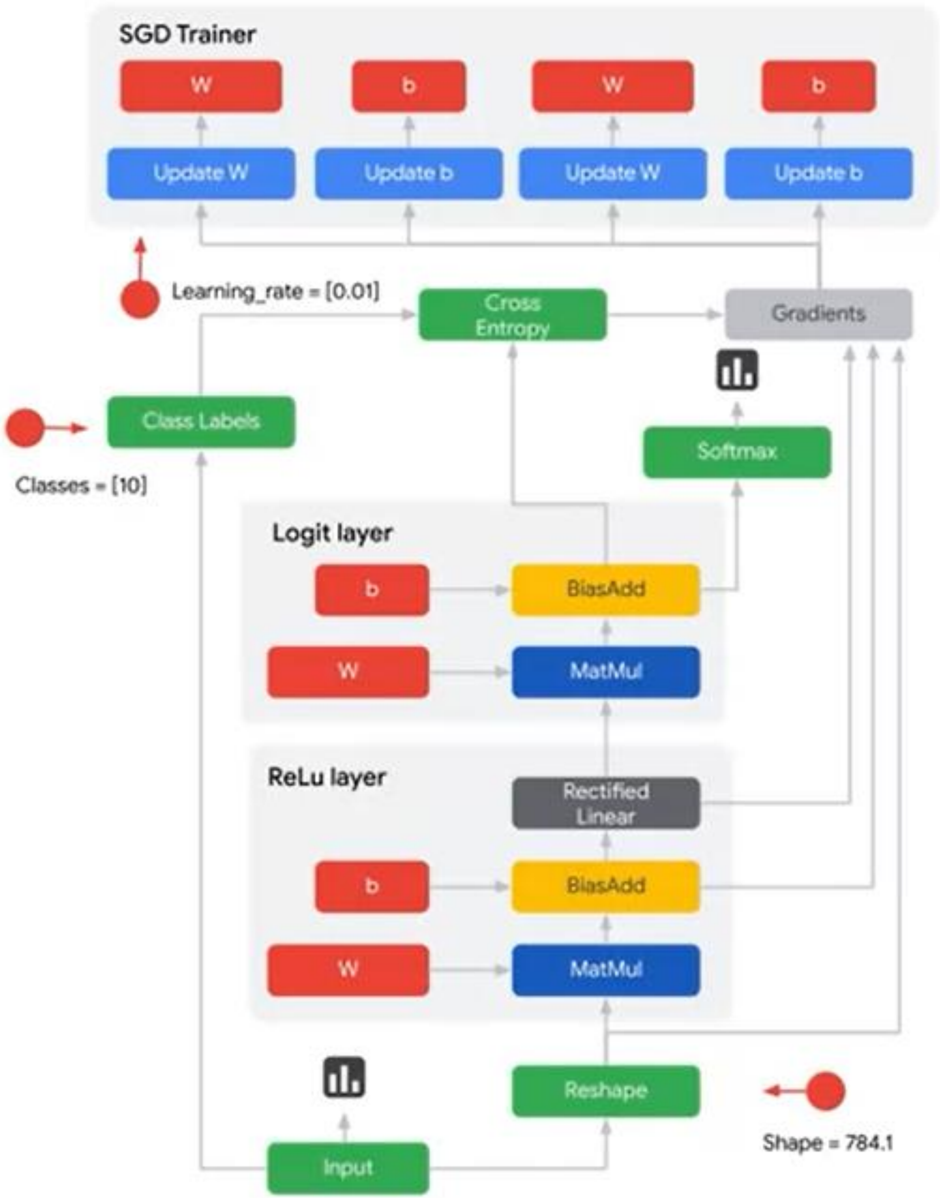
Tensorflow on Google Cloud

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



A tensor is an N-dimensional array of data

TensorFlow API Hierarchy

Note that AI Platform is now Vertex AI.

Components of TensorFlow: Tensors and Variables

A tensor is an N-dimensional array of data

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

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

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

A tensor is an N-dimensional array of data

They behave like numpy n-dimensional arrays except that:

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

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

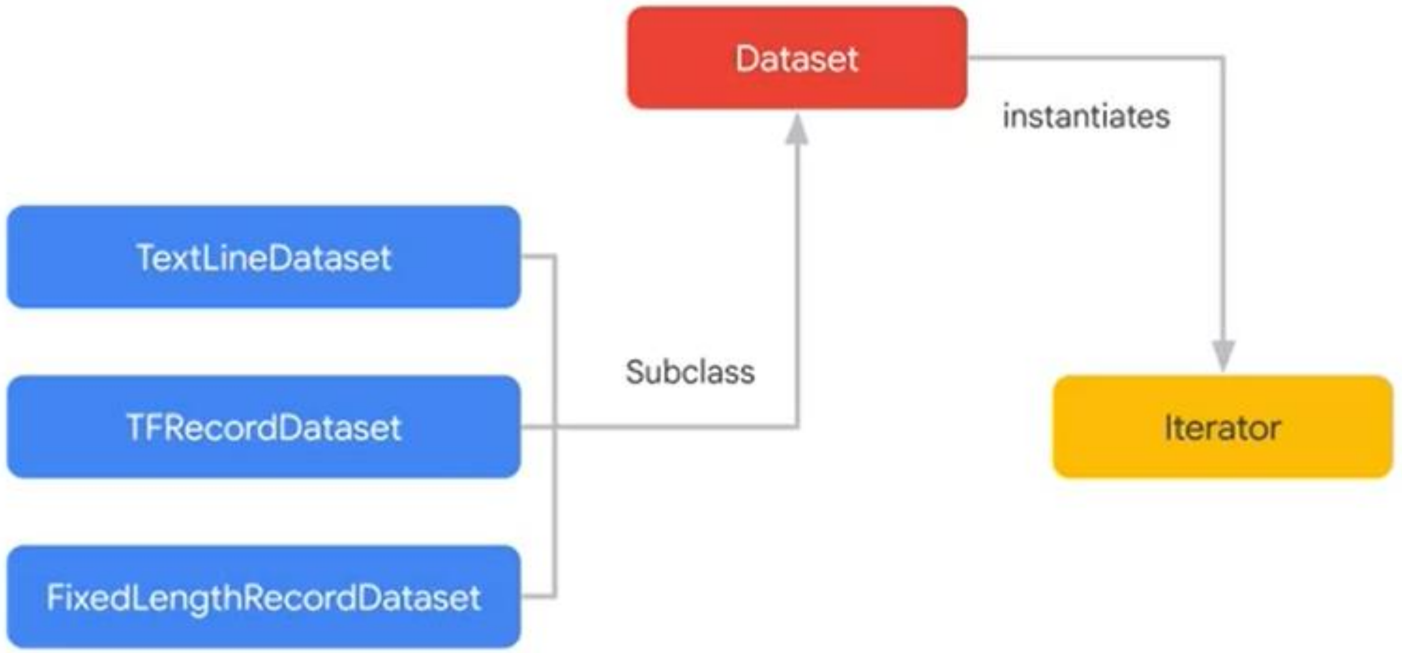
tf.Variable will typically hold model weights
that need to be updated in a training loop.

```
import tensorflow as tf

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

# x <- 48.5
x.assign(45.8)

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

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

tf.data API

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

Multiple ways to feed TensorFlow models with data



A tf.data.Dataset allows you to

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

- Preprocess data in parallel (and cache result of costly operations)

```
dataset = dataset.map(preproc_fun).cache()
```

- Configure the way the data is fed into a model with a number of chaining methods

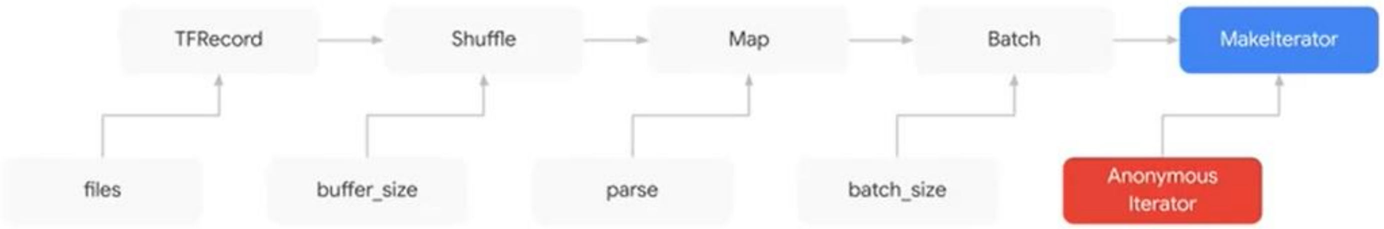
```
dataset = dataset.shuffle(1000).repeat(epochs).batch(batch_size,
drop_remainder=True)
```

in a easy and very compact way

Training on large datasets with tf.data API

TFRecordDataset example

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

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



Working in-memory and with files

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

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.

Use from_tensors() or from_tensor_slices()

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


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

Read one CSV file using TextLineDataset

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

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

dataset = “[line1, line2, etc.]”

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

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

Read a set of sharded CSV files using TextLineDataset

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


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

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

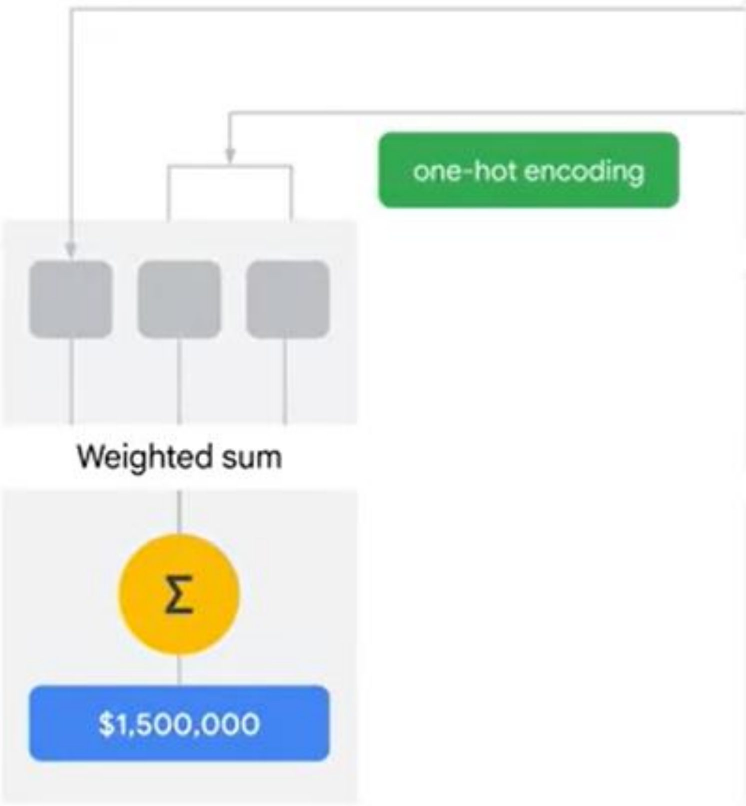
- train.csv-00000-of-00011
- train.csv-00001-of-00011
- train.csv-00002-of-00011
- train.csv-00003-of-00011
- train.csv-00004-of-00011
- train.csv-00005-of-00011

Without prefetching

With prefetching

With prefetching + multithreaded loading & preprocessing

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




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

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

```
" house" -> 1, 0
" apt"    -> 0, 1
Many more...
```

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

Representing feature columns as sparse vectors

These are all different ways to create a categorical column.

If you know the keys beforehand:

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

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

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

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

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

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

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

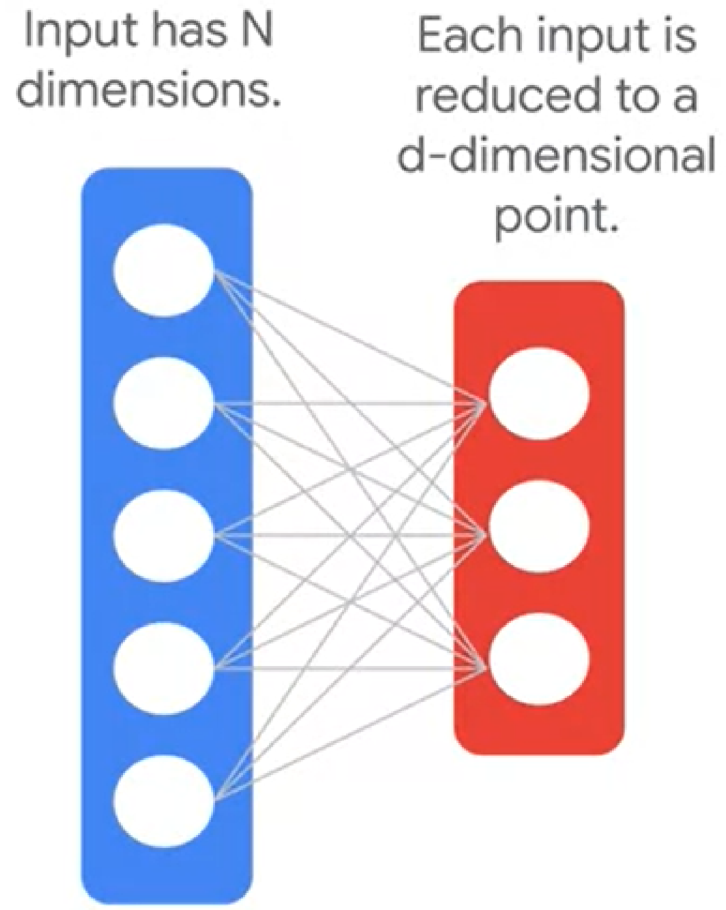
lower dimensional, dense vector in which each cell contains a number, not just 0 or 1

5:30 / 6:00

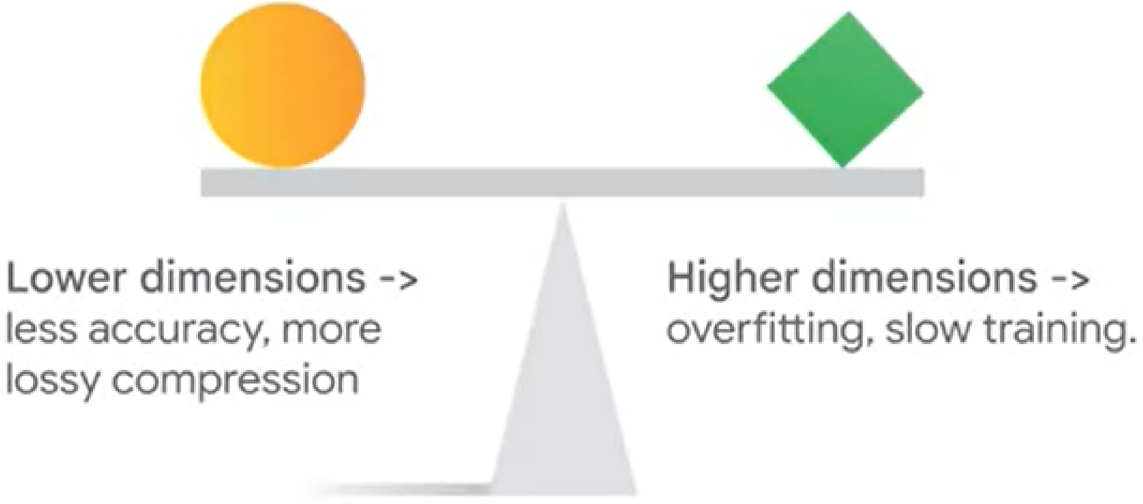
Embeddings

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

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



A good starting point for number of embedding dimensions



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

Empirical tradeoff.

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

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

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

Scaling data processing with tf.data and Keras preprocessing layers

Scaling data preprocessing with
tf.data and Keras preprocessing layers

Data preprocessing

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

Keras preprocessing layers

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

Text features preprocessing

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

Text vectorization layer

[tf.keras.layers.TextVectorization:]()

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

Numerical features preprocessing

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

Normalization class

Feature-wise normalization of the data

tf.keras.layers.Normalization:

performs feature-wise normalization of input features.

Numerical preprocessing

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

Discretization class

Buckets data into discrete ranges.

[tf.keras.layers.Discretization:]()

turns continuous numerical features into bucket data with discrete ranges.

Categorical features preprocessing

**Tf.keras.layers.
CategoryEncoding**

Turns integer categorical features into one-hot, multi-hot, or count dense representations.

**Tf.keras.layers.
Hashing**

Performs categorical feature hashing, also known as the "hashing trick."

**Tf.keras.layers.
StringLookup**

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

**Tf.keras.layers.
IntegerLookup**

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

The adapt() method

Stateful preprocessing layers that compute based on training data/:

01

TextVectorization:

Holds a mapping between string tokens and integer indices.

02

StringLookup and IntegerLookup:

Hold a mapping between input values and integer indices.

03

Normalization:

Holds the mean and standard deviation of the features.

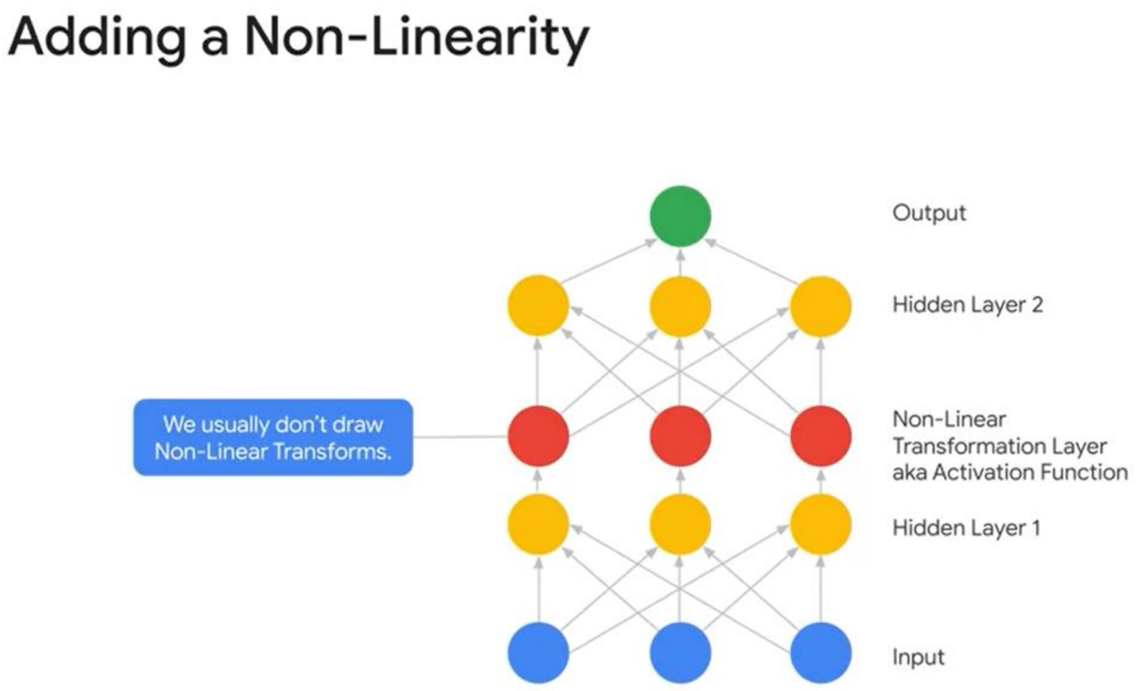
04

Discretization:

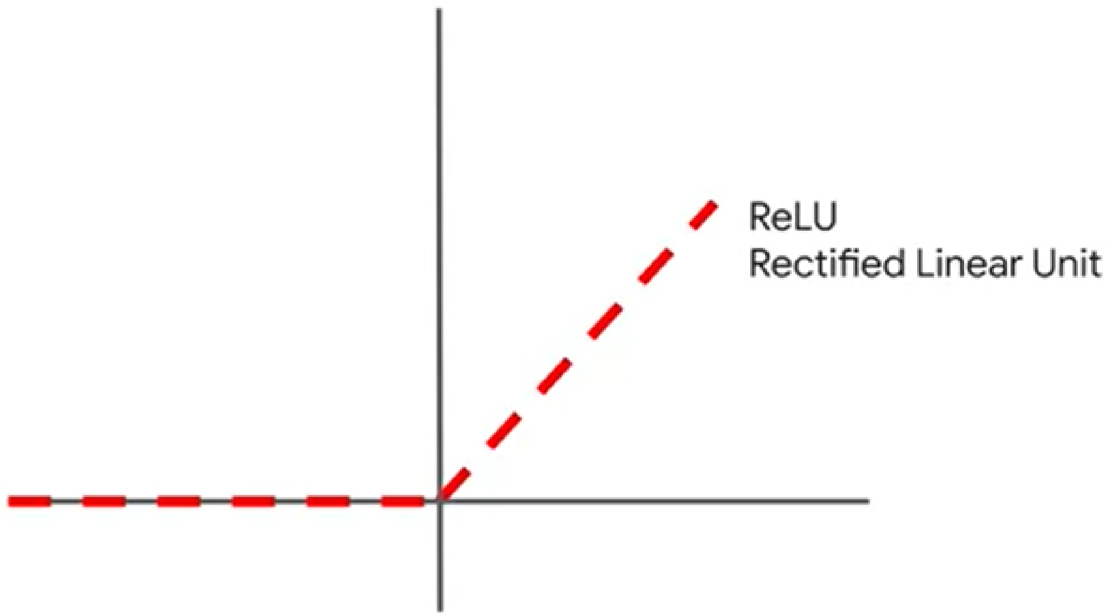
Holds information about value bucket boundaries.

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

Activation functions



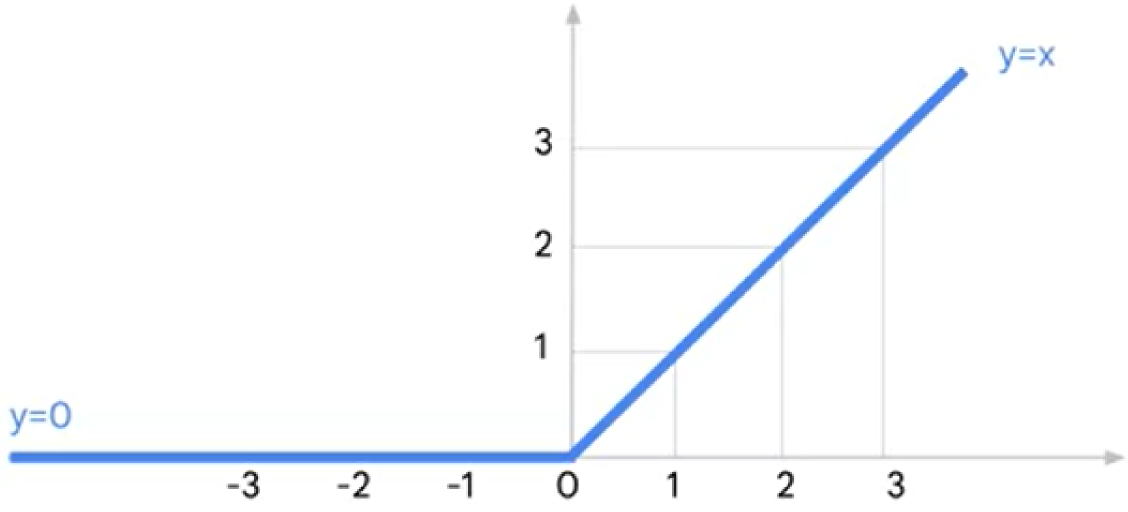
Our favorite non-linearity is the Rectified Linear Unit



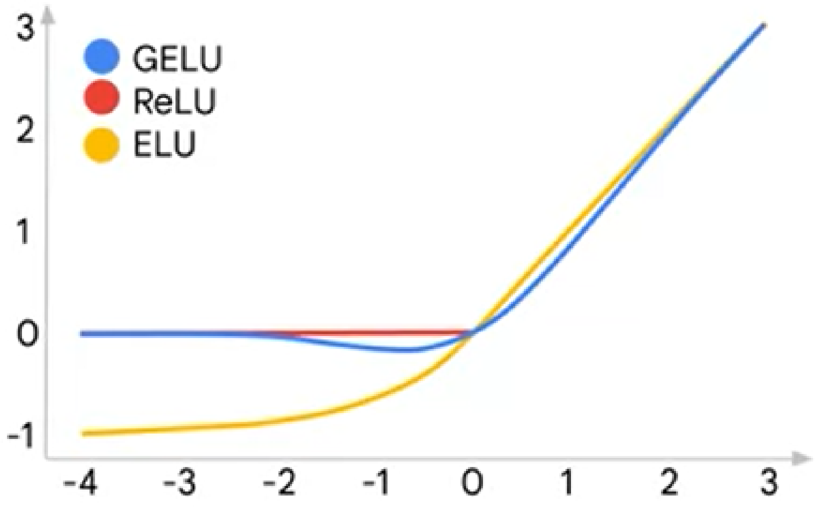
$$f(x) = \max(0, x)$$

There are many different ReLU variants

Normal ReLU activation function



There are many different ReLU variants



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

SavedModel is the universal serialization format for TensorFlow models

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

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

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

the directory in which to write the SavedModel

exports a model object to a SavedModel format

a trackable object such as a trained keras model