weight update of 0, and thus the weights don't change and training fails for that layer.

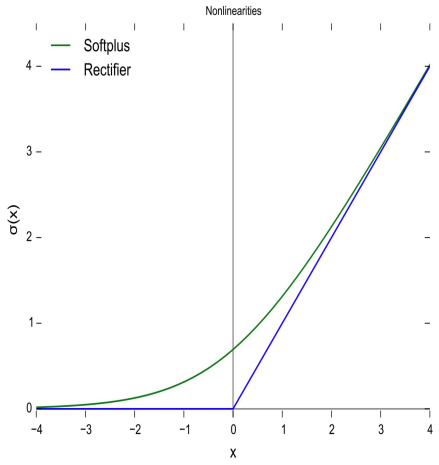


Fortunately, a lot of clever methods have been developed to slightly modify the ReLU and avoid the "dying ReLU" effect to ensure training doesn't stall, but still with much of the benefits of the vanilla ReLU.

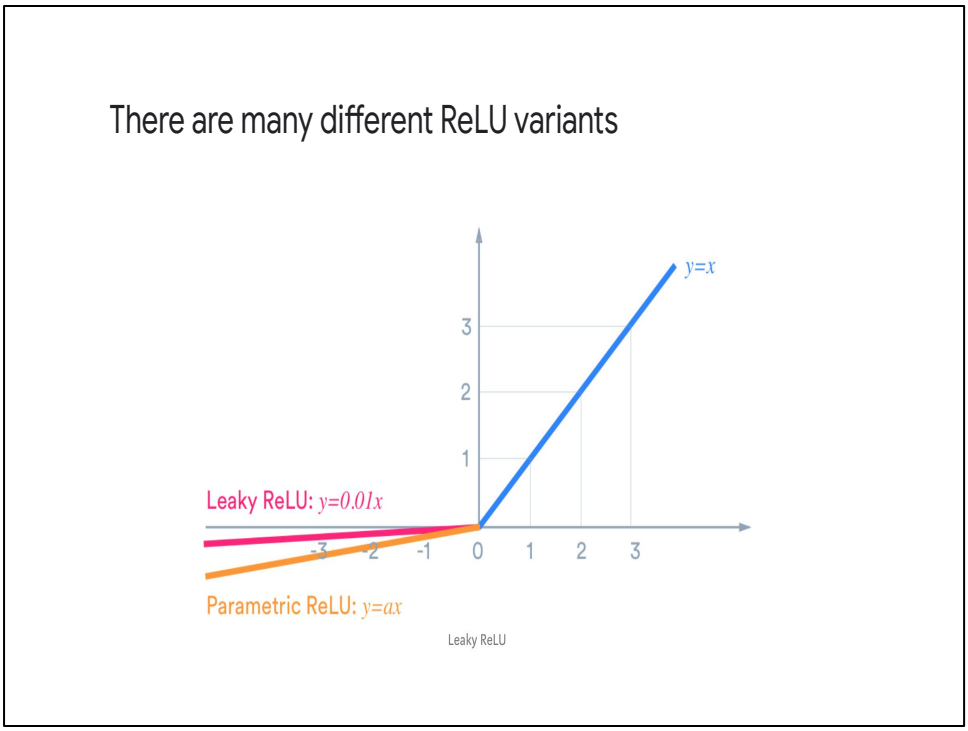
Here again is the vanilla ReLU. The maximum operator can also be represented by the piecewise linear equation where less than 0 the function is 0 and greater than or equal to 0 the function is x.

Some extensions to ReLU meant to relax the non-linear output of the function and allow small negative values are:

There are many different ReLU variants

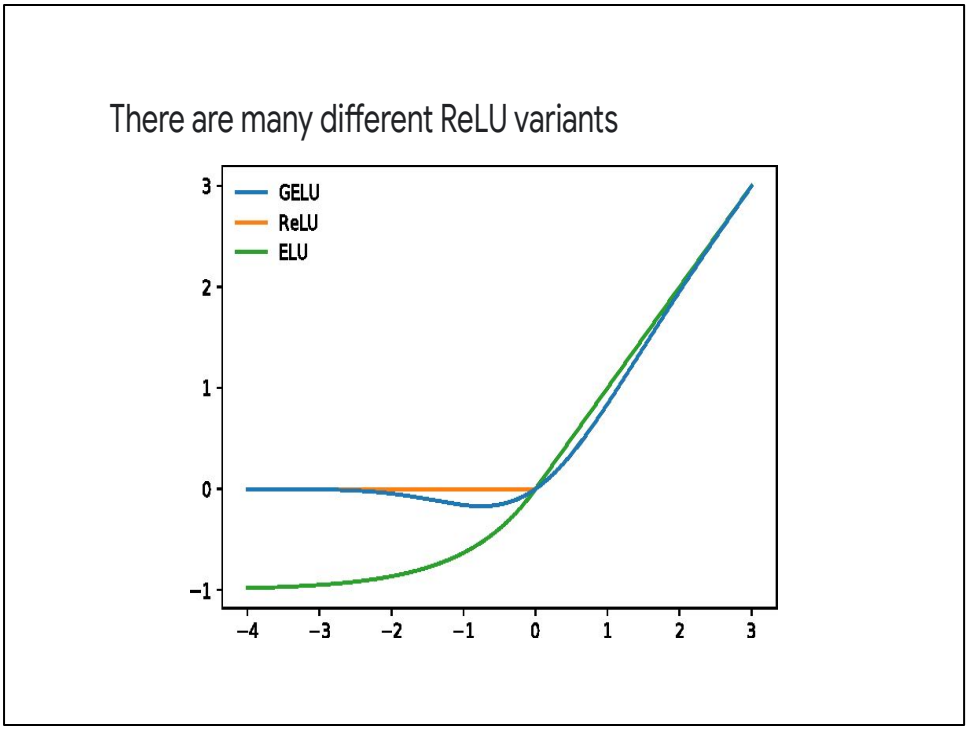


Softplus or SmoothReLU function. This function has as its derivative the logistic function. The logistic sigmoid function is a smooth approximation of the derivative of the rectifier.



The Leaky ReLU function is modified to allow small negative values when the input is less than zero. Its rectifier allows for a small, non-zero gradient when the unit is saturated and not active.

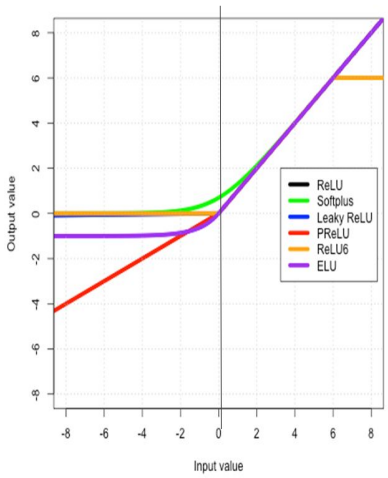
The Parametric ReLU (PReLU) learns parameters that control the leakyness and shape of the function. It adaptively learns the parameters of the rectifiers.



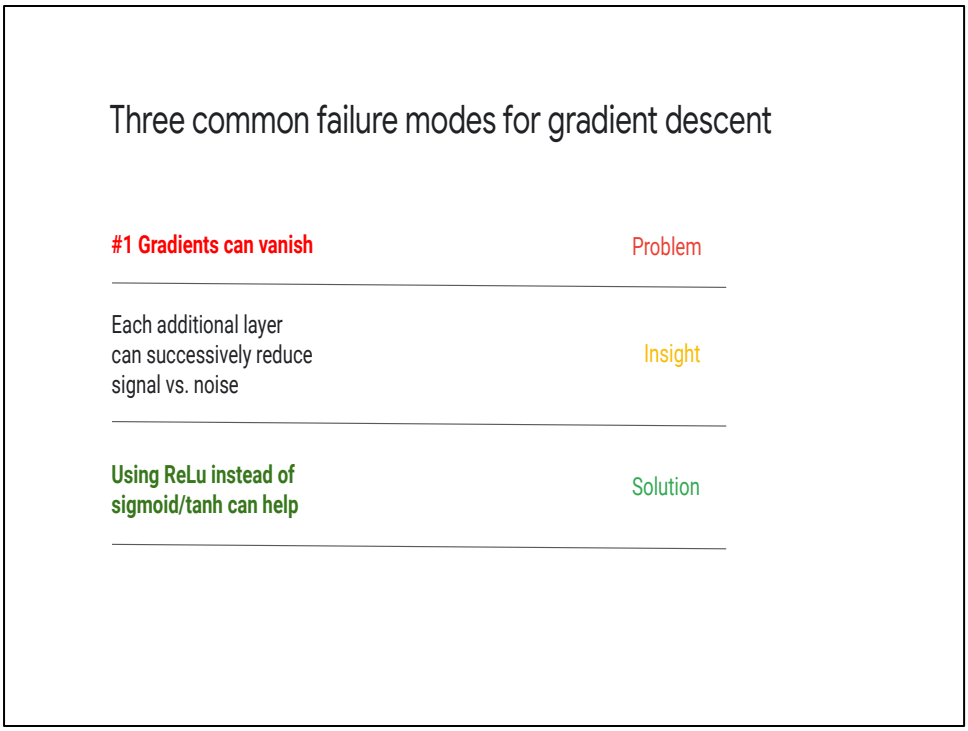
The Exponential Linear Unit, or ELU, is a generalization of the ReLU that uses a parameterized exponential function to transition from the positive to small negative values. Its negative values push the mean of the activations closer to zero. Mean activations that are closer to zero enable faster learning as they bring the gradient closer to the natural gradient.

The Gaussian Error Linear Unit (GELU), is a high-performing neural network activation function. It's nonlinearity results in the expected transformation of a stochastic regularizer which randomly applies the identity or zero map to a neuron's input.

There are many different ReLU variants

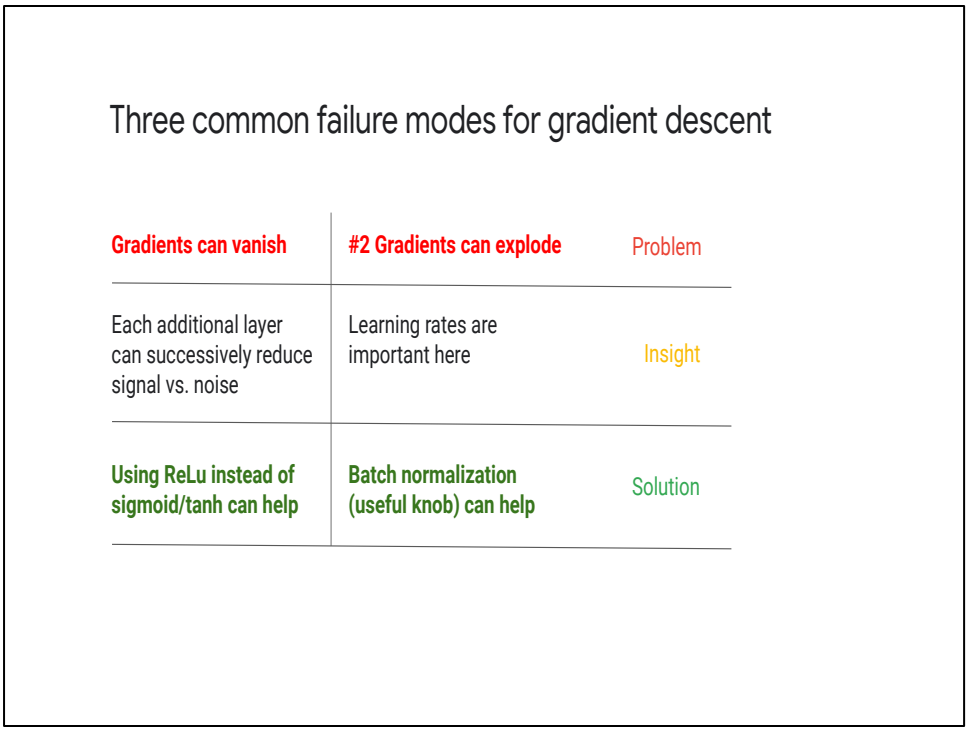


Here’s a quick visualization overlay showing the most popular ReLU variants.



Backpropagation is one of the traditional topics in an ML/neural networks course, but at some level it's kind of like teaching people how to build a compiler. It's essential for deeper understanding, but not necessarily needed for initial understanding. The main thing to know is that there's an efficient algorithm for calculating derivatives and TensorFlow will do it for you automatically. There are some interesting failure cases to talk about though such as vanishing gradients, exploding gradients, and dead layers.

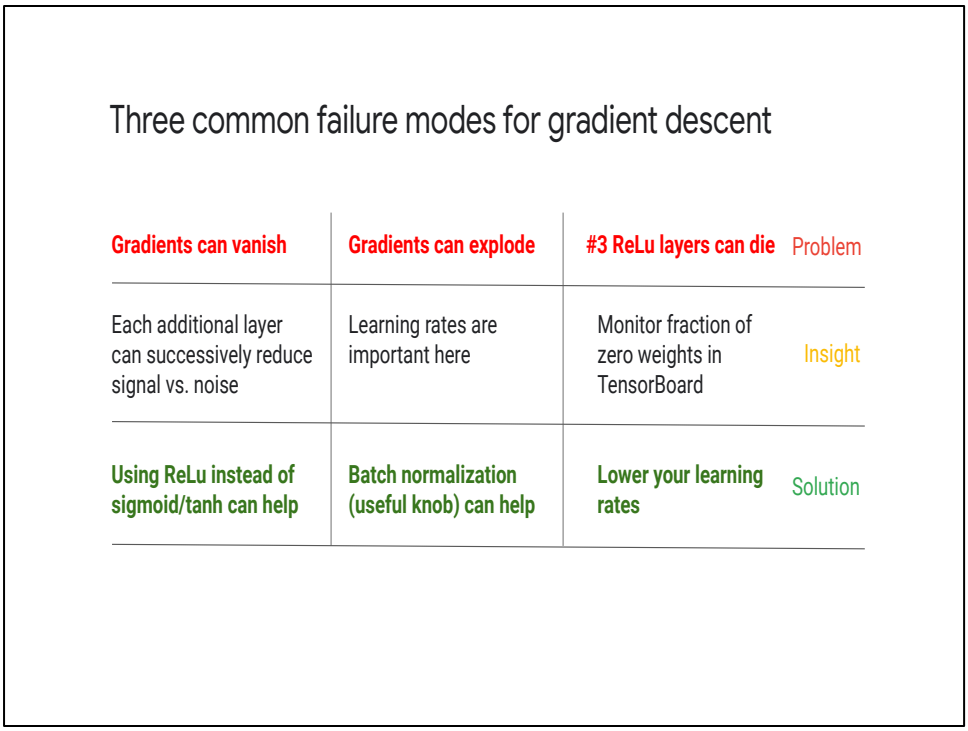
First, during the training process, especially for deep networks, gradients can vanish. Each additional layer in your network can successively reduce signal vs. noise. An example of this is when using sigmoid or tanh activation functions throughout your hidden layers. As you begin to saturate, you end up in the asymptotic regions of the function which begin to plateau. The slope is getting closer and closer to approximately zero. When you go backwards through the network during backprop, your gradient can become smaller and smaller because you are compounding all of these small gradients until the gradient completely vanishes. When this happens, your weights are no longer updating and therefore training grinds to a halt. A simple way to fix this is to use non-saturating, nonlinear activation functions such as ReLUs, ELUs, etc.



Next, we can also have the opposite problem, where gradients explode, by getting bigger and bigger until our weights get so large we overflow. Even starting with relatively small gradients such as a value of 2 can compound and become quite large over many layers. This is especially true for sequence models with long sequence lengths. Learning rates can be a factor here because in our weight updates remember we multiply the gradient with the learning rate and then subtract that from the current weight. So even if the gradient isn't that big, with a learning rate greater than 1 it can now become too big and cause problems for us and our network.

There are many techniques to try and minimize this such as weight regularization and smaller batch sizes. Another technique is gradient clipping, where we check to see if the norm of the gradient exceeds some threshold, which you can hyperparameter tune, and if so then you can rescale the gradient components to be below your maximum.

Another useful technique is batch normalization, which solves a problem called internal covariate shift. It speeds up training because gradients flow better. It also can often use a higher learning rate and might be able to get rid of dropout which slows computation down due to its own kind of regularization due to mini-batch noise. To perform batch normalization, first find the mini-batch mean, then the mini-batches standard deviation, then normalize the inputs to that node, then scale and shift by y = gamma times x plus beta where gamma and beta are learned parameters. If gamma = sqrt(var(x)) & beta=mean(x), the original activation is restored. This way you can control the range of your inputs so that they don't become too large. Ideally you would like to keep your gradients as close to one as possible, especially for very deep nets, so that you don't compound and eventually underflow or overflow.



Another common failure mode of gradient descent is that ReLU layers can die. Fortunately, using TensorBoard, we can monitor the summaries during and after training of our neural network models. If using a canned DNN estimator, there is automatically a scalar summary saved for each DNN hidden layer showing the fraction of zero values of the activations for that layer. ReLUs stop working when their inputs keep them in the negative domain giving their activation a value of zero. It doesn't end there because then their contribution to the next layer is 0 because despite what the weights are connecting it to the next neurons, its activation is zero thus the input becomes zero. A bunch of zeros coming into the next neuron doesn't help it get into the positive domain and then these neurons' activations become zero too and the problem continues to cascade. Then we perform backprop and the gradients are zero so we don't update the weights and thus training halts. Not good.

We've talked about using leaky or parametric ReLUs or even the slower ELUs, but you can also lower your learning rates to help stop ReLU layers from not activating and thus dying. A large gradient, possibly due to too high of a learning rate, can update the weights in such a way that no datapoint will ever activate it again and since the gradient is zero we won't update the weight to something more reasonable so the problem will persist indefinitely.

Agenda

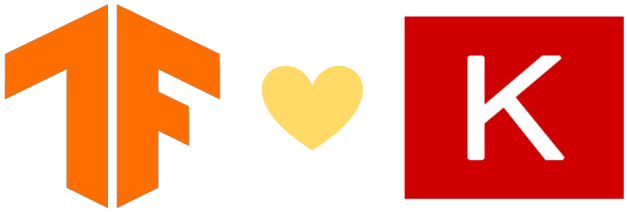
Activation Functions

Neural Networks with TF 2 and Keras

Regularization

Let's see how Tensorflow 2 and Keras make it easy to write models using neural networks.

Keras is built-in to TF 2.x



tf.keras is TensorFlow's high-level API for building and training deep learning models. It's used for fast prototyping, state-of-the-art research, and production, with three key advantages:

It is user-friendly

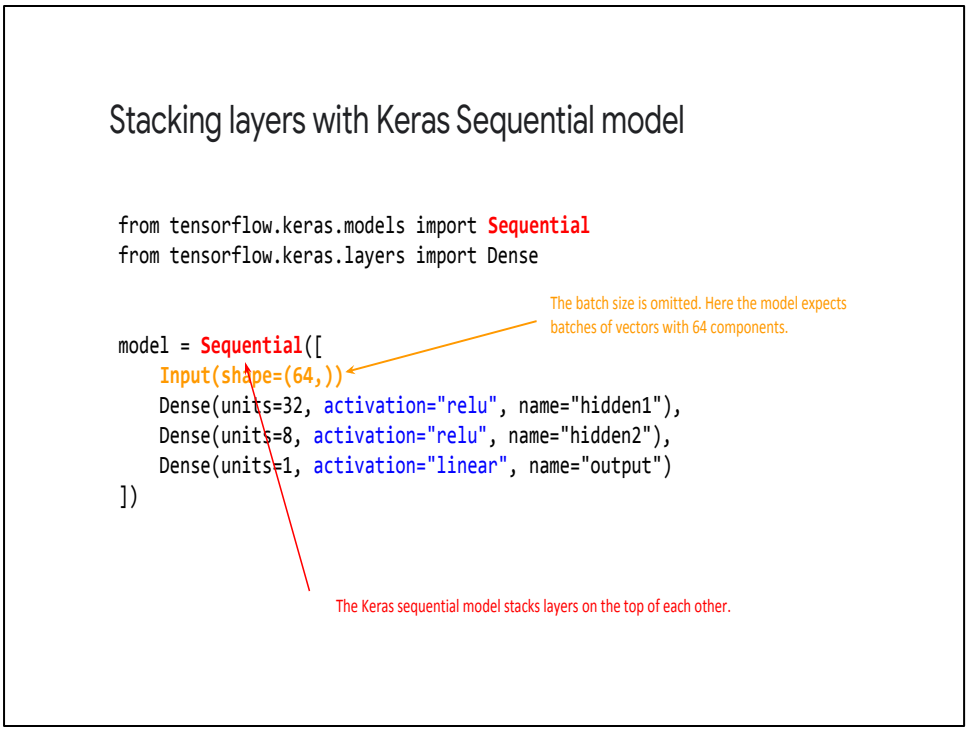
Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors.

It is modular and composable

Keras models are made by connecting configurable building blocks together, with few restrictions.

It is easy to extend

Write custom building blocks to express new ideas for research. Create new layers, metrics, loss functions, and develop state-of-the-art models.



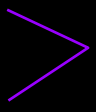
A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.
Sequential models are not advisable if:

- The model you're building has multiple inputs or multiple outputs
- Any of the layers of the model has multiple inputs or multiple outputs
- The model needs to do layer sharing
- The model has a non-linear topology such as a residual connection or multi-branches

```
%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')
])
```

In the code example, you see that there is one single dense layer being defined. That layer is defined with 10 nodes (or neurons) and the activation is softmax.

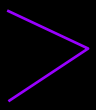
```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()


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



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

With a single layer, the model is linear. This example is able to perform logistic regression and classify examples across 10 classes.

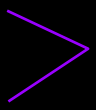
```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()


# 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

With the addition of another dense layer, the model becomes a neural network with one hidden layer but it is possible to map nonlinearities through the relu activation.

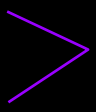
```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()



# 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)

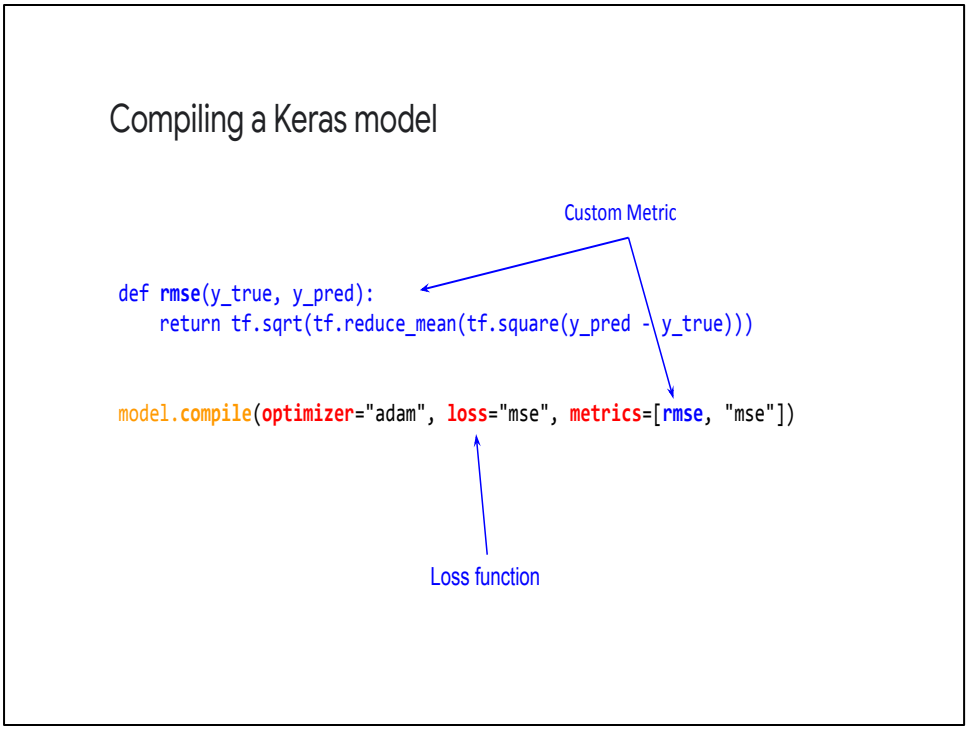
Once more than one layer is added to the network. It becomes a deep neural network.

```
%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')
])
```



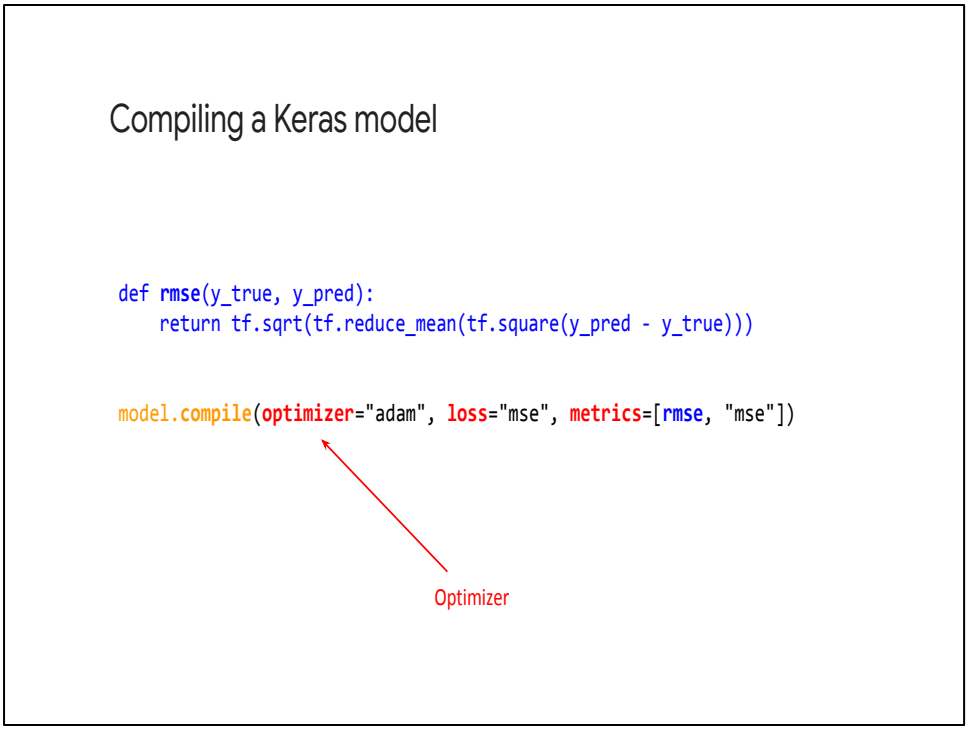
A deeper neural network

And an even deeper one. Needless to say, the deeper a neural network gets, the more powerful it becomes in learning patterns of the data. This can cause the model to overfit as it may learn almost all the patterns and not generalize to unseen data. There are mechanisms to avoid that, and we will talk about these later.

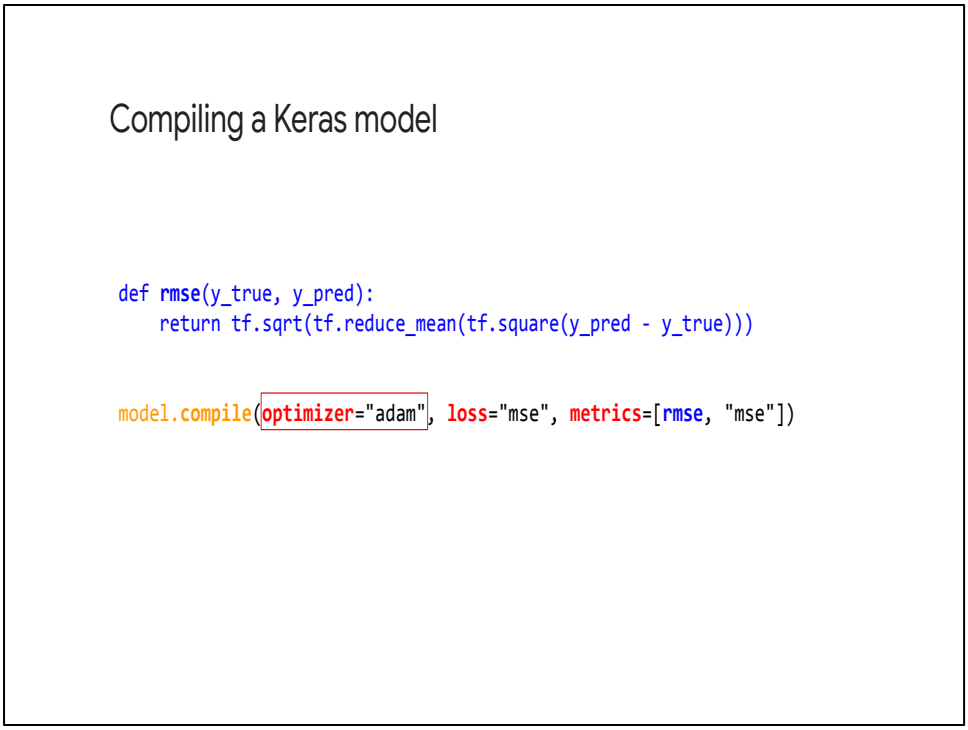


Once we define the model object, we compile it. During compilation, a set of additional parameters are passed to the method. These parameters will determine the optimizer that should be used, the loss function and the evaluation metric(s) (other parameter options are loss_weights, sample_weight_mode and weighted_metrics).

The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.



Optimizers tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by playing with the weights. An optimizer that is used commonly is SGD (Stochastic Gradient Descent). SGD is an algorithm that descends a slope (hence the name) to reach the lowest point on a surface. Think of the surface as the graphical representation of the data and the lowest point of the graph as where the error is minimum. Optimizers aim to take the model there.

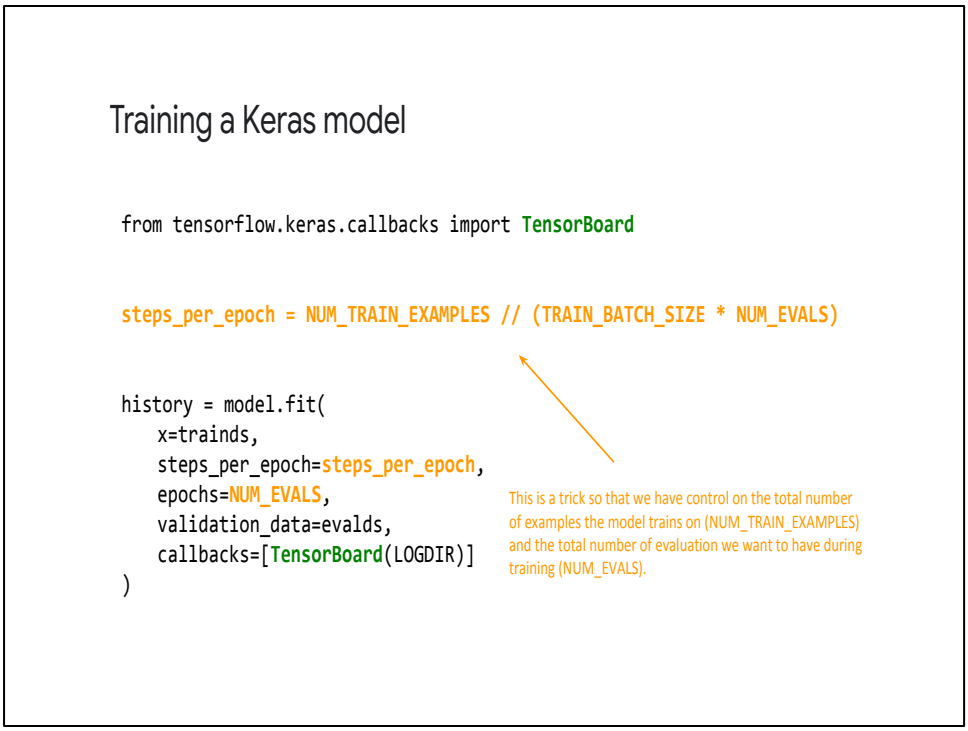


In this example, we are using Adam. Adam is an optimization algorithm that can be used, instead of the classical stochastic gradient descent procedure, to update network weights iteratively based in training data. The algorithm is straightforward to implement, besides being computationally efficient and having little memory requirements. Another advantage of Adam is its invariability to diagonal rescaling of the gradients.

Adam is well suited for models that have large dataset and/or many parameters. The method is also appropriate for problems with very noisy and/or sparse gradients and non-stationary objectives.

Some additional optimizers are:
Momentum which reduces learning rate when gradient values are small.
AdaGrad gives frequently occurring features low learning rates.
AdaDelta improves AdaGrad by avoiding reducing LR to zero
And **Ftrl** or "**Follow the regularized leader**" works well on wide models.

At this time, Adam and Ftrl make good defaults for Deep Neural Nets as well as Linear models.

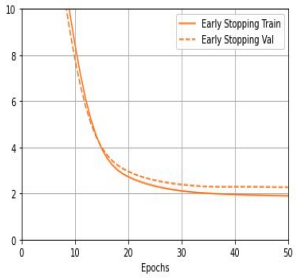


Now it is time to train the model that we defined. We train models in Keras by calling the `fit` method. You can pass parameters to fit that define the number of epochs (an epoch is a complete pass on the training dataset), steps per epoch (which is the number of batch iterations before a training epoch is considered finished), validation data and validation steps, batch_size (which determines the number of samples in each mini batch - its maximum is the number of all samples), and others such as callbacks. Callbacks are utilities called at certain points during model training for activities such as logging and visualization on tools such as Tensorboard.

Saving the training iterations to a variable allows for plotting...

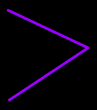
Training a Keras model

```
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)]
)
```



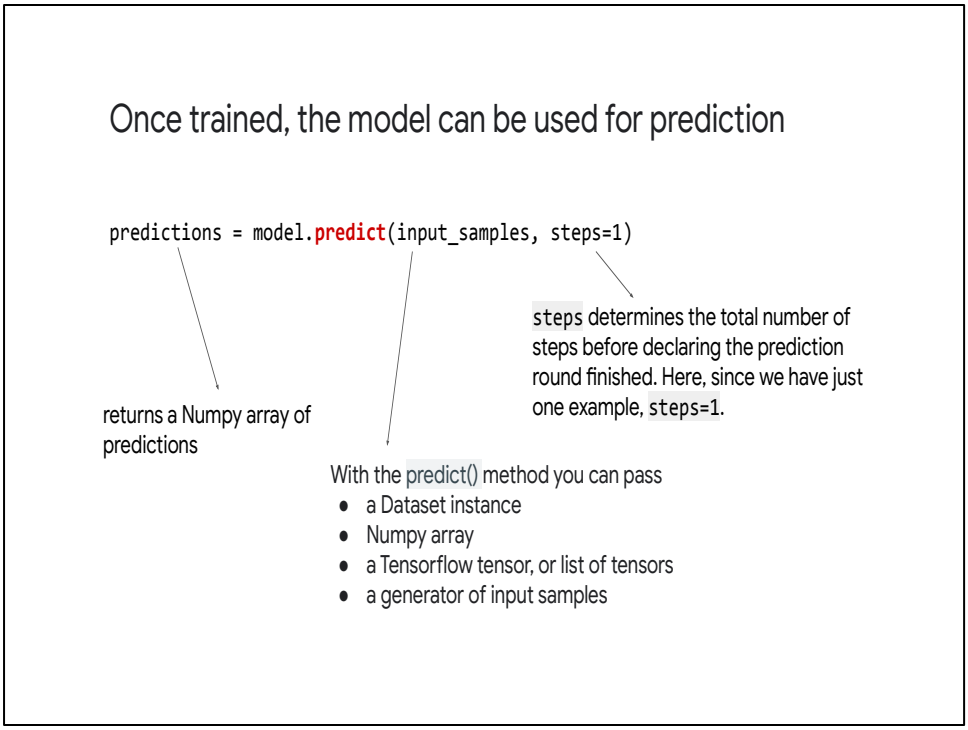
...of the chosen evaluation metric (Mean Absolute Error, Root Mean Square Error, Accuracy, etc) versus the epochs, for example.

```
%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

Here is a code snippet with all the steps put together: model definition, compilation, fitting and evaluation.



Once trained, the model can be used for predictions. You will need an input function that provides data for the prediction. Back to our example of the house pricing model, we could predict the house prices of examples of a 1500 sqf house and an 1800 sqf apartment, for example. The predict function in the tf.keras API returns a Numpy array(s) of predictions.

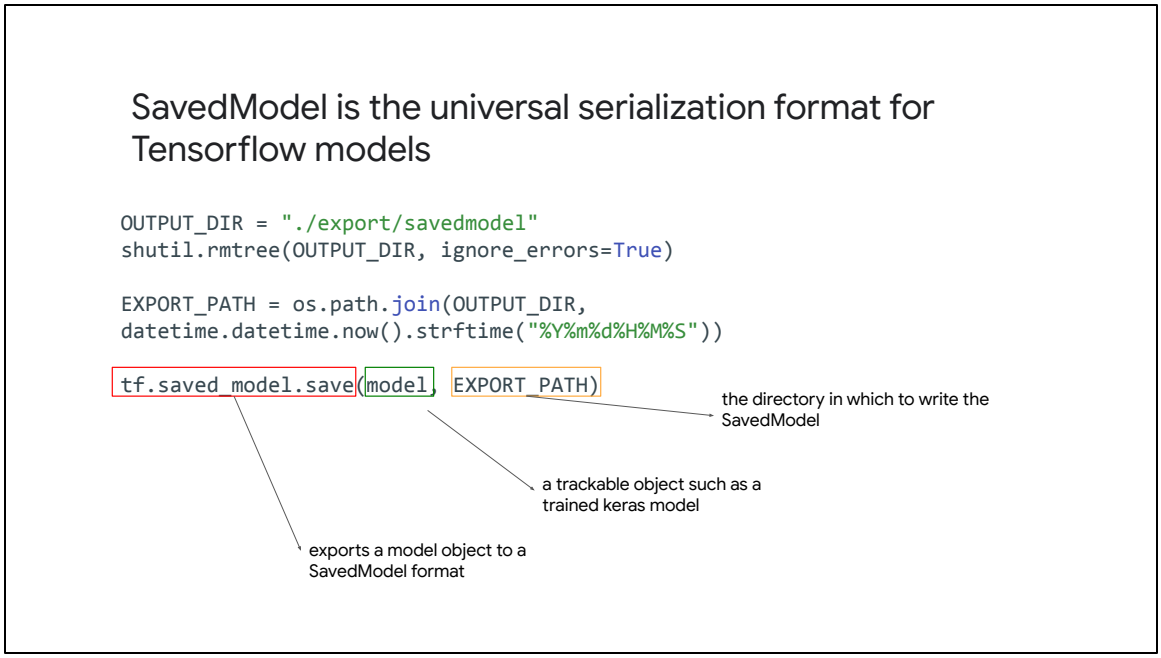
The steps parameter determines the total number of steps before declaring the prediction round finished. Here since we have just one example, we set steps=1 (setting steps=None would also work). Note, however, that if input_samples is a tf.data dataset or a dataset iterator, and steps is set to None, predict will run until the input dataset is exhausted.

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

Once you have selected the features, transformed them as needed, chosen your model architecture, applied any regularization that is necessary to ensure good performance, trained your model, and iterated through this process a couple of times, it is time to serve the model for prediction.

Of course, making individual predictions is not realistic, because we can't expect client code to have a model object in memory. For others to use our trained model, we'll have to save (or export) our model to a file, and expect client code to instantiate the model from that exported file.

We'll export the model to a TensorFlow SavedModel format. Once we have a model in this format, we have lots of ways to "serve" the model, from a web application, from JavaScript, from mobile applications, etc.



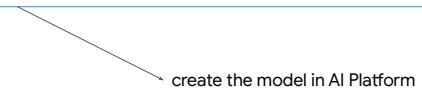
SavedModel is the universal serialization format for TensorFlow models.

SavedModel provides a language-neutral format to save machine-learned models that is recoverable and hermetic. It enables higher-level systems and tools to produce, consume and transform TensorFlow models.

The resulting SavedModel is then servable. Models saved in this format can be restored using `tf.keras.models.load_model` and are compatible with TensorFlow Serving.

Create a model object in Cloud AI Platform

```
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

...
```



One of the ways we can serve the model is to utilize the Cloud AI Platform managed service. The AI Platform service also performs scaled training, but for now we are focusing on serving a trained model.

You start by creating a model object in AI Platform.

Will call our model 'propertyprice'

Create a version of the model in Cloud AI Platform

```
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

Next, we need to create a version for our model

Will call our version 'dnn'. You can also utilize timestamp or another differentiator for multiple versions of the same model type.

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
```


Once model and version are created, you can run this command to push the model to the cloud.
Remember to point to the output directory in which the SavedModel was saved to.
The command to push the model also takes other flags such as python and tensorflow runtime versions, the framework (in case you are using Scikit learn or xgboost - this flag defaults to Tensorflow), and a bucket in which to stage training archives. A staging bucket is required only if a file upload is necessary (that is, other flags include local paths).

Make predictions using `gcloud ai-platform`

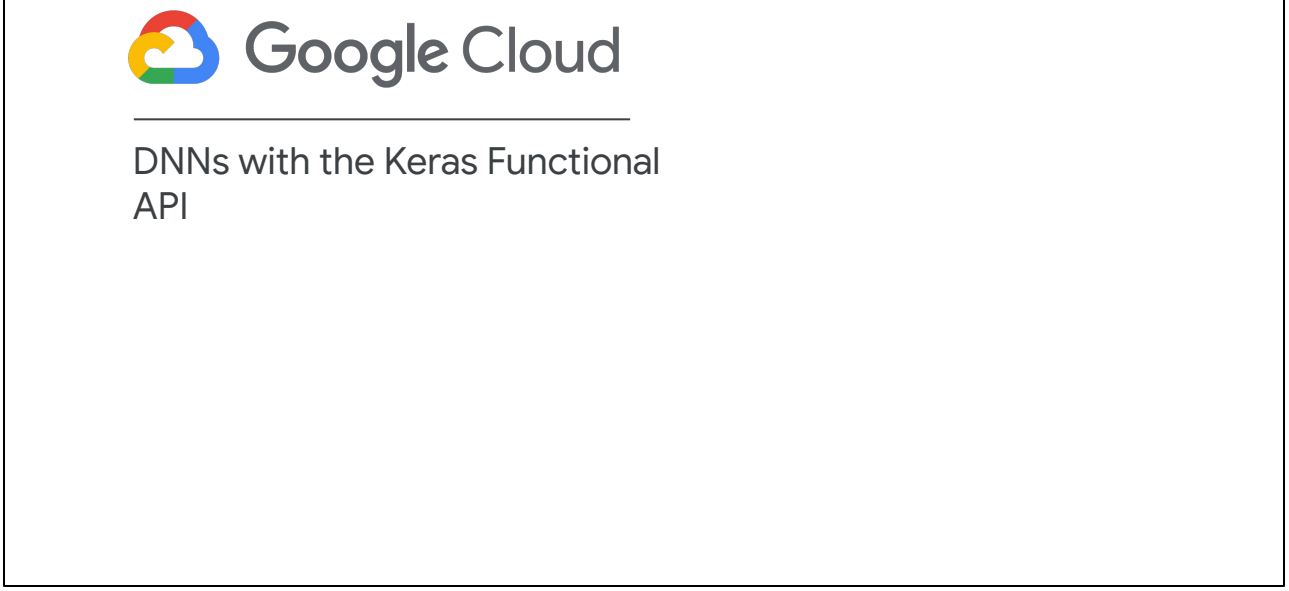
```
input.json = {"sq_footage": 3140,
              "type": 'house'}
```

```
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

Once the model is created and pushed to Ai Platform, use the `gcloud ai-platform predict` command to perform predictions. Make sure that the flags include the model name, its version, and the path to a file containing the examples you want to get predictions on.



Now let's talk about Deep Neural Networks with the Keras Functional API. In this section you'll learn how to create wide and deep models in Keras with just a few lines of tensorflow code.

Seagulls
can fly.

No, this is not a section about ornithology. But we all know that seagulls can fly, right?

Pigeons
can fly.

We also know that pigeons can fly as well.

Animals
with wings
can fly.

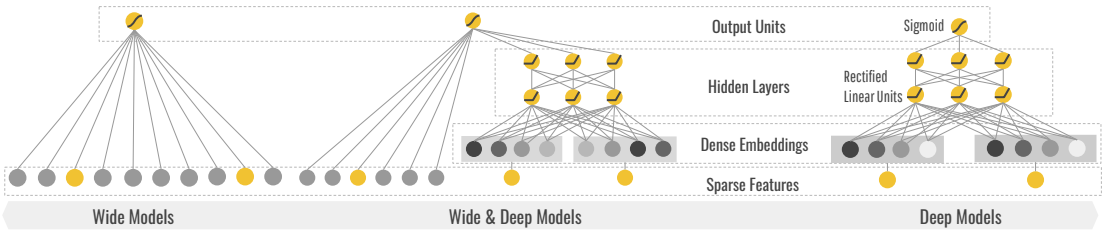
It is intuitive that animals with wings can fly, so making that generalization feels kinda natural.

Penguins...

But what about penguins?

Usings Wide and Deep learning

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



It's not an easy question to answer, but by jointly training a wide linear model (for **memorization**) alongside a deep neural network (for **generalization**), one can combine the strengths of both to bring us one step closer. At Google, we call it Wide & Deep Learning. It's useful for generic large-scale regression and classification problems with sparse inputs (categorical features with a large number of possible feature values), such as recommender systems, search, and ranking problems.

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.”

The human brain is a sophisticated learning machine, forming rules by memorizing everyday events (“seagulls can fly” and “pigeons can fly”) and generalizing those learnings to apply to things we haven't seen before (“animals with wings can fly”). Perhaps more powerfully, memorization also allows us to further refine our generalized rules with exceptions (“penguins can't fly”). As we were exploring how to advance machine intelligence, we asked ourselves the question—can we teach computers to learn like humans do, by combining the power of memorization and generalization?

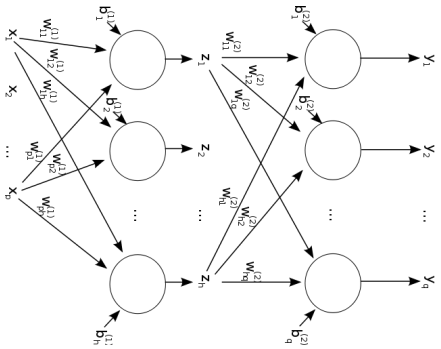
Linear models are good for sparse, independent features

```
[ 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.]
```

This is what a sparse matrix looks like -- very, very wide, with lots and lots of features. You want to use linear models to minimize the number of free parameters. And if the columns are independent, linear models may suffice.

DNNs are good for dense, highly correlated features

pixel_values ()

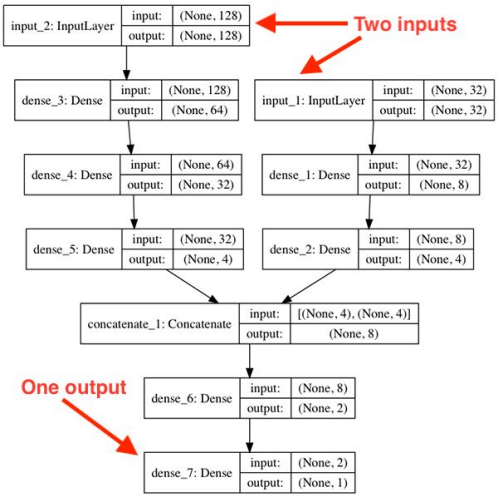


1024^2 input nodes

10 hidden nodes -> 10 image features

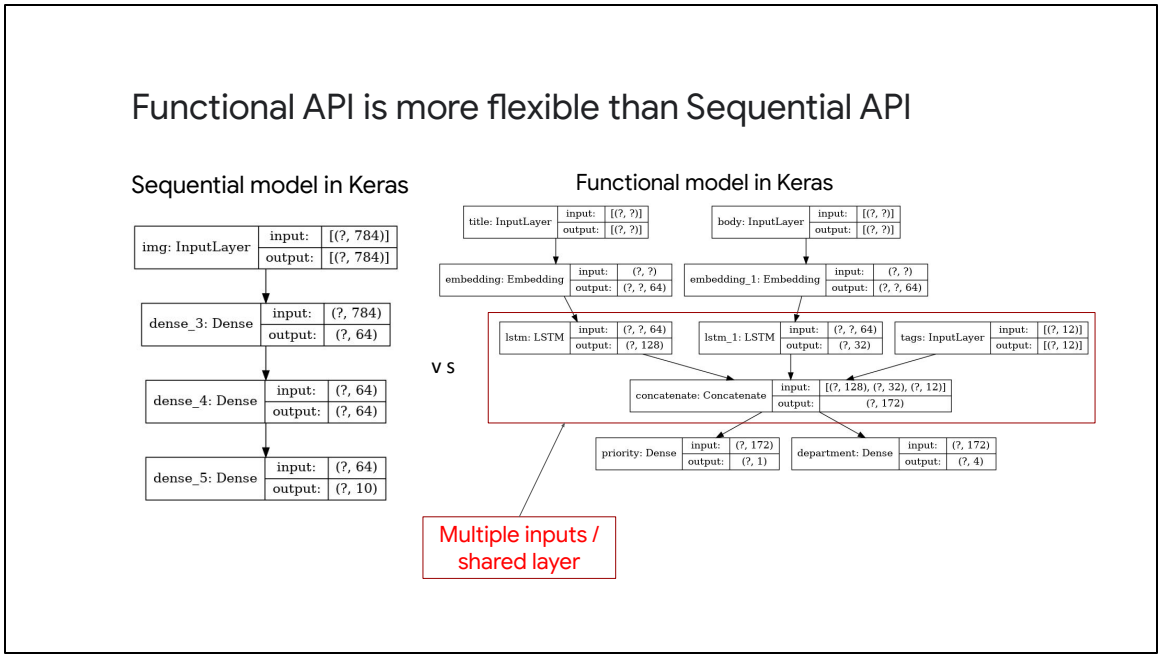
Nearby pixels, however, tend to be highly correlated, so putting them through a NN, we have the possibility that the inputs get decorrelated and mapped to a lower dimension (intuitively, this is what happens when your input layer takes each pixel value, and the number of hidden nodes is much less than the number of input nodes).

Wide and deep networks using Keras Functional API



A Wide and Deep model architecture is an example of a complex model that can be built using the Keras Functional API.
The Functional API gives the model the ability to have multiple inputs and outputs. It also allows for models to share layers. Actually, more than that, it allows you to define ad hoc acyclic network graphs.

With the functional API, models are defined by creating instances of layers and connecting them directly to each other in pairs, then defining a Model that specifies the layers to act as the input and output to the model.



The Functional API is a way to create models that is more flexible than `Sequential`: it can handle models with non-linear topology, models with shared layers, and models with multiple inputs or outputs.

The functional API makes it easy to manipulate multiple inputs and outputs. This cannot be handled with the Sequential API.

Here's a simple example.

Let's say you're building a system for ranking custom issue tickets by priority and routing them to the right department.

You model will have four inputs:

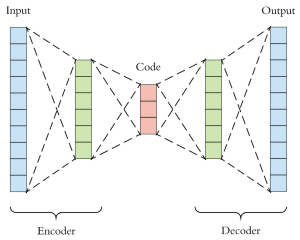
- Title of the ticket (text input)
- Text body of the ticket (text input)
- Any tags added by the user (categorical input)
- An image representing different logos that can appear on the ticket

It will have two outputs:

- The department that should handle the ticket (softmax output over the set of departments)
- A text sequence with a summary of the text body

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')
```



In the functional API, models are created by specifying their inputs and outputs in a graph of layers. That means that a single graph of layers can be used to generate multiple models.

You can treat any model as if it were a layer, by calling it on an input or on the output of another layer. Note that by calling a model you aren't just reusing the architecture of the model, you're also reusing its weights.

This is an example of what code for an autoencoder might look like. Notice how the operations are treated like functions with the outputs serving as inputs for subsequent layers

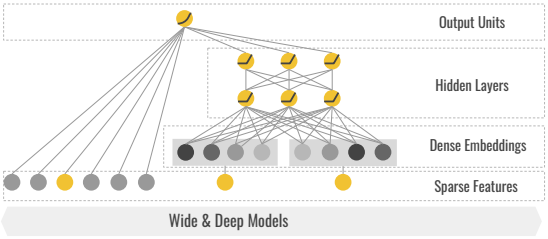
Another good use for the functional API are models that use shared layers. Shared layers are layer instances that get reused multiple times in a same model: they learn features that correspond to multiple paths in the graph-of-layers.

Shared layers are often used to encode inputs that come from similar spaces (say, two different pieces of text that feature similar vocabulary), since they enable sharing of information across these different inputs, and they make it possible to train such a model on less data. If a given word is seen in one of the inputs, that will benefit the processing of all inputs that go through the shared layer.

To share a layer in the Functional API, just call the same layer instance multiple times.

Creating a Wide and Deep model in Keras

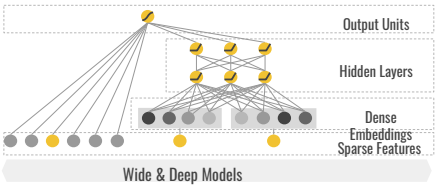
```
INPUT_COLS = [
    'pickup_longitude',
    'pickup_latitude',
    'dropoff_longitude',
    'dropoff_latitude',
    'passenger_count'
]

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

...
```



To create a wide and deep model in Keras, start by setting up the input layers for the model, using the features of the model data. For this example, we are using pickup and dropoff latitude and longitude, as well as the number of passengers, to try and predict the taxi fare for a ride.
These inputs will be fed to the wide and deep portions of the model.

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)
```



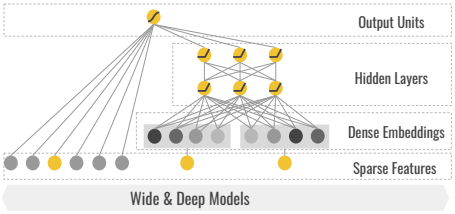
created in the previous slide

Using the inputs above, we can then create the deep portion of the model. layers.Dense is a densely-connected NN layer. By stacking multiple layers, we make it deep.

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)
```

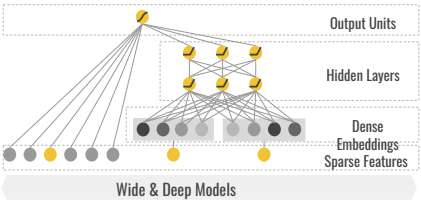
created in the previous slide



we can also create the wide portion of the model using, for example, DenseFeatures, which produces a dense Tensor based on given feature_columns.

Creating a Wide and Deep model in Keras

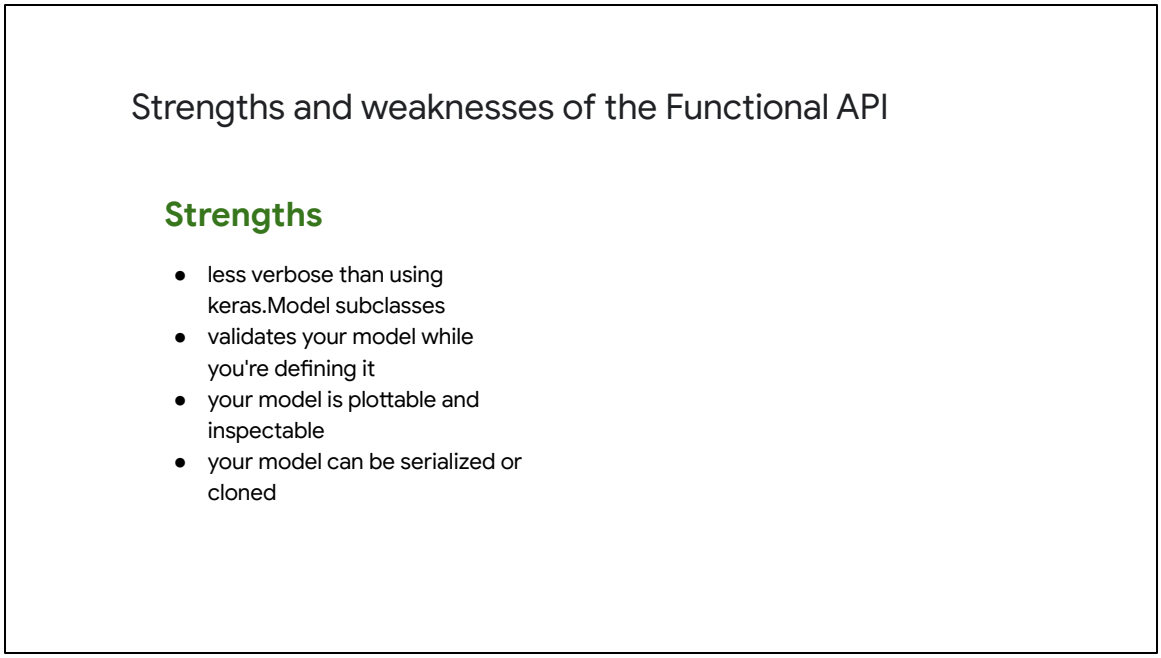
```
# Combine outputs
combined = concatenate(inputs=[deep, wide],
                       name='combined')
output = layers.Dense(1,
                      activation=None,
                      name='prediction')(combined)

# 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

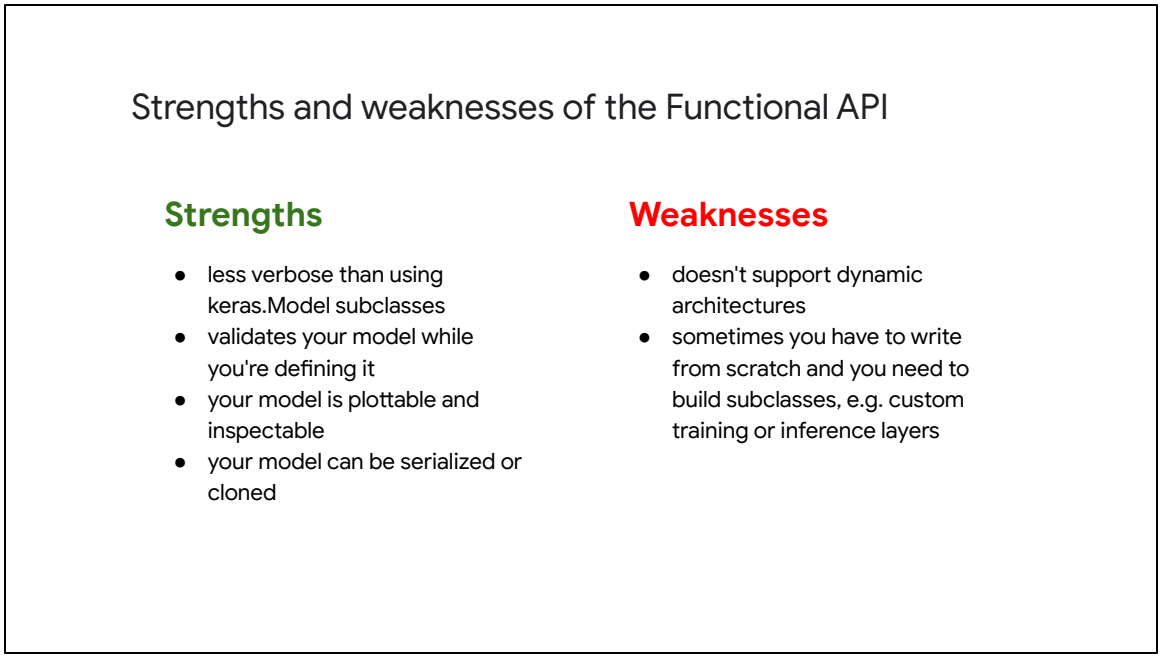
Lastly we combine the wide and deep portions and compile the model.

Training, evaluation, and inference work exactly in the same way for models built using the Functional API as for Sequential models.



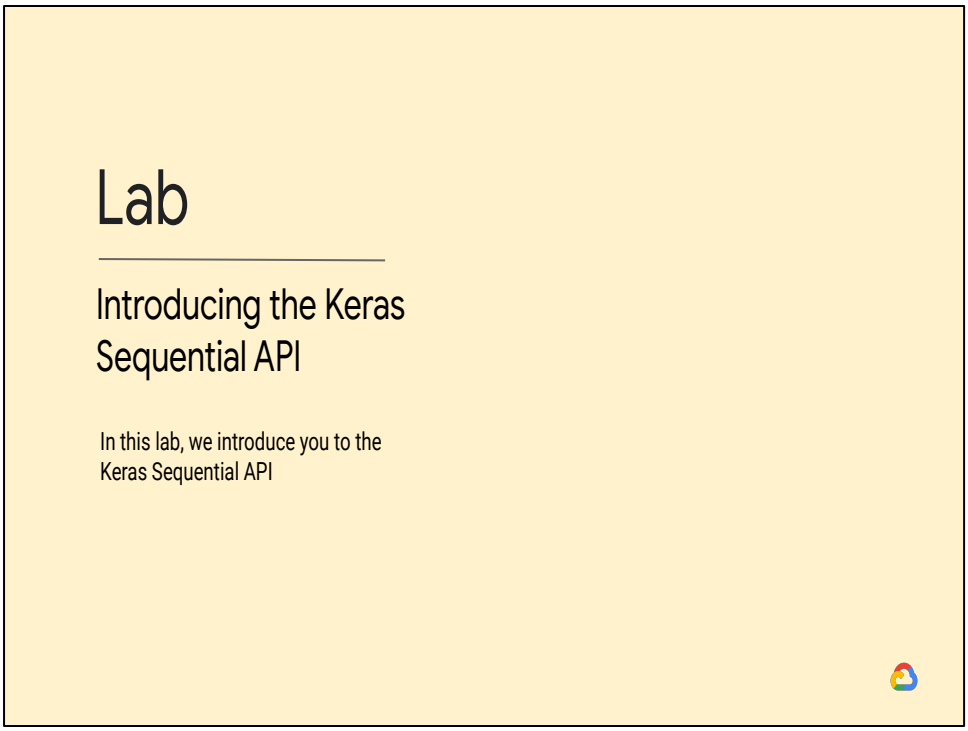
Strengths

- It is less verbose than using keras.Model subclasses
- It validates your model while you're defining it. In the Functional API, your input specification (shape and dtype) is created in advance (via `Input`), and every time you call a layer, the layer checks that the specification passed to it matches its assumptions, and it will raise a helpful error message if not. This guarantees that any model you can build with the Functional API will run. All debugging (other than convergence-related debugging) will happen statically during the model construction, and not at execution time. This is similar to typechecking in a compiler.
- Your Functional model is plottable and inspectable.
- You can plot the model as a graph, and you can easily access intermediate nodes in this graph -- for instance, to extract and reuse the activations of intermediate layers. Your Functional model can be serialized or cloned. Because a Functional model is a data structure rather than a piece of code, it is safely serializable and can be saved as a single file that allows you to recreate the exact same model without having access to any of the original code. See our saving and serialization guide for more details.



Weaknesses

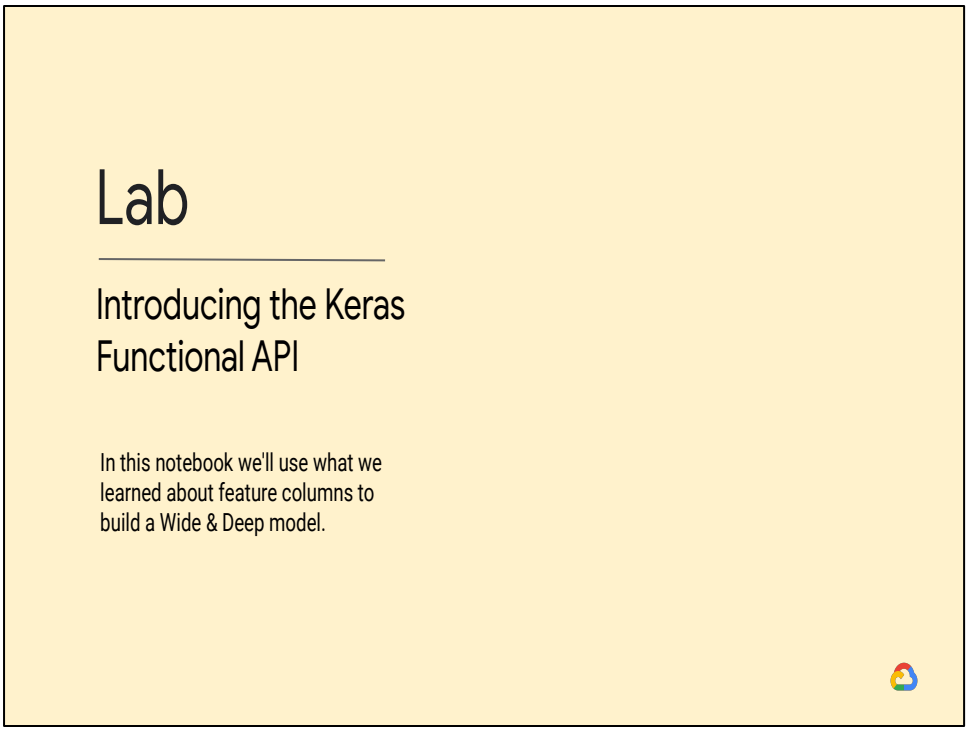
- It does not support dynamic architectures. The Functional API treats models as DAGs of layers. This is true for most deep learning architectures, but not all: for instance, recursive networks or Tree RNNs do not follow this assumption and cannot be implemented in the Functional API.
- Sometimes, you just need to write everything from scratch. When writing advanced architectures, you may want to do things that are outside the scope of "defining a DAG of layers": for instance, you may want to expose multiple custom training and inference methods on your model instance. This requires subclassing.



The Keras sequential API allows you to create Tensorflow models layer-by-layer. This is useful for building most kinds of machine learning models but it does not allow you to create models that share layers, re-use layers or have multiple inputs or outputs.

Link to lab:
https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/machine_learning/deepdive2/introduction_to_tensorflow/solutions/3_keras_sequential_api.ipynb

1. [ML on GCP C3] Introducing the Keras Sequential API (qwiklabs). Github repo link here. CBL123



In the last notebook, we learned about the Keras Sequential API. The Keras Functional API provides an alternate way of building models which is more flexible. With the Functional API, we can build models with more complex topologies, multiple input or output layers, shared layers or non-sequential data flows (e.g. residual layers).

Link to lab: [ML on GCP C3] Introducing the Keras Functional API (qwiklabs) github repo link here. CBL124

Agenda

Activation Functions

Neural Networks with TF 2 and Keras

Regularization

Remember our goal while training a model is to minimize the loss value? It's now time to talk about how to do that at scale with Regularization

What is happening here? How can we address this?

Remember our goal while training a model is to minimize the loss value. If you graphed the loss curve both on training and test data, it may look something like this. The graph shows Loss on the y axis vs. Time on the x axis.

Notice anything wrong here?

Yeah, the loss value is nicely trending down on the training data but shoots upwards at some point on the test data. That cannot be good!

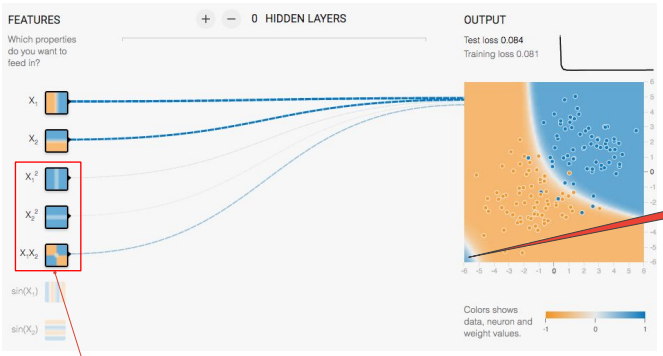
Clearly, some amount of overfitting is going on here. Seems to be correlated with the number of training iterations.

How could we address this?

We could reduce number of training iterations and stop earlier. Early stopping is definitely an option, but there must be better ones...

Here is where Regularization comes into the picture!

Remember the splotch of blue?
Why does it happen?



https://goo.gl/ofiHCT

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

Let's tickle our intuition using Tensorflow playground.

You must have seen and used this playground in previous courses, but to quickly remind you:

TensorFlow Playground is a handy little tool for visualizing how neural networks learn.

We extensively use it throughout this specialization to intuitively grasp the concepts.

Let me draw your attention to the screen. There is something odd going on here.

Notice this region in the bottom left that's hinting towards blue?

There is nothing in the data suggesting blue.

The model's decision boundary is kind of crazy! Why do you think that is?

Notice the relative thickness of the five lines running from INPUT to OUTPUT.

These lines show the relative weights of the five features.

The lines emanating from X1 and X2 are much thicker than those coming from the feature crosses.

So, the feature crosses are contributing far less to the model than the normal (uncrossed) features.

Removing all the feature crosses gives a saner model.

You should try this for yourself and see how curved boundary suggestive of overfitting disappears and test loss converges.

After 1,000 iterations, test loss should be a slightly lower value than when the feature crosses were in play. Although your results may vary a bit, depending on the data set.

The data in this exercise is basically linear data plus noise.

If we use a model that is too complicated, such as one with too many crosses, we give it the opportunity to fit to the noise in the training data, often at the cost of making the model perform badly on test data.

Clearly, early stopping cannot help us here.

It’s the model complexity that we need to bring under control.

But, how could we measure model complexity and avoid it?

The simpler the better

Don’t cook with every spice in the spice rack!

Image Source: https://pixabay.com/en/spice-rack-cooking-spices-1650049/ (cc0)

We concluded that simpler models are usually better; we don’t want to cook with every spice in the spice rack!

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).

source:https://en.wikipedia.org/wiki/Occam%27s_razor

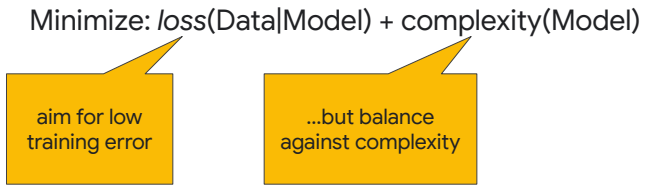
There is a whole field around this called Generalization Theory or G Theory that goes about defining the statistical framework.

The easiest way to think about it, though, is by intuition, based on 14th century principle laid out by William Ockham.

While training a model we will apply Okham's razor principle as our heuristic guide in favoring simpler models with less assumptions about the training data.

Let's look into some of the most common Regularization techniques that can help us apply this principle in practice.

Factor in model complexity when calculating error



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

The idea is to penalize model complexity.

So far in our training process, we have been trying to minimize loss of the data, given the model; we need to balance that against complexity of the model.

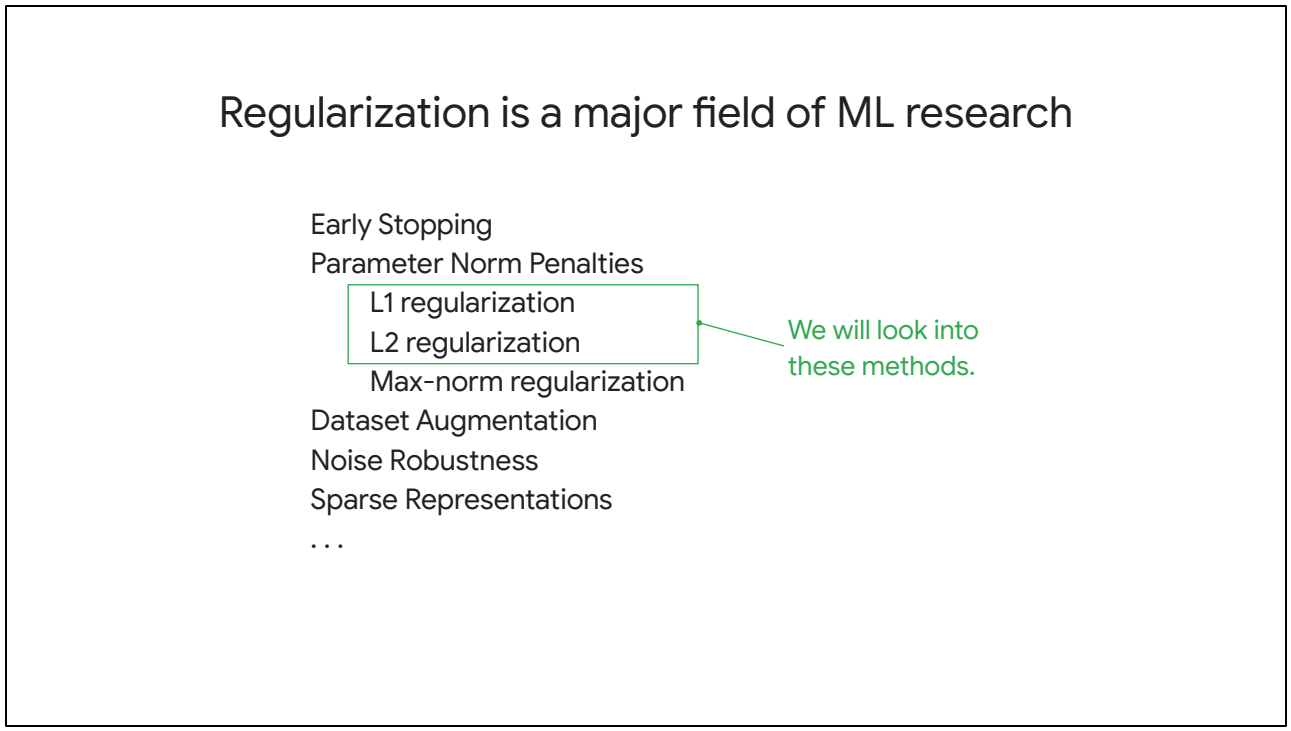
Before we talk about how to measure model complexity, let's pause and understand why we said "balance" complexity against loss.

The truth is that oversimplified models are useless! If we take it to extreme, we will end up with a null model. We need to find the right balance between simplicity and accurate fitting of training data.

Later you will see that the complexity measure is multiplied by a lambda coefficient, which will allow us to control our emphasis on model simplicity. This makes up yet another hyperparameter that requires tuning.

Optimal lambda value for any given problem is data dependent, which means we almost always need to spend some time tuning this (either manually or via automated search). I told you ML needs some artful skills!

I hope by now it is clear why this approach is arguably more principled than early stopping.



Regularization is one of the major fields of research within Machine Learning.

There are many published techniques and more to come:

- We already mentioned “Early Stopping”
- We also started exploring the group of methods under the umbrella of “Parameter Norm Penalties”.
- There is also “Dataset Augmentation” methods, “Noise Robustness”, “Sparse Representations” and many more.

In this module, we will have a closer look at L1 and L2 regularization methods from “Parameter Norm Penalties” group of techniques.

But, before we do that, let’s quickly remind ourselves what problem “Regularization” is solving for us:

Regularization refers to any technique that helps generalize a model.

A generalized model performs well not just on training data, but also on never-seen test data.

How can we measure model complexity?

We know we are going to use regularization methods that penalize model complexity.

Now, the question is how to measure model complexity?

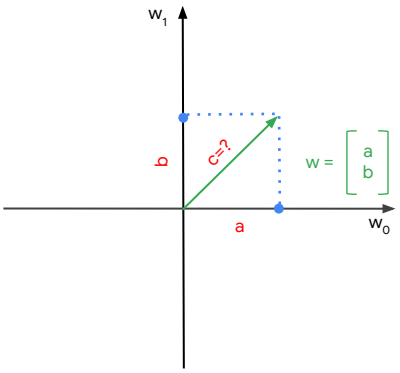
Both L1 and L2 regularization methods represent model complexity as the magnitude of the weight vector and try to keep that in check.

From Linear Algebra, you should remember that the magnitude of a vector is represented by the Norm function.

Let’s quickly review L1 and L2 Norm functions.

L2 vs. L1 Norm

$$w = [w_0, w_1, \ldots, w_n]^T$$



The weight vercor can be of any number of dimensions, but it's easier to visualize in 2-dimensional space.

So, a vector with w0=a and w1=b would look like this green arrow.

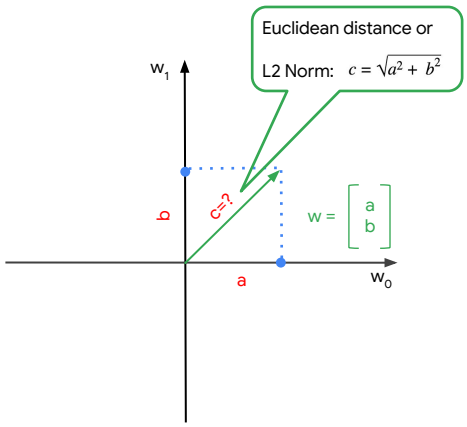
Now, what's the magnitude of this vector?

L2 vs. L1 Norm

$$w = [w_0, w_1, \ldots, w_n]^T$$

$$\|w\|_2 = (w_0^2 + w_1^2 + \ldots + w_n^2)^{1/2}$$

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



You may instantly think c?

Because you are applying the most common way that we learnt in high school: the Euclidean distance from the origin.

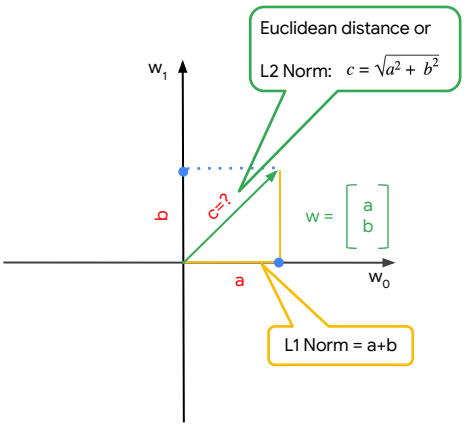
c would be the square root of sum of **a**-squared plus **b**-squared.

L2 vs. L1 Norm

$$w = [w_0, w_1, \ldots, w_n]^T$$

L2 Norm

$$\|w\|_2 = (w_0^2 + w_1^2 + \ldots + w_n^2)^{1/2}$$



In Linear Algebra, this is called the L2 norm : denoted by the double bars and the subscript of 2, or no subscript at all, because 2 is the known default.

The L2 norm is calculated as the square root of sum of the squared values of all vector components.

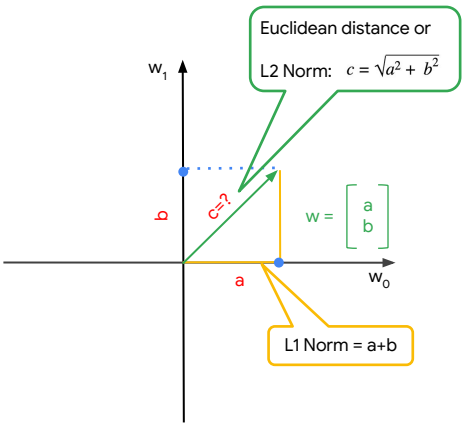
But that's not the only way the magnitude of a vector can be calculated.

L2 vs. L1 Norm

$$w = [w_0, w_1, \ldots, w_n]^T$$

L2 Norm $\|w\|_2 = (w_0^2 + w_1^2 + \ldots + w_n^2)^{1/2}$

L1 Norm $\|w\|_1 = (|w_0| + |w_1| + \ldots + |w_n|)$



Another common method is L1 norm.

L1 measures absolute value of **a** plus absolute value of **b**. Basically, the yellow path highlighted here.

Now, remember we were looking for a way to define model complexity.

We used L1 and L2 as regularization methods where model complexity is measured in the form of magnitude of the weight vector.

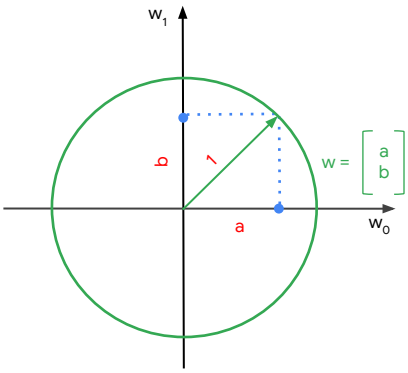
In other words, if we keep the magnitude of our weight vector smaller than certain value, we've achieved our goal.

L2 vs. L1 Norm

$$w = [w_0, w_1, \ldots, w_n]^T$$

L2 Norm

$$\|w\|_2 = (w_0^2 + w_1^2 + \ldots + w_n^2)^{1/2}$$



Now let's visualize what it means for the L2 norm of our weight vector to be under certain value, let's say 1.

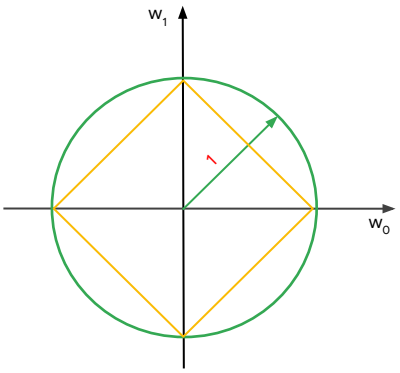
Since L2 is the Euclidean distance from the origin, our desired vector would be bound within this circle with a radius of 1 centered on the origin.

L2 vs. L1 Norm

$$w = [w_0, w_1, \ldots, w_n]^T$$

L2 Norm $\|w\|_2 = (w_0^2 + w_1^2 + \ldots + w_n^2)^{1/2}$

L1 Norm $\|w\|_1 = (|w_0| + |w_1| + \ldots + |w_n|)$



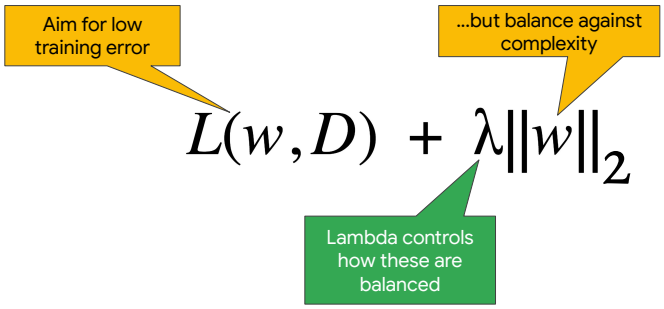
When trying to keep L1 norm under certain value, the area in which our weight vector can reside will take the shape of this yellow diamond.

The most important takeaway here is that when applying L1 regularization, the optimal value of certain weights can end up being zero.

And that is because of the extreme diamond shape of the optimal region that we are interested in.

That is as opposed to the smooth circular shape in L2 regularization.

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



Let's go back to the problem at hand: how to regularize our model using vector norm.

This is how we apply L2 regularization, also known as weight decay.

Remember we try to keep the weight values close to the origin. In 2D space the weight vector would be confined within a circle, you can easily expand the concept to 3D space, but beyond 3D is hard to visualize, don't try!

To be perfectly honest, in machine learning we cheat a little in the math department. We use the square of the L_2 norm to simplify calculation of derivatives.

Notice there's a new parameter here: lambda. This is a simple scalar value that allows us to control how much emphasis we want to put on model simplicity over minimizing training error.

It is another tuning parameter which must be explicitly set. Unfortunately, the best value for any given problem is data dependent. So, we'll need to do some tuning, either manually or automatically using a tool like hyperparameter tuning (which we will cover in the next module).

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

$$L(w, D) + \lambda \|w\|_1$$

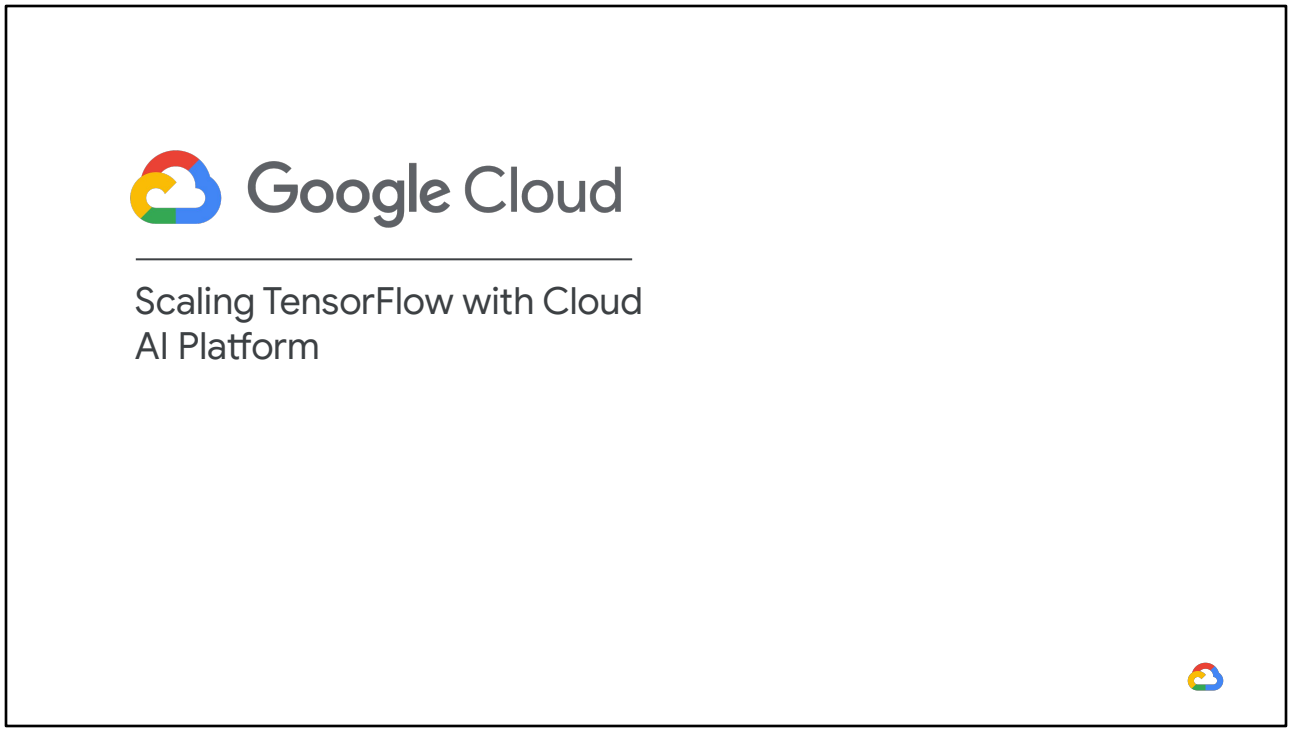
L1 regularization can be used as a feature selection mechanism.

To apply L1 regularization, we simply swap L2 norm with L1 norm. Careful though, the outcome could be very different!

L1 regularization results in a solution that is more sparse. Sparsity in this context refers to the fact that some of the weights end up having an optimal value of zero. Remember the diamond shape of the optimal area?

This property of L1 regularization is extensively used as a feature selection mechanism.

Feature selection simplifies the ML problem by causing a subset of the weights to become zero. Zero weights then highlight the subset of features that can be safely discarded.



Welcome to Scaling Tensorflow with Cloud AI Platform.

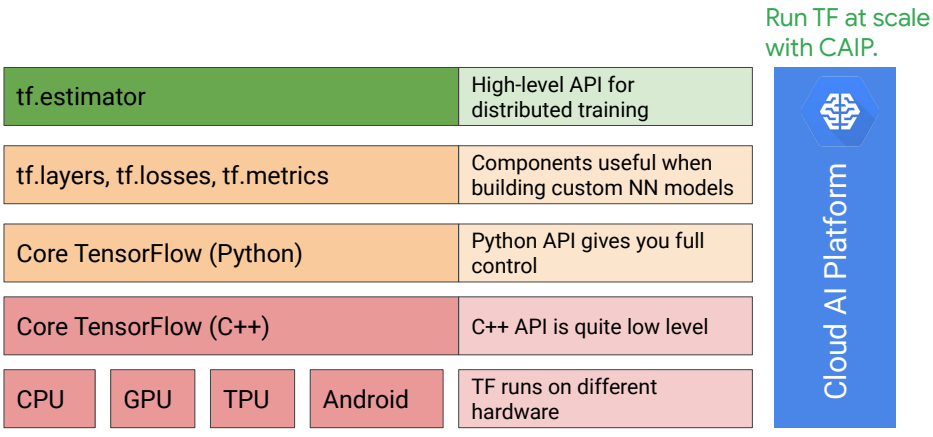
Agenda

Why Cloud AI Platform

Train a model

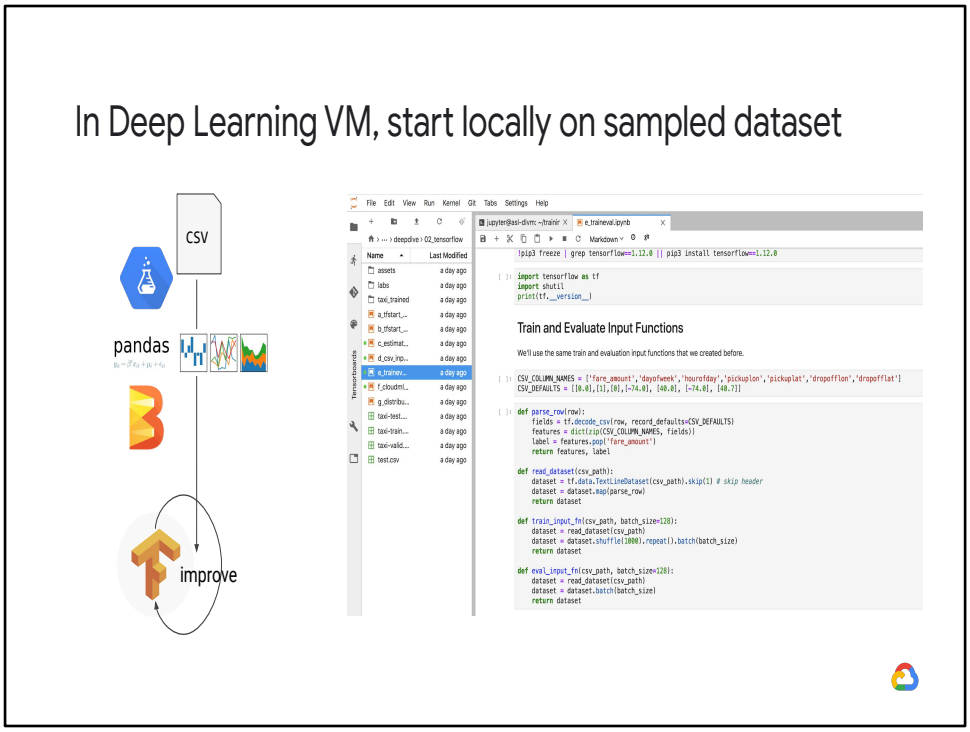
Deploy a trained model and make predictions

We will use distributed TensorFlow on Cloud AI Platform



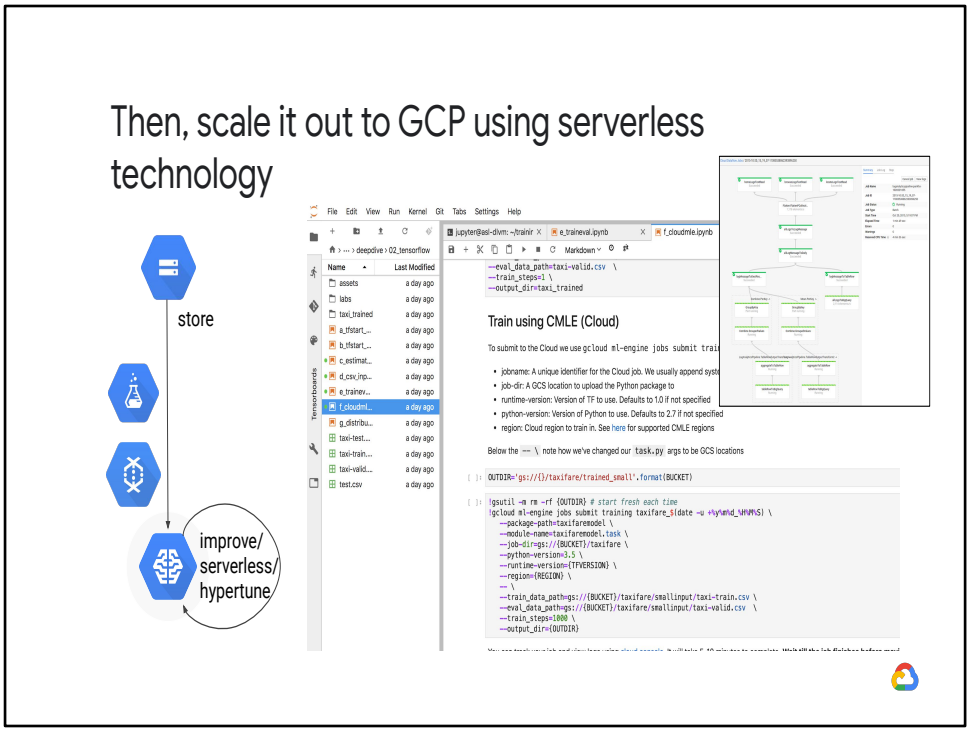
Now this diagram you've already seen before.
Recall that TensorFlow can run on different hardware,
You could program it in the low-level C++ API,
But more likely you'll use the Python API as we practice in this course,
And you've already started to see the different abstraction layers for distributed training

But do you actually run distributed TF at scale in production? For that let's introduce AI Platform, formerly known as Cloud ML Engine.



Notebooks, like Google Cloud's Datalab or Kaggle Kernels, are a great way to get started and iterate quickly while developing your model. Notebooks let you interactively explore the data, define and probe new features--even launch training and eval jobs. The interface combines code, results, docs all into a human readable format.

And since you're on Cloud you have great sharing & collaboration support and a rich set of tutorials



When you initiate a Cloud job from a deep learning VM (whether it is a training job or data preprocessing with Dataflow), the job is automatically distributed across as many VMs as needed.

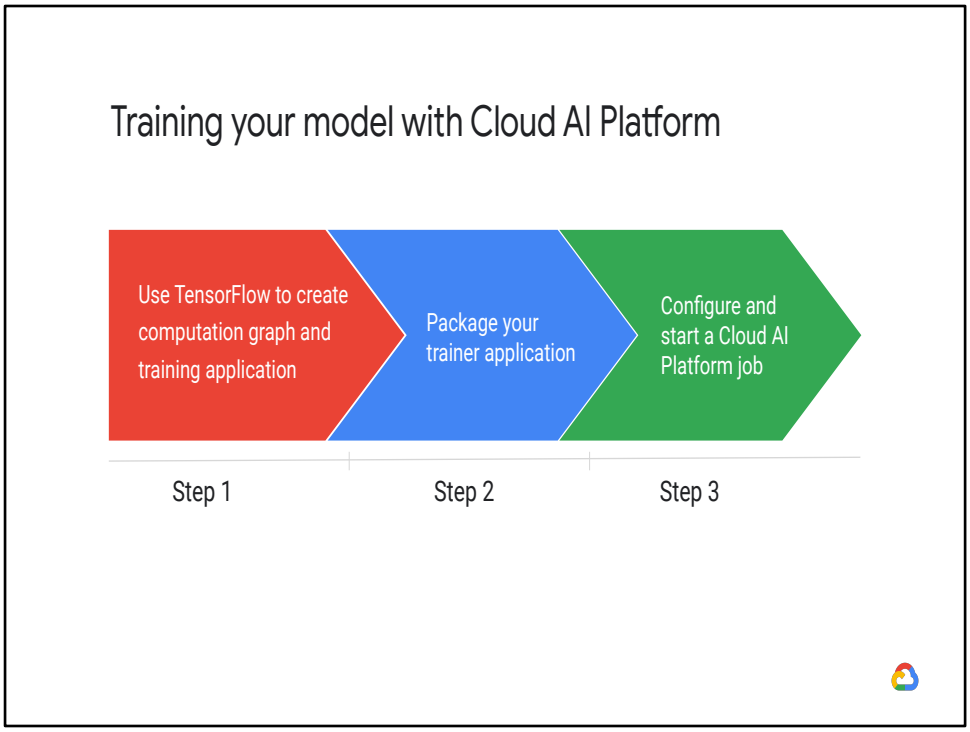
Agenda

Why Cloud AI Platform

Train a model

Deploy a trained model and make predictions

Now let’s look at how training a model works with Cloud AI Platform.



Before you begin training though, be sure to (1) gather and prepare (clean, split, engineer features, preprocess features) your training data, and (2) put that training data in an online source that Cloud AI Platform can access (e.g. Cloud Storage).

An example of a model.py file

model.py

```
def train_and_evaluate(args):
    model = build_wide_deep_model(args["nnsize"], args["nembeds"])
    print("Here is our Wide-and-Deep architecture so far:\n")
    print(model.summary())

    trainds = load_dataset(
        args["train_data_path"],
        args["batch_size"],
        tf.estimator.ModeKeys.TRAIN)

    evalds = load_dataset(
        args["eval_data_path"], 1000, tf.estimator.ModeKeys.EVAL)
    if args["eval_steps"]:
        evalds = evalds.take(count=args["eval_steps"])

    num_batches = args["batch_size"] * args["num_epochs"]
    steps_per_epoch = args["train_examples"] // num_batches

    checkpoint_path = os.path.join(args["output_dir"], "checkpoints/babyweight")
    cp_callback = tf.keras.callbacks.ModelCheckpoint(
        filepath=checkpoint_path, verbose=1, save_weights_only=True)
```

When sending training jobs to Cloud AI Platform, its common to split most of the logic into a task.py file and a model.py file.

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

task.py

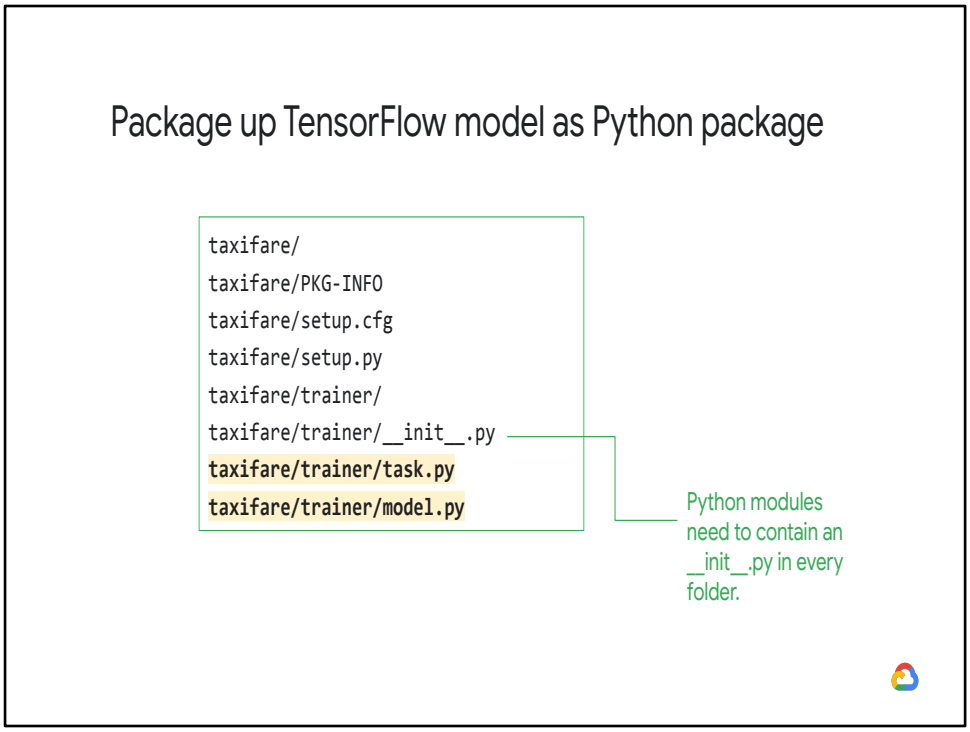
```
parser.add_argument(
    "--eval_data_path",
    help="GCS location of evaluation data",
    required=True
)
parser.add_argument(
    "--output_dir",
    help="GCS location to write checkpoints and export models",
    required=True
)
parser.add_argument(
    "--batch_size",
    help="Number of examples to compute gradient over.",
    type=int,
    default=512
)
parser.add_argument(
    "--nnsize",
    help="Hidden layer sizes for DNN -- provide space-separated layers",
    nargs="+",
    type=int,
    default=[128, 32, 4]
```

Task.py is the entrypoint to your code that CAIP will start and knows job-level details like: how to parse the command line arguments, how long to run, where to write the outputs, how to interface with hyperparameter tuning and so on. To do the core ML, task.py will invoke model.py

The `model.py` contains the ML model in TensorFlow

```
# Determine CSV, label, and key columns
CSV_COLUMNS = ["weight_pounds",
               "is_male",
               "mother_age",
               "plurality",
               "gestation_weeks"]
LABEL_COLUMN = "weight_pounds"

# Set default values for each CSV column.
# Treat is_male and plurality as strings.
DEFAULTS = [[0.0], ["null"], [0.0], ["null"], [0.0]]


def features_and_labels(row_data):
    """Splits features and labels from feature dictionary.

    Args:
        row_data: Dictionary of CSV column names and tensor values.
    Returns:
        Dictionary of feature tensors and label tensor.
    """
    label = row_data.pop(LABEL_COLUMN)
```

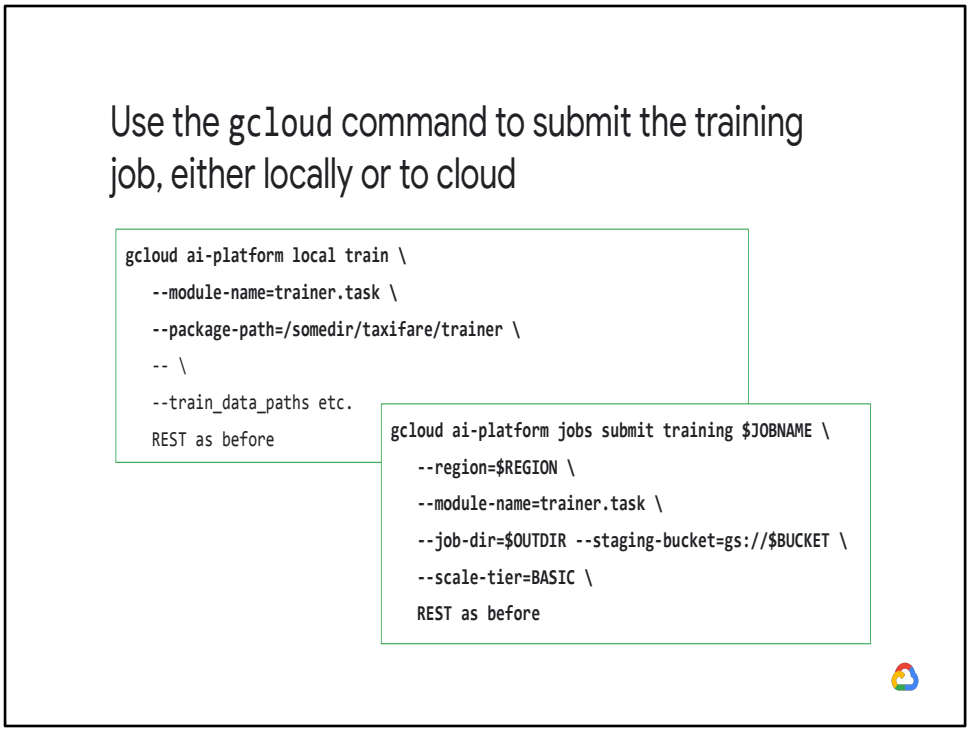
Model.py, however, focuses more on the core ML tasks like: fetching the data, defining features, configure the service signature and of course the actual train and eval loop.



Sharing code between computers always involves some type of packaging. Sending your model to CAIP for training is no different.

Tensorflow, and Python in particular, require a very specific, but standardized, packaging structure shown here.

Check out more about that here:
http://python-packaging.readthedocs.io/en/latest/minimal.html

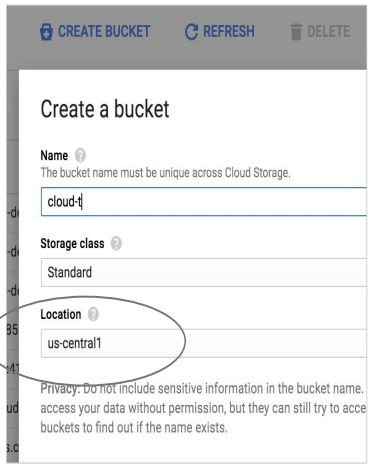


Next let's use gcloud to locally test our code. This will do some quick sanity checks that your package structure is correct. Once satisfied, you can submit a training job to send the task to Cloud to scale out!

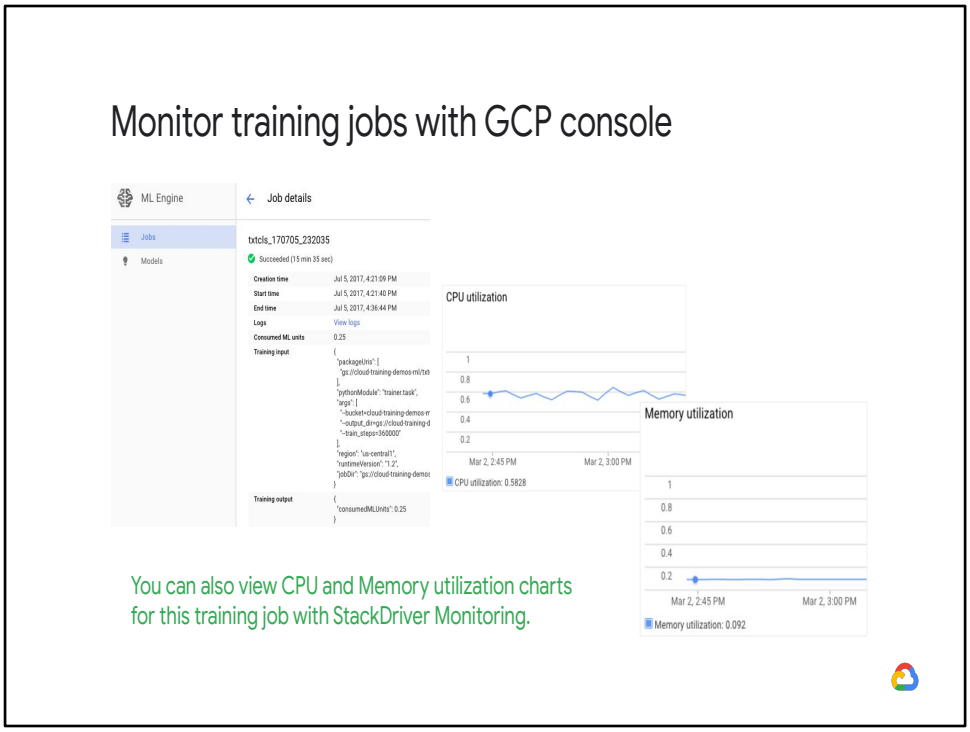
The key command line args here are "package path" to specify where the code is located, the "module name" to specify which of the files in the package to execute and a "scale tier" to specify what kind of hardware you want the code executed on.

You'd specify scale-tier=BASIC to run on one machine
scale-tier=STANDARD to run on a smallish cluster
scale-tier=BASIC_GPU to run on a single GPU
To run on a TPU? You guessed it; scale-tier=BASIC_TPU
You can also specify custom tiers and define each machine type
The scale tiers keep expanding. Look up the Cloud AI Platform documentation for all your current options.

Tip: Use single-region bucket for ML

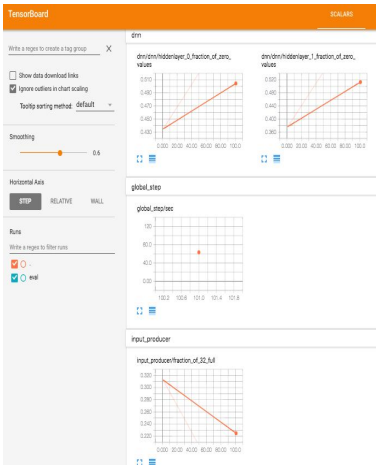


Just a pro-tip here. To get the best performance for ML jobs, make sure you select a single-region bucket in Google Cloud storage. The default is multi-region which is better suited for web serving than ML training!



Lets take a few minutes to discuss monitoring our jobs now. The GCP web console has a great UI for monitoring your jobs. You can see exactly how they were invoked, check out their logs and see how much CPU and memory they are consuming.

Monitor training jobs with TensorBoard



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

While inspecting log entries may help you debug technical issues like an exception, it's really not the right tool to investigate the ML performance. Tensorboard however is a great tool. To use it, make sure your job saves summary data to a Google Cloud Storage location, and then when you start Tensorboard simply provide that directory. It can even handle multiple jobs per folder.

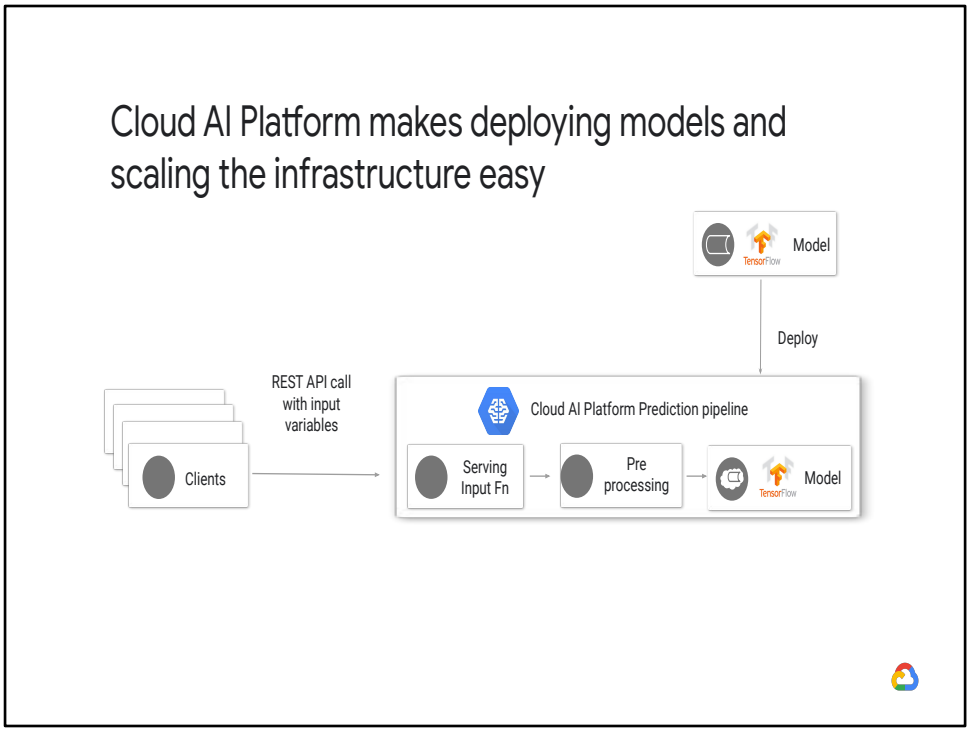
Agenda

Why Cloud AI Platform

Train a model

Deploy a trained model and make predictions

Now that we've got a model, lets see what we can do with it.



Once your training jobs complete, you'll have a TensorFlow model ready to serve for predictions. Cloud AI Platform provides a great infrastructure for this. CAIP will build you a production-ready web app out of your trained model and offer a batch service for your less latency sensitive predictions. Since these are both REST APIs, you'll be able to make scalable, secure inferences from whatever language you want to write the client in.

Deploy the saved model to GCP

```
MODEL_NAME="taxifare"
MODEL_VERSION="v1"
MODEL_LOCATION="gs://${BUCKET}/taxifare/smallinput/taxi_trained/export/Servo/.../"

gcloud ai-platform models create ${MODEL_NAME} --regions $REGION
gcloud ai-platform versions create ${MODEL_VERSION} --model ${MODEL_NAME} --origin
${MODEL_LOCATION} --runtime-version 1.4
```

Could also be a locally-trained model.

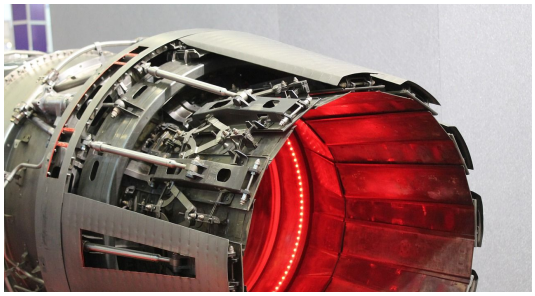
So to send your TF model artifact to Cloud for serving, you need to create a CAIP model and version resource. The individual TF trained model file you have will correspond to a specific ***version***. On CAIP a ***model*** is actually a group of these versions that has a default version as well. This extra layer of abstraction and grouping allows us to seamlessly migrate traffic from one TF model version to the next; just need to change a model's default version!

Client code can make REST calls

```
credentials = GoogleCredentials.get_application_default()
api = discovery.build('ml', 'v1', credentials=credentials)
request_data = [
    {'pickup_longitude': -73.885262,
     'pickup_latitude': 40.773008,
     'dropoff_longitude': -73.987232,
     'dropoff_latitude': 40.732403}]
parent = 'projects/%s/models/%s/versions/%s' % ('cloud-training-demos', 'taxifare', 'v1')
response = api.projects().predict(body={'instances': request_data},
name=parent).execute()
```

Here's a simple example of how to use the remotely deployed model for predictions with a REST call. CAIP Online Prediction is a completely serverless system, so you don't have to worry about any resource allocations; it will just scale for you.

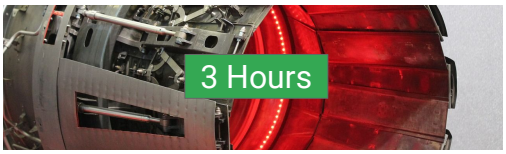
High-performance ML



What does high-performance machine learning mean to you?

Does it mean “powerful”? the ability to handle large datasets?
Doing it as fast as possible?
The ability to train for long periods of time?
Achieving the best possible accuracy?

Model training time

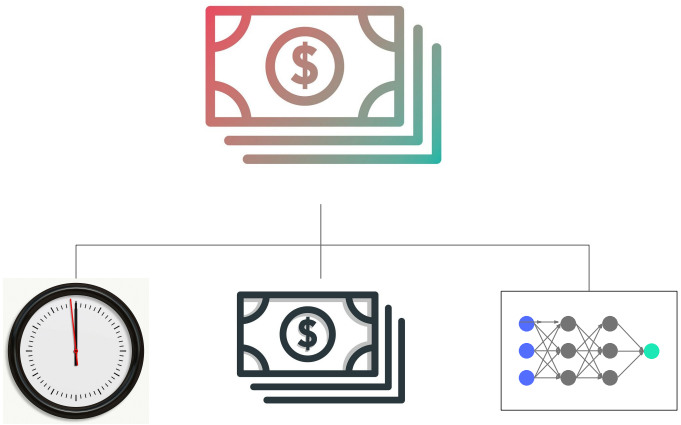


One key aspect is the time taken to train a model.

If it takes 6 hours to train a model on some hardware/software architecture but only 3 hours to train the same model to the same accuracy on some other hardware/software architecture, I think we will all agree that the second architecture is twice as performant as the first one.

Notice that I said “train the model to the same accuracy”. Throughout this module, we will assume that we are talking of models that have the same accuracy or RMSE or whatever your evaluation measure is. Obviously, when we talk about high-performance ML models, accuracy is important. We just aren’t going to consider that in this module. The rest of the courses in this specialization will look at how to build more accurate ML models, and there we will be looking at model architectures that will help us get to a desired accuracy. Here, in this course, we will look solely at infrastructure performance.

Optimizing your training budget

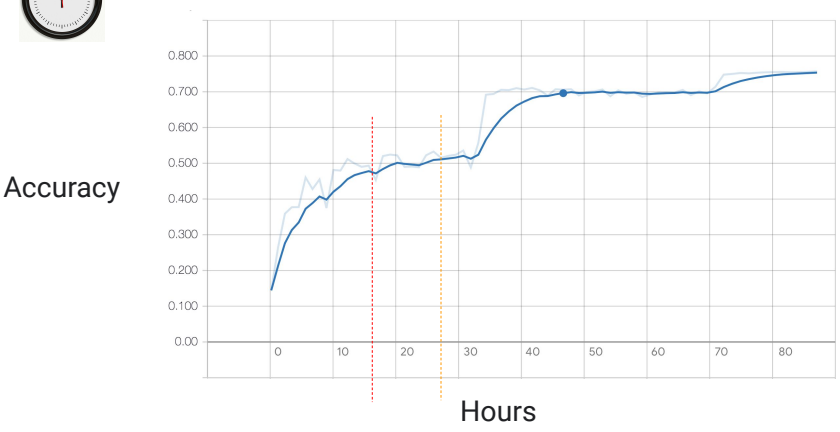


Besides time to train, there is another aspect. Budget.
You often have a training budget. You might be able to train faster on better hardware, but the hardware might cost more, and so you might make the explicit choice to train on slightly slower infrastructure.

When it comes to your training budget, you have three considerations, three levers that you can adjust:

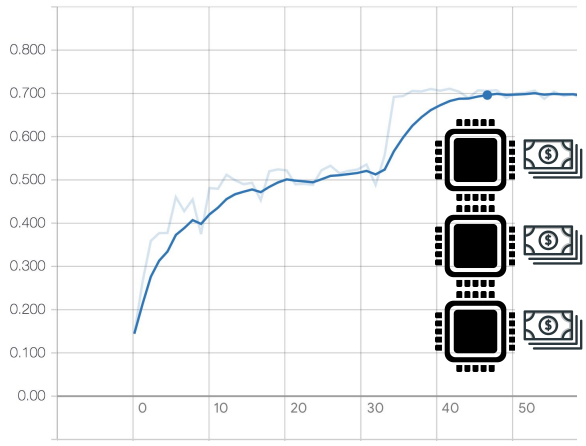
- Time
- Cost
- Scale

Model training can take a long time



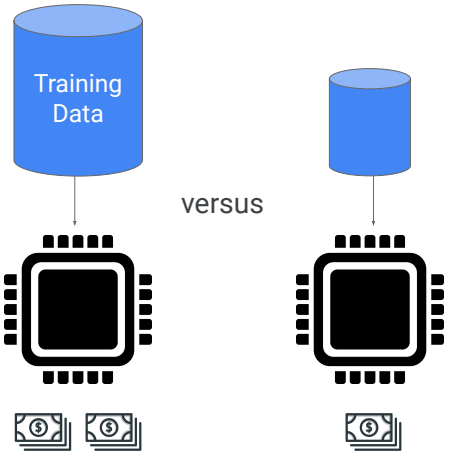
How long are you willing to spend on the model training? This might be driven by the business use case. If you are training a model everyday so as to recommend products to users the next day, then your training has to finish within 24 hours. Realistically, you will need to time to deploy, to A/B test, etc. So, your actual budget might be only 18 hours.

Analyze the benefit of the model versus the running cost



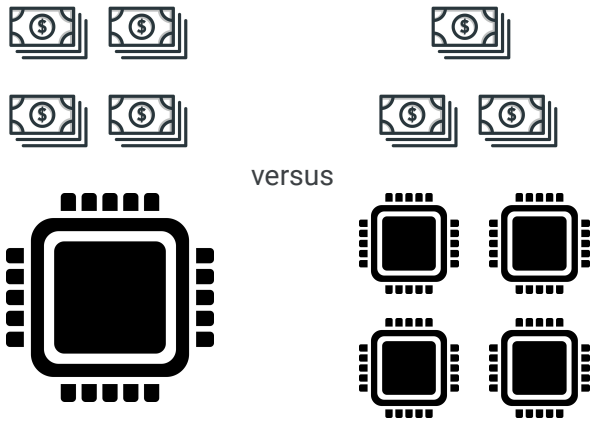
How much are you willing to spend on model training in terms of computing costs? This, too, is a business decision. You don't want to train for 18 hours every day if the incremental benefit is not sufficient.

Optimize the training dataset size



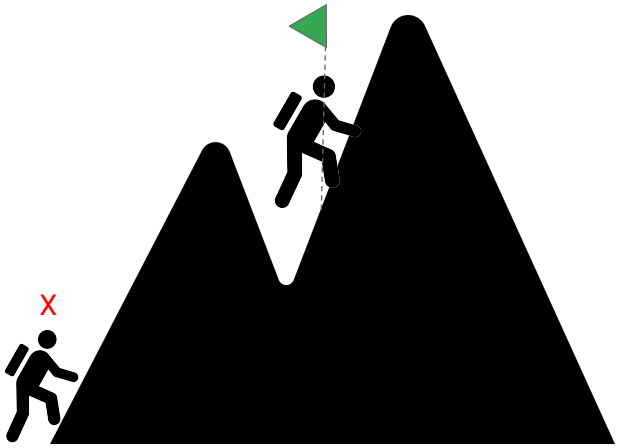
Scale is another aspect of your budget. Models differ in terms of how computationally expensive they are. Even keeping to the same model, you have a choice of how much data you are going to train on -- generally, the more data, the more accurate the model, but there are diminishing returns to larger and larger data sizes, So, your time and cost budget might dictate the data set size.

Choosing optimized infrastructure



Similarly, you often have a choice between training on a single, more expensive machine or multiple, cheaper machines. But to take advantage of this, you may have to write your code somewhat differently. That is another aspect of scale.

Use earlier model checkpoints



Also, you have the choice of starting from an earlier model checkpoint, and training for just a few steps. Typically, this will converge faster than training from scratch each time. This compromise might allow you to reach the desired accuracy faster and cheaper.

Tuning performance to reduce training time, reduce cost, and increase scale

Constraint	Input / Output	CPU	Memory
Commonly occurs	Large inputs. Input requires parsing. Small models.	Expensive computations. Underpowered hardware.	Large number of inputs. Complex model.
Take action	Store efficiently. Parallelize reads. Consider batch size.	Train on faster accel. Upgrade processor. Run on TPUs. Simplify model.	Add more memory. Use fewer layers Reduce batch size.

In addition, there are ways to tune performance to reduce the time, reduce the cost, or increase the scale.
In order to understand what these are, it helps to understand that model training performance will be bound by one of three things:

- **input/output --** how fast can you get data into the model in each training step?
- **Cpu** -- how fast can you compute the gradient in each training step?
- **Memory** -- how many weights can you hold in memory, so that you can do the matrix multiplications in-memory on the GPU or TPU?

Your ML training will be I/O bound if the number of inputs is large, heterogeneous (requires parsing), or if the model is so small that the compute requirements are trivial. This also tends to be the case if the input data is on a storage system with low throughput.
Your ML training will be CPU bound if the I/O is simple, but the model involves lots of expensive computations. You will also encounter this situation if you are running a model on underpowered hardware.
Your ML training might be memory-bound if the number of inputs is large or if the model is complex and has lots of free parameters. You will also face memory limitations if your accelerator doesn't have enough memory.

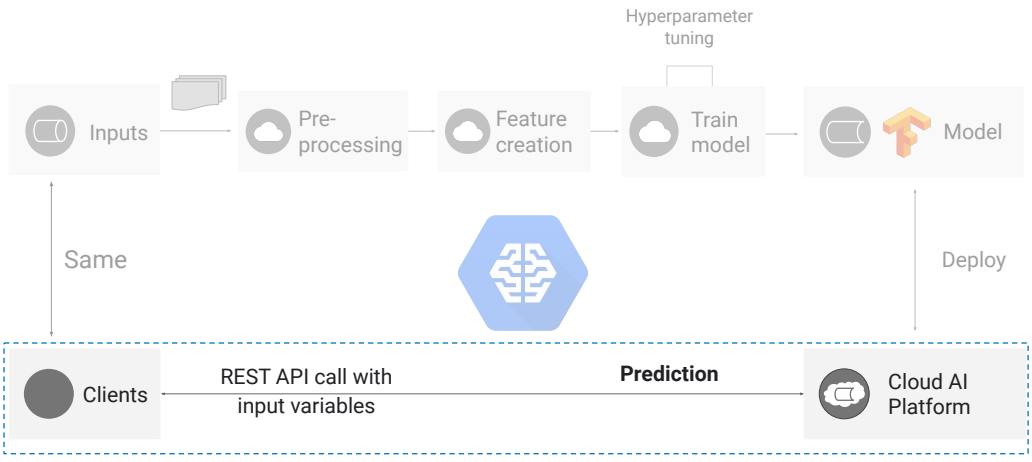
So, knowing what you are bound by, you can look at how to improve performance.
If you are I/O bound, look at storing the data more efficiently, on a storage system with higher throughput, or parallelizing the reads. Although it is not ideal, you might

consider reducing the batch size so that you are reading less data in each step.

If you are CPU-bound, see if you can run the training on a faster accelerator. GPUs keep getting faster, so move to a newer generation processor. And on Google Cloud, you also have the option of running on TPUs. Even if it is not ideal, you might consider using a simpler model, a less computationally expensive activation function or simply just train for fewer steps.

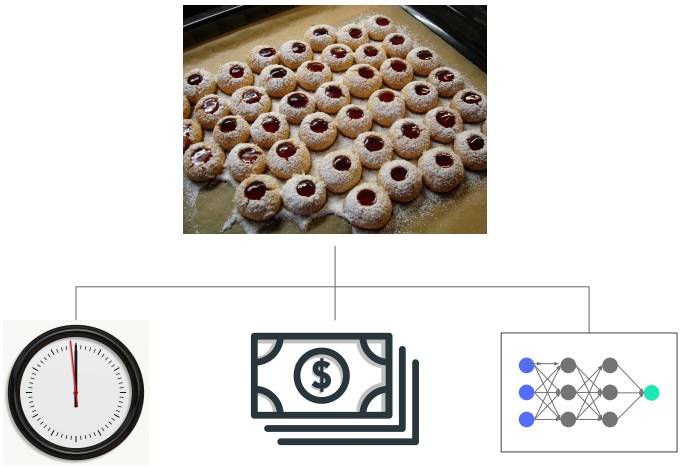
If you are memory-bound, see if you can add more memory to the individual workers. Again, not ideal, but you might consider using fewer layers in your model. Reducing the batch size can also help with memory-bound ML systems.

Performance must consider prediction-time, not just training



We have talked about time to *train*, but there is another aspect to performance. Predictions.
During inference, you have performance considerations as well.

Optimizing your batch prediction



If you are doing batch prediction, the considerations are very similar to that of training. You are concerned with

- Time: How long does it take you to do all the predictions? This might be driven by a business need as well -- if you are doing product recommendations for the next day, you might want recommendations for the top 20% of users precomputed and available in about 5 hours, if it takes 18 hours to train …
- Cost: What predictions you are doing, and how much you precompute is going to driven by cost considerations
- Scale: do you have to do this all on one machine, or can you distribute it to multiple workers? What kind of hardware are on these workers? Do they have GPUs?

Optimizing your online predictions

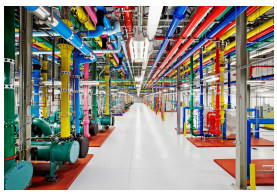
If you are doing online prediction, the performance considerations are quite different. This is because the end-user is waiting for the prediction. How is it different?

- You typically can not distribute the prediction graph; instead, you carry out the computation for one end-user on one machine
- However, you almost always scale out the predictions on to multiple workers. Essentially, each prediction is handled by a microservice, and you can replicate and scale out the predictions using Kubernetes or AppEngine -- Cloud AI Platform predictions are a higher-level abstraction, but they are equivalent to doing this.
- The performance consideration is not how many training steps can you carry out per minute, but how many queries you can handle per second. Queries Per Second or QPS. That's the performance target you need to hit.

When you design for high performance, you want consider training and prediction separately, especially if you will be doing online predictions.

Machine learning gets complex quickly

Heterogeneous systems



Distributed systems

Model architectures

As I kind-of suggested in my line about precomputing batch predictions for the top 20% of users, and handling the rest of your users via online prediction, performance considerations will often involve striking the right balance. And ultimately, you will know the exact tradeoff (is it 20% or 10% or 25%, that you need to do) only after you build your system and measure things. However, unless you plan to be able to do both batch predictions and online predictions, you will be stuck with a solution that doesn't meet your needs.

The idea behind this module, and this course in general, is so that you are aware of the possibilities. Once you are aware that it can be done, it's not that difficult to accomplish -- the technical part is usually quite straightforward. Especially if you are using TensorFlow on a capable cloud platform.

In the previous lesson, we talked about several ways of achieving high performance. We said that you could run the ML model on hardware that was more suitable -- faster CPUs, or GPUs or TPUs.

You could also scale out the training on to multiple machines.

However, all these three, using heterogeneous systems for ML training, training on distributed systems of machines, or experimenting with model architectures, they all add complexity to your production ML system.

Production Machine learning Systems become more complex because of this kind of flexibility you need. Your production ML systems are complex precisely so that they can provide the high performance that you need.

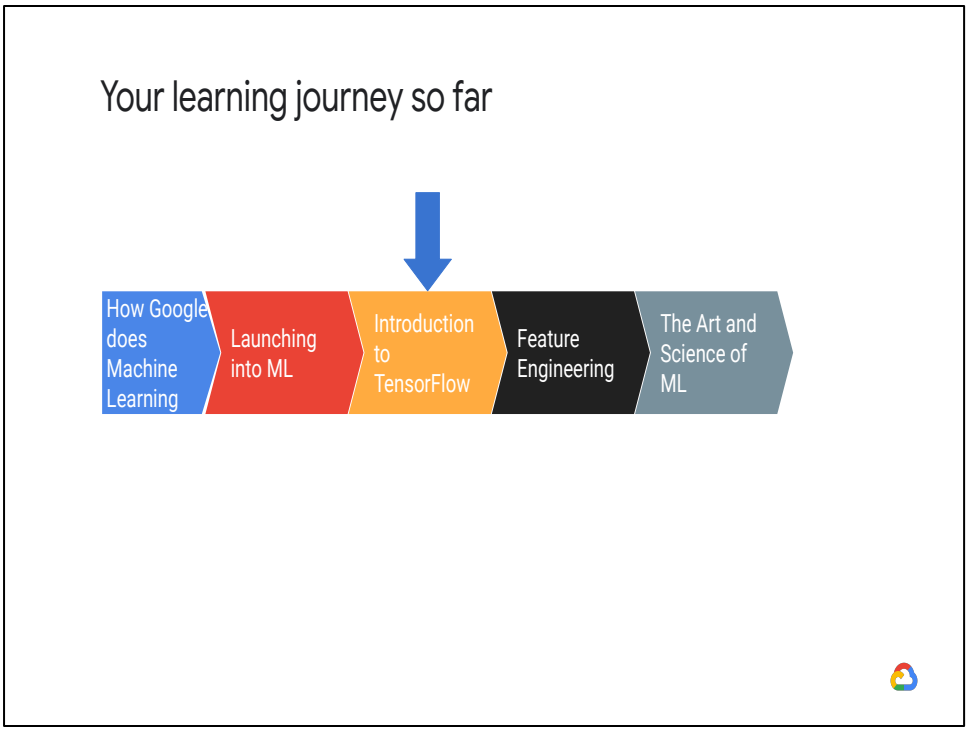
Lab

Training Models At Scale with AI Platform

In this notebook we'll make the jump from training locally, to do training in the cloud. We'll take advantage of Google Cloud's AI Platform Training Service.

AI Platform Training Service is a managed service that allows the training and deployment of ML models without having to provision or maintain servers. The infrastructure is handled seamlessly by the managed service for us.

Link to lab: [ML on GCP C5] Training Models at Scale with AI Platform. (qwiklabs). Github repo link here. CBL292



And with that, we come to the end of the third chapter of this specialization.

cloud.google.com