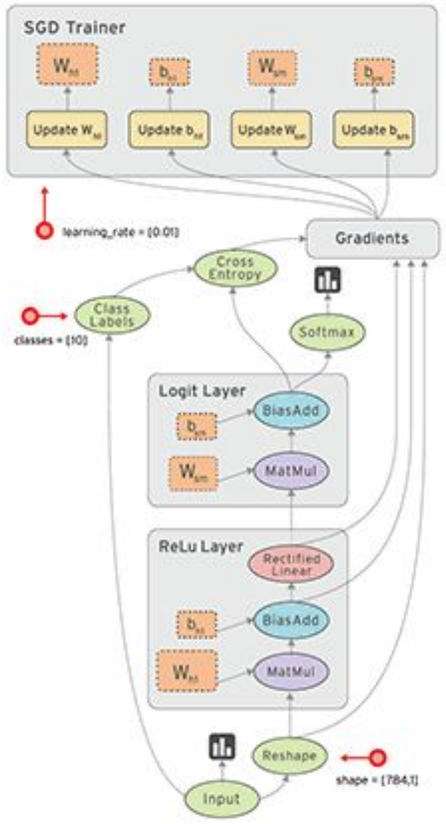
Agenda

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

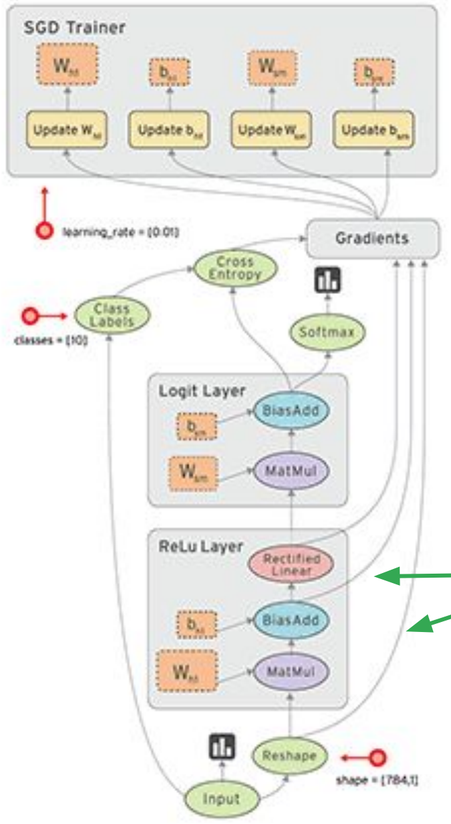
Google Cloud

Introduction to Tensorflow

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

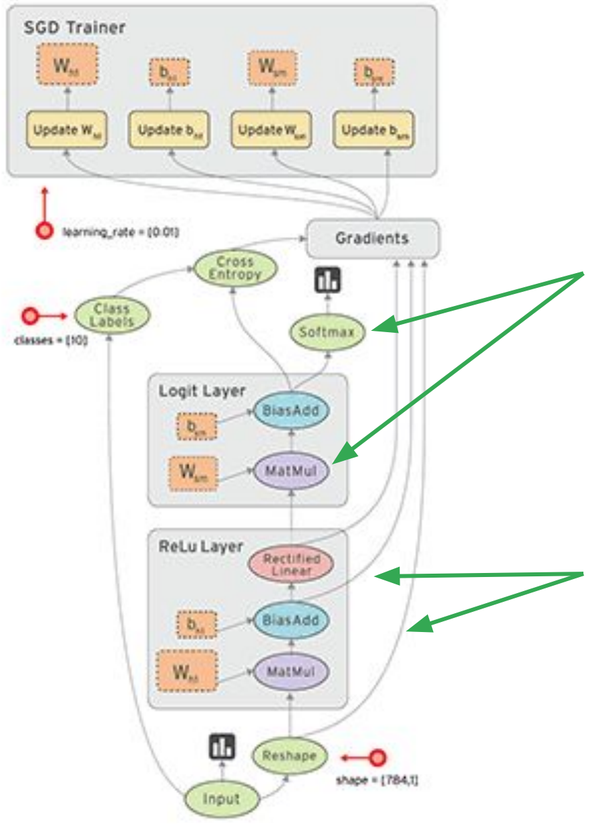


TensorFlow is an open-source, high-performance library for numerical computation that uses directed graphs



Edges represent arrays of data.

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



Nodes represent mathematical operations.

Edges represent arrays of data.

A tensor is an N-dimensional array of data

Rank 0 Tensor

scalar

Rank 1 Tensor

vector

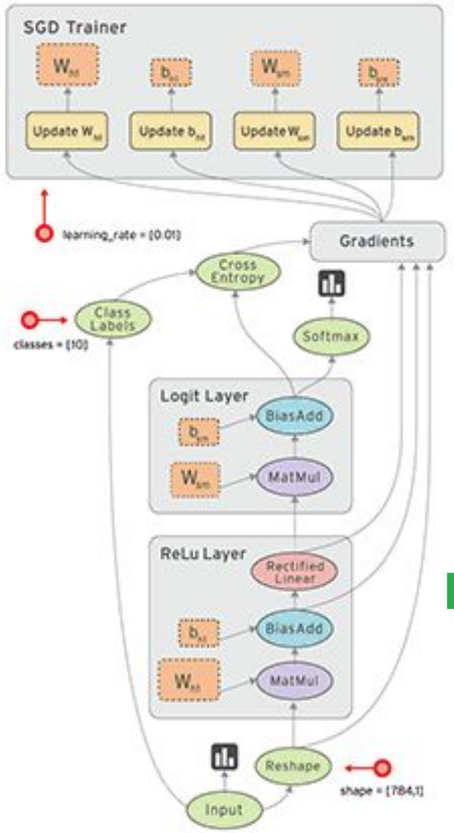
Rank 2 Tensor

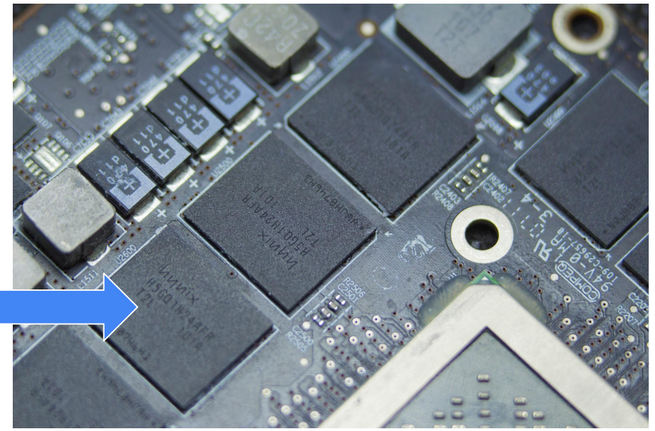
matrix

Rank 3 Tensor

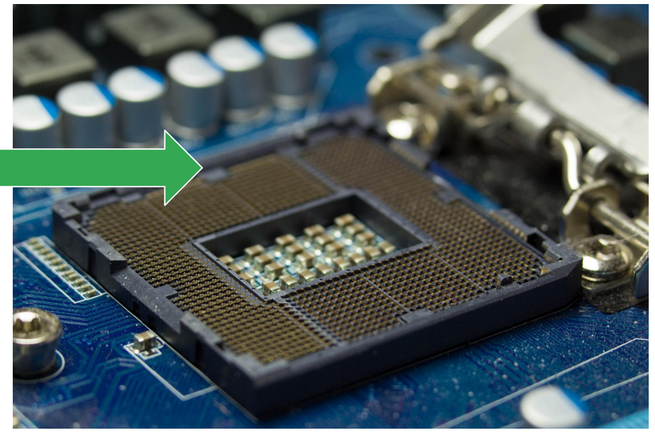
Rank 4 Tensor

TensorFlow graphs are portable between different devices





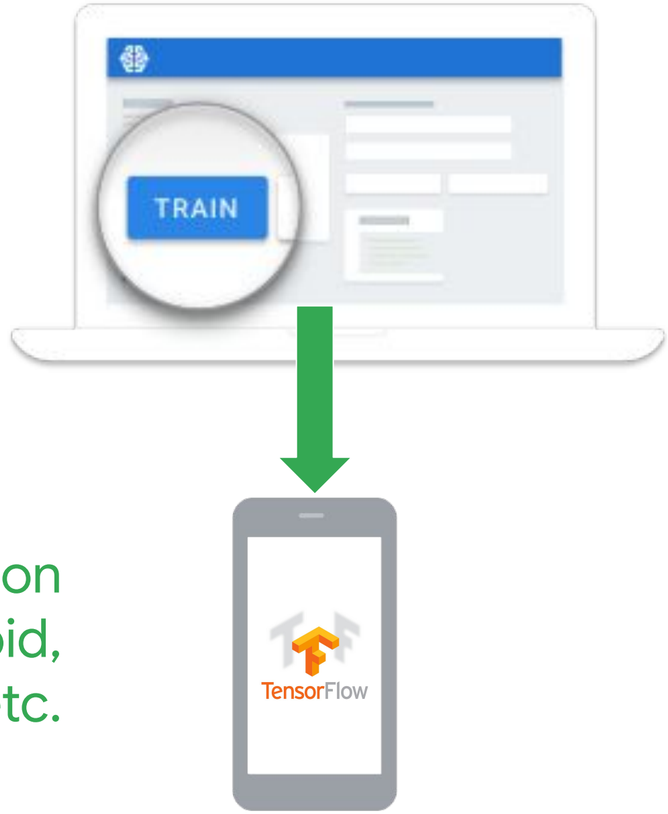
CPUs



GPUs

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

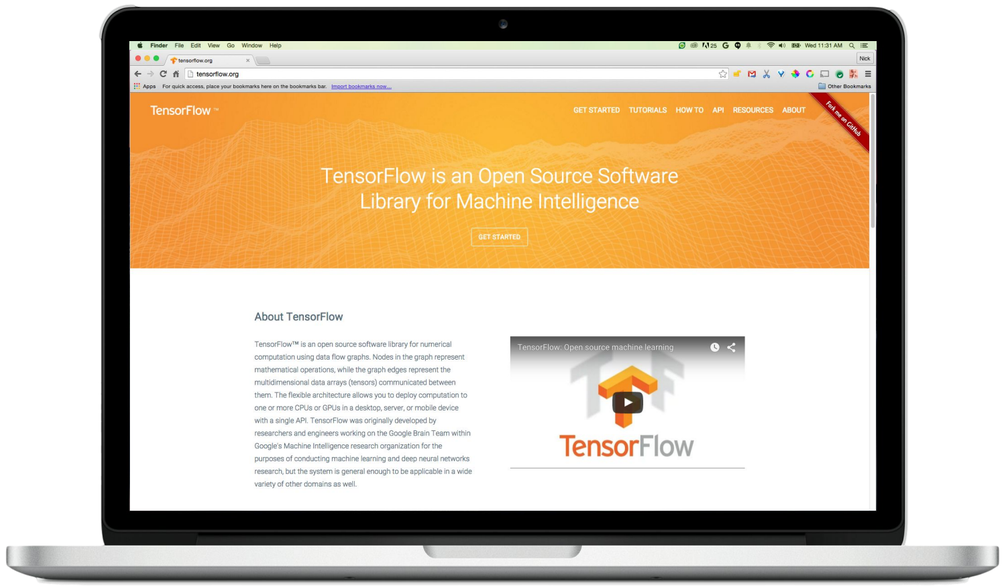
Train on cloud.



Run inference on iOS, Android, Raspberry Pi, etc.

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

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



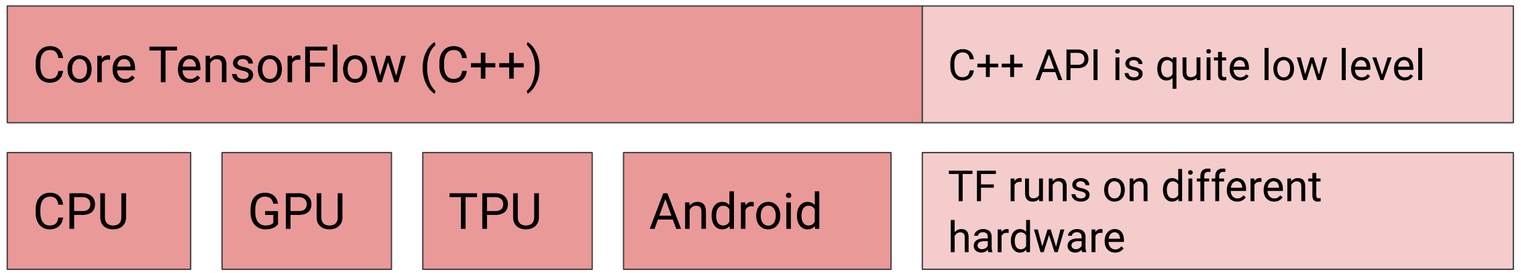
#1 repository
for “machine learning”
category on GitHub

Google Cloud

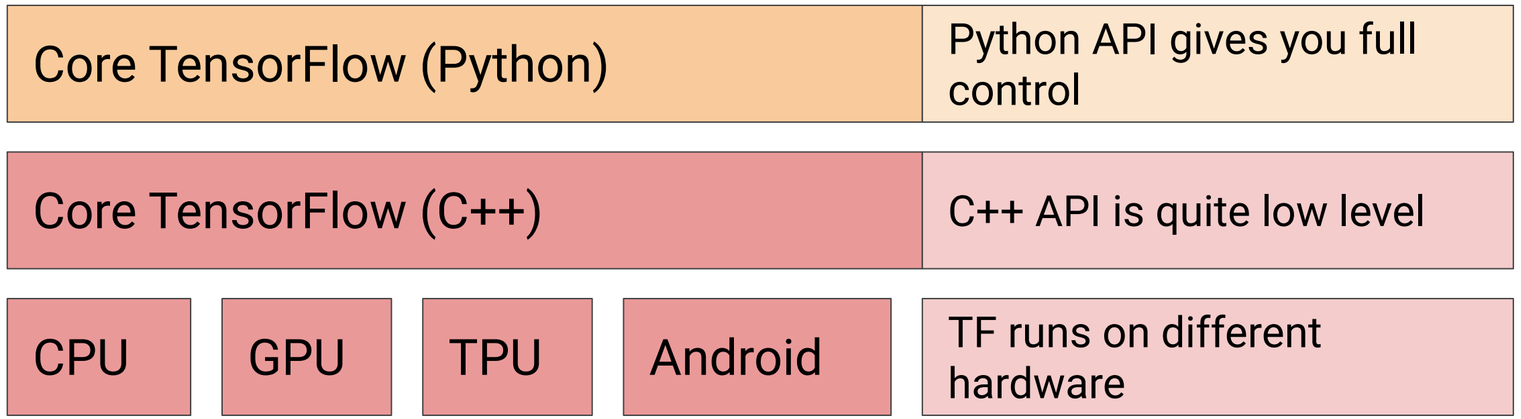
TensorFlow API Hierarchy

TensorFlow contains multiple abstraction layers

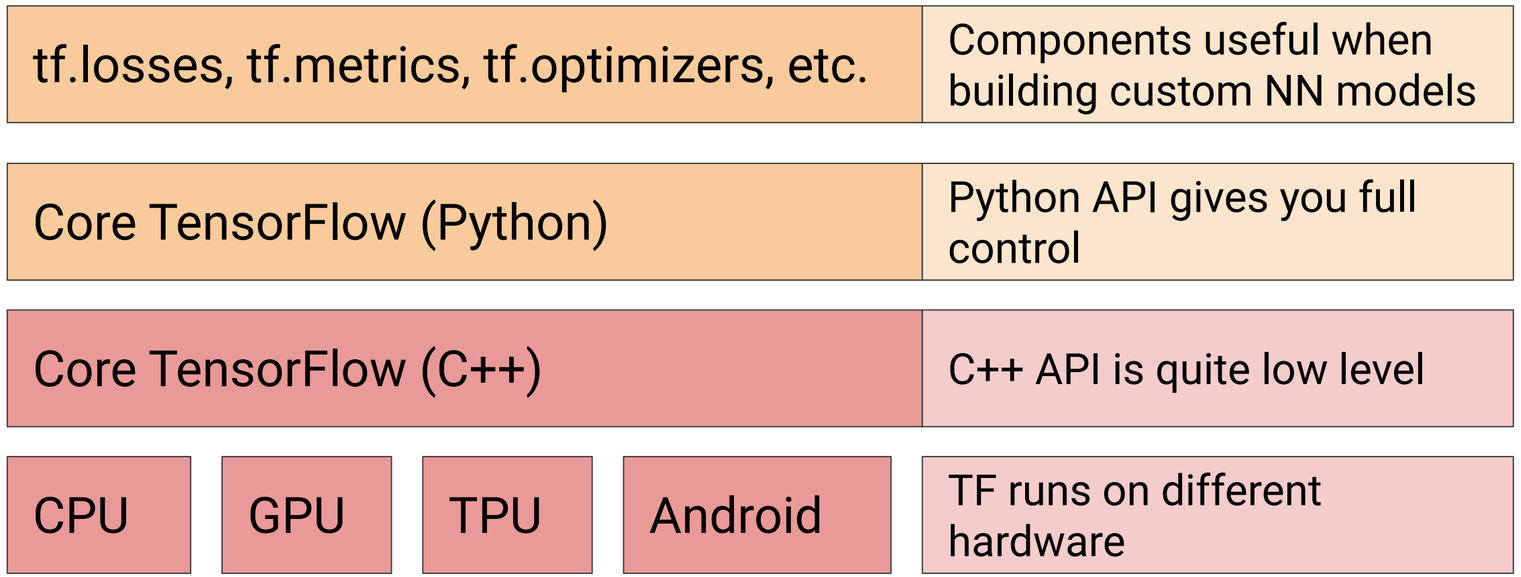
TensorFlow contains multiple abstraction layers



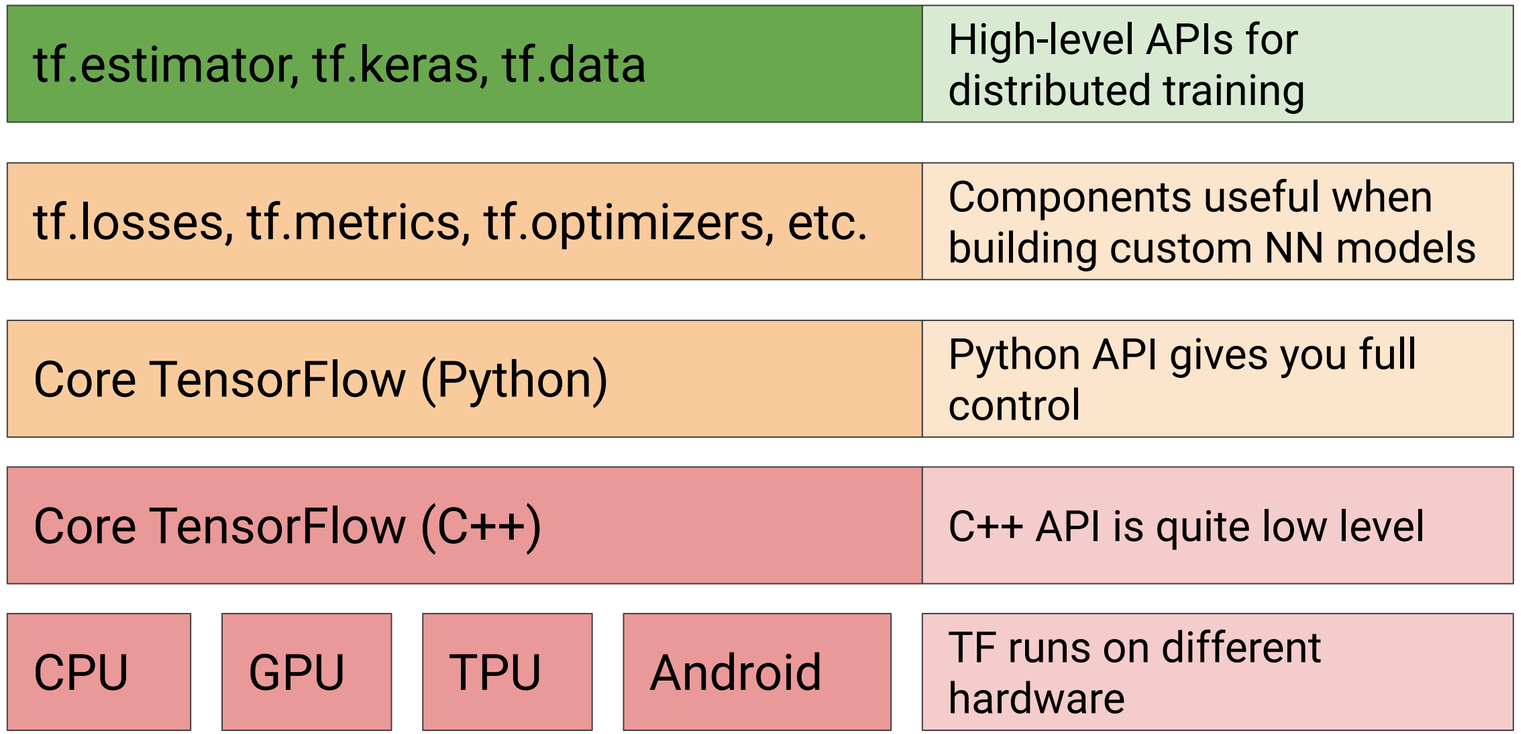
TensorFlow contains multiple abstraction layers



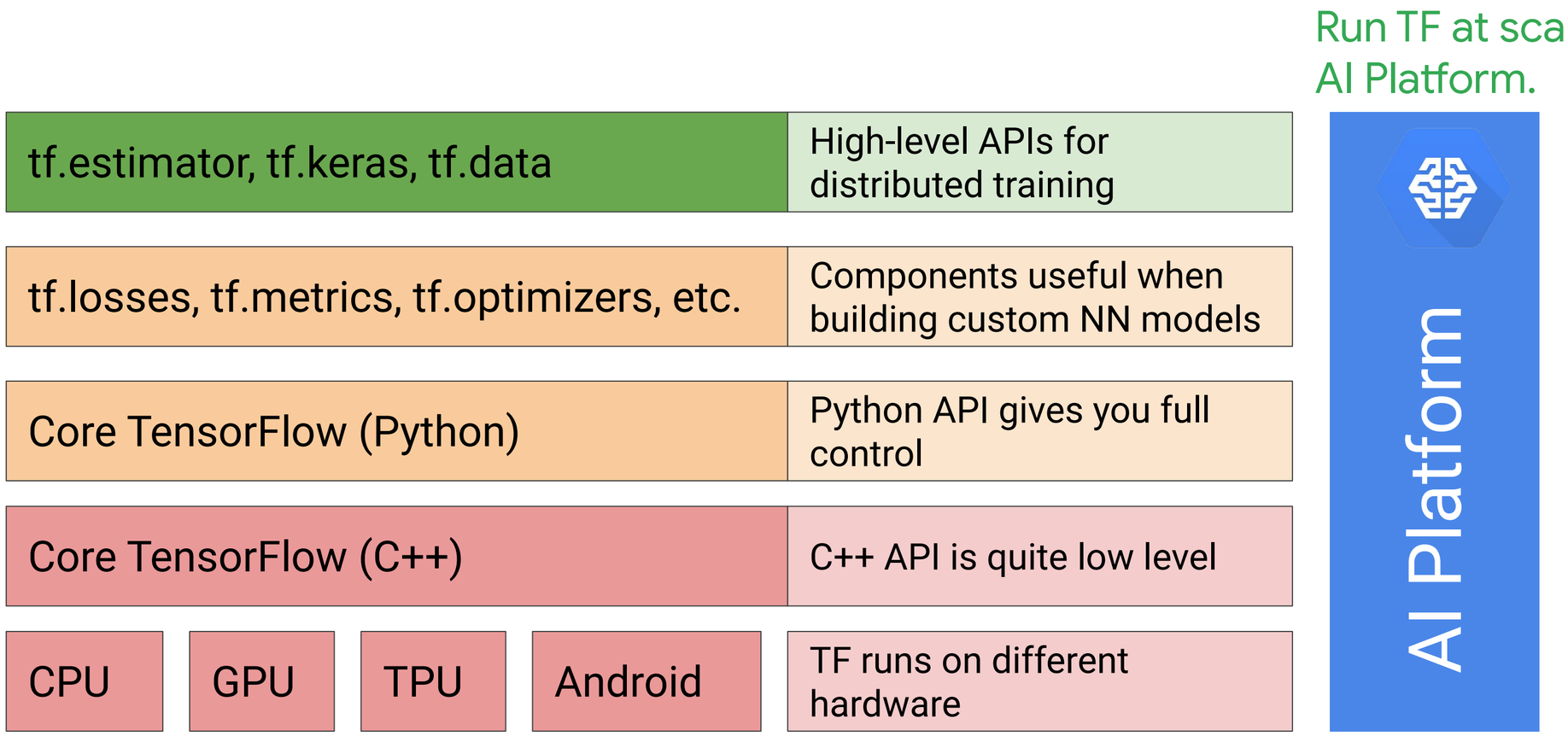
TensorFlow contains multiple abstraction layers



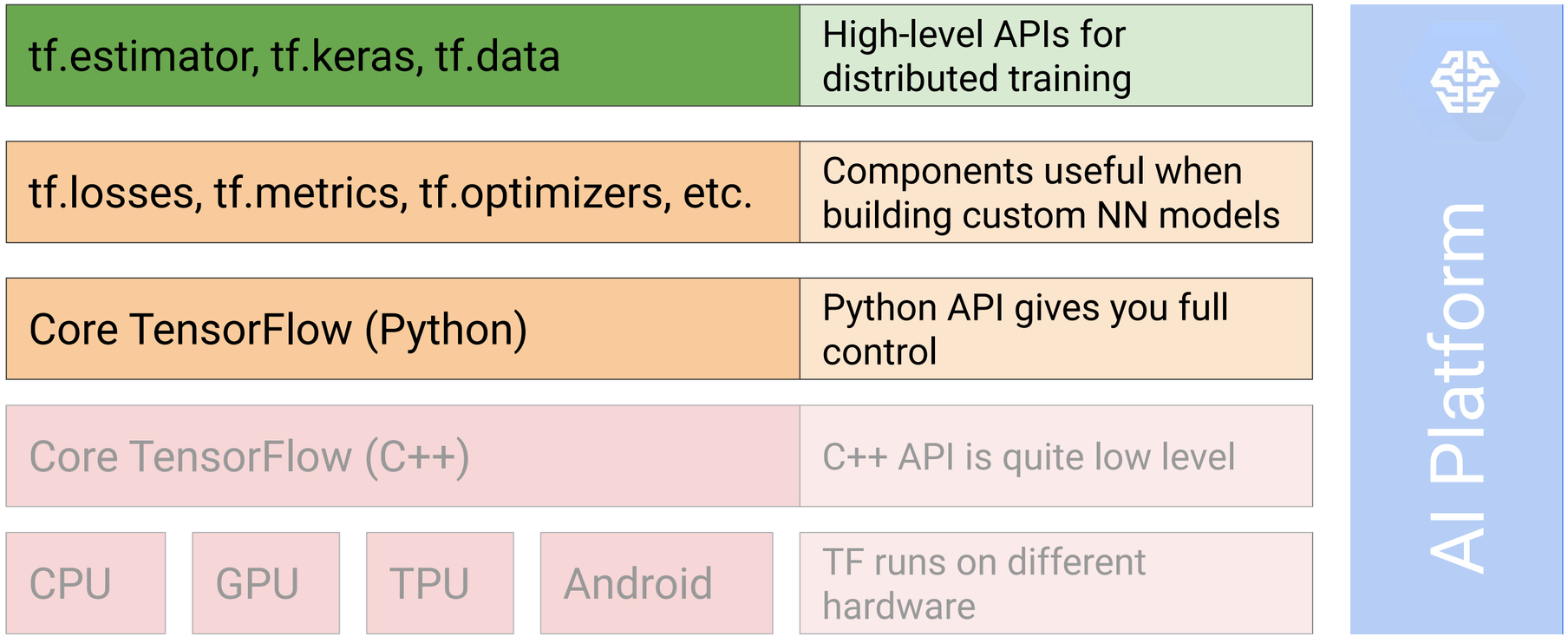
TensorFlow contains multiple abstraction layers



TensorFlow contains multiple abstraction layers



TensorFlow toolkit hierarchy



Google Cloud

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)`	()

A tensor is an N-dimensional array of data

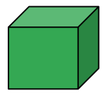
	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)

A tensor is an N-dimensional array of data

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

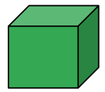
A tensor is an N-dimensional array of data

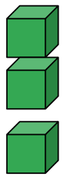
	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

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

A tensor is an N-dimensional array of data

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

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

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

A tensor is an N-dimensional array of data

They behave like numpy n-dimensional arrays except that

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

tf.constant produces constant tensors...

```
import tensorflow as tf

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

Tensors can be sliced

```
import tensorflow as tf

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

y = x[:, 1]

[5, 6]
```

Tensors can be reshaped

```
import tensorflow as tf

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

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

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

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

```
import tensorflow as tf

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

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

```
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

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

```
import tensorflow as tf

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

GradientTape records operations for Automatic Differentiation

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

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

GradientTape records operations for Automatic Differentiation

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

record the computation with GradientTape when it's executed (not when it's defined!)

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

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

GradientTape records operations for Automatic Differentiation

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

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

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

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

Google Cloud

Training on Large Datasets with tf.data

A tf.data.Dataset allows you to

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

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

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

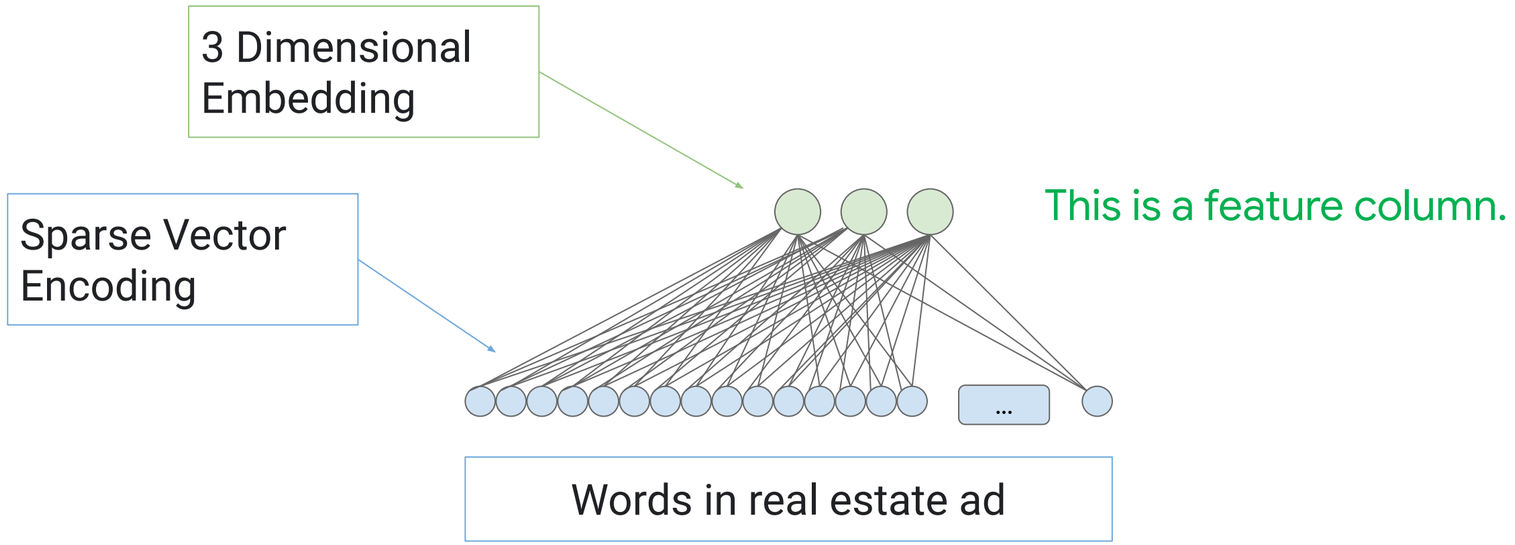
- Configure the way the data is fed into a model with a number of chaining methods

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

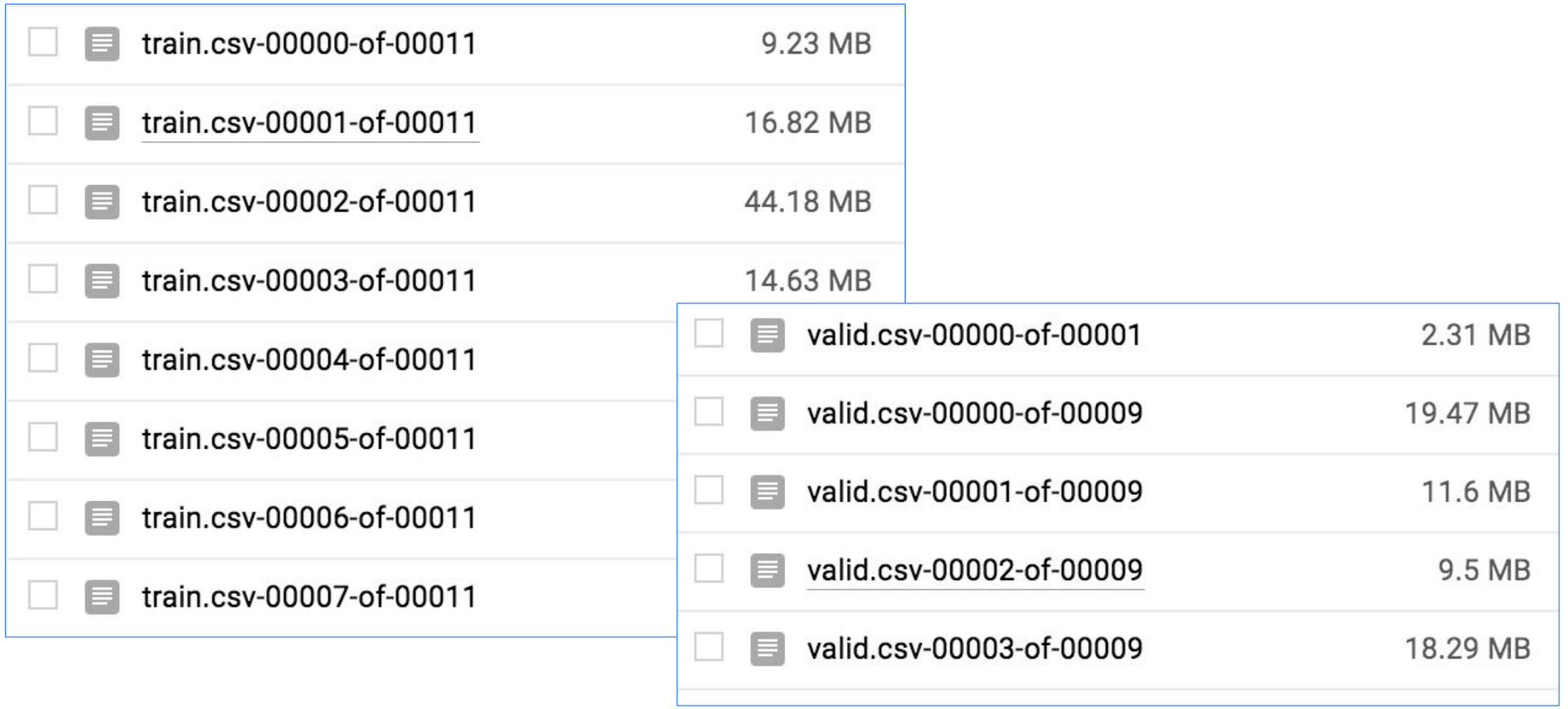
in a easy and very compact way

Embeddings are feature columns that function like layers

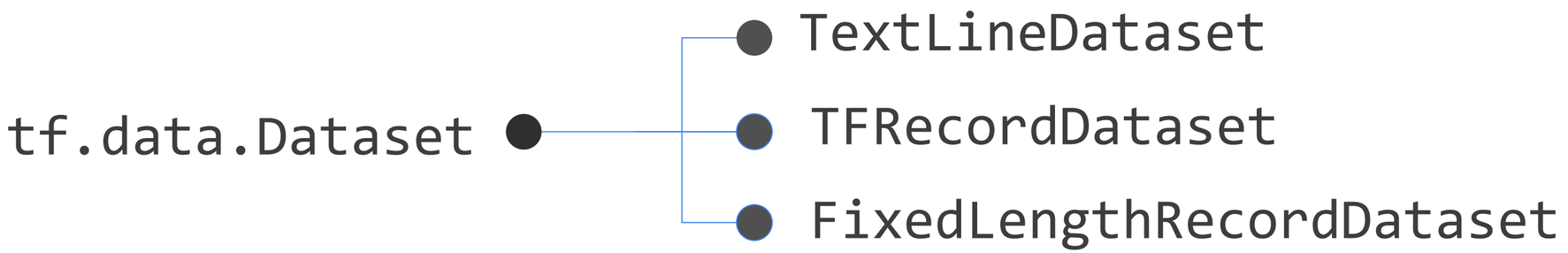
```
import tf.feature_column as fc

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

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



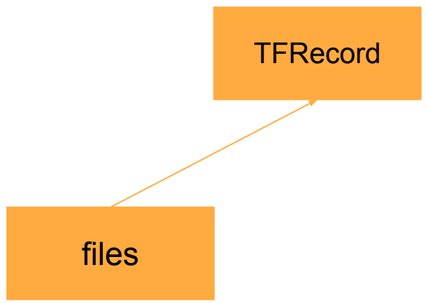
What about out-of-memory sharded datasets?



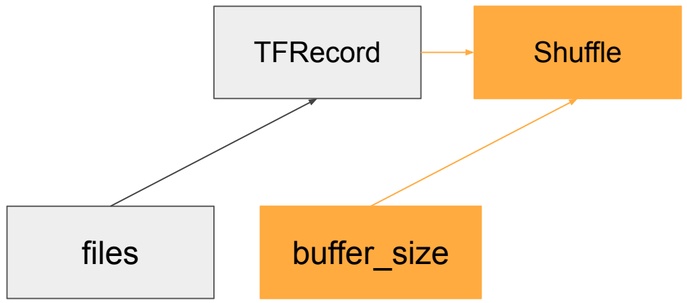
Datasets can be created from different file formats.



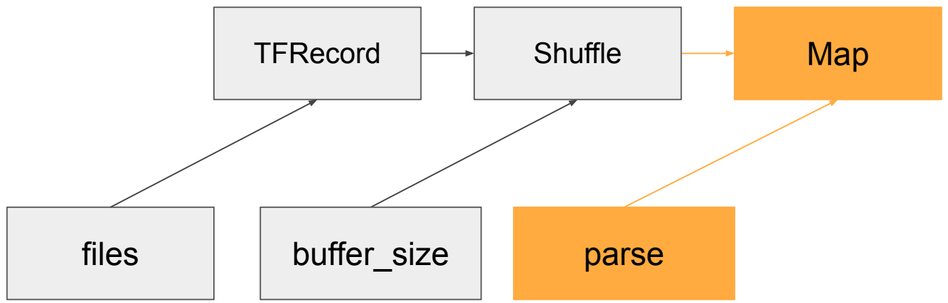
```
dataset = tf.data.TFRecordDataset(files)
```



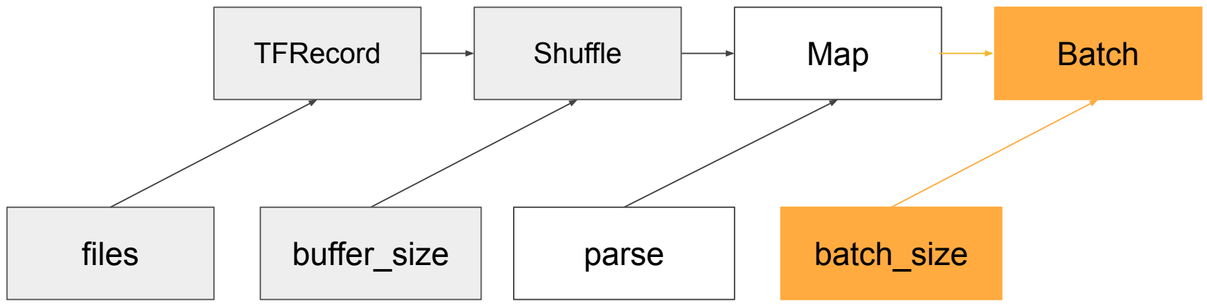
```
dataset = tf.data.TFRecordDataset(files)

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



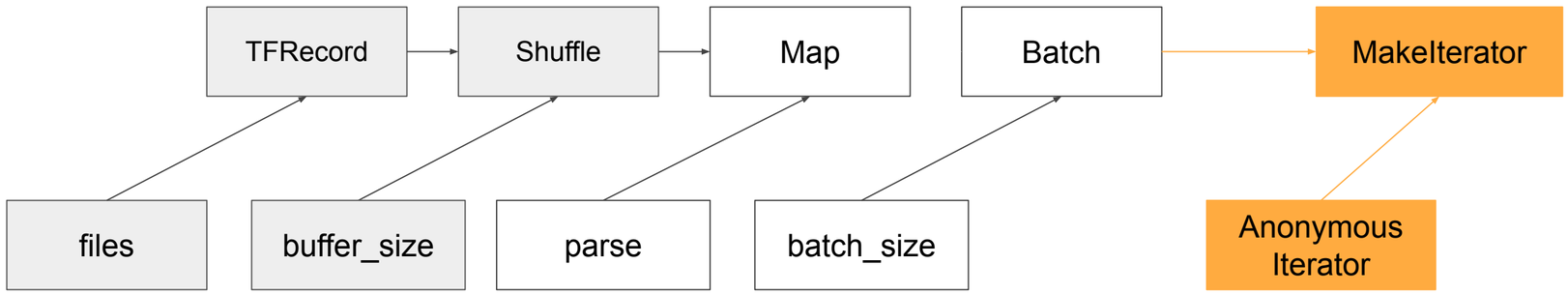
```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

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



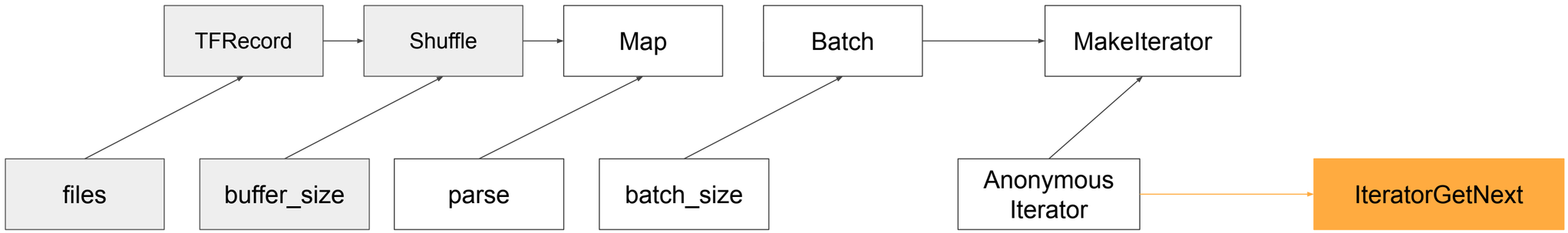
```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

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

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



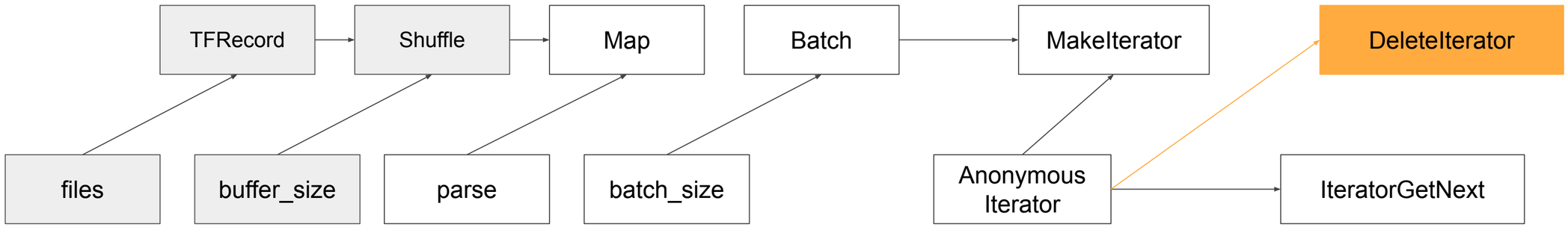
```
dataset = tf.data.TFRecordDataset(files)

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

  ...
```



```
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
```

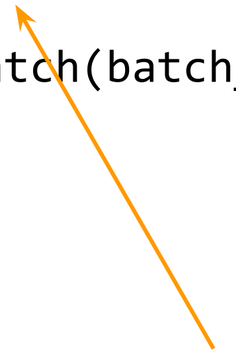


```
dataset = tf.data.TFRecordDataset(files)

dataset = dataset.shuffle(buffer_size=X)

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

dataset = dataset.batch(batch_size=Y)


for element in dataset:

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



Creating a dataset from in-memory tensors

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



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

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

Use from_tensors() or from_tensor_slices()

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

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

Read one CSV file using TextLineDataset

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


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

dataset = “[line1, line2, etc.]”

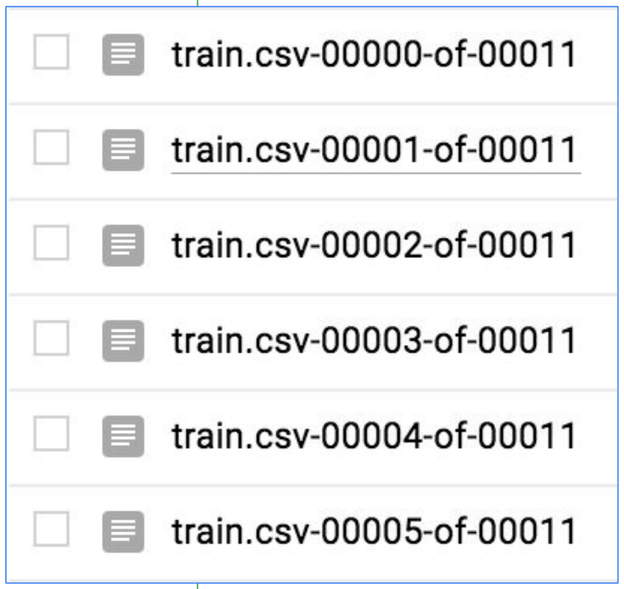
dataset = “[parse_row(line1), parse_row(line2), etc.]”

property type

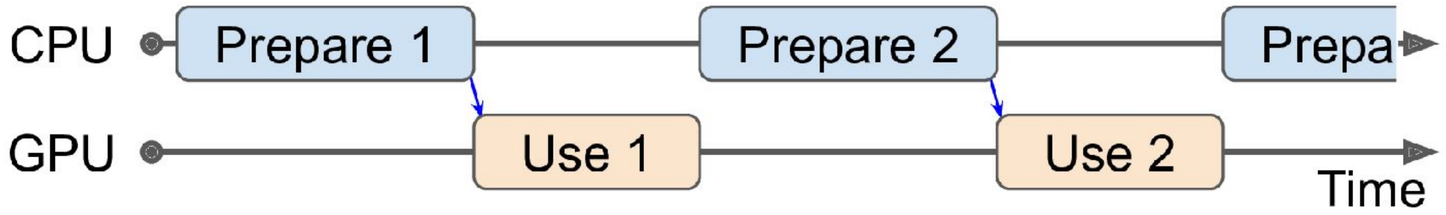
sq_footage		PRICE in K$
1001,	house,	501
2001,	house,	1001
3001,	house,	1501
1001,	apt,	701
2001,	apt,	1301
3001,	apt,	1901
1101,	house,	526
2101,	house,	1026

Read a set of sharded CSV files using TextLineDataset

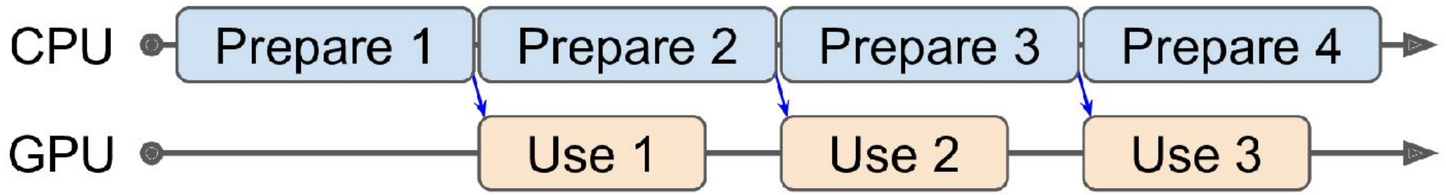
```
def parse_row(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label


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

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



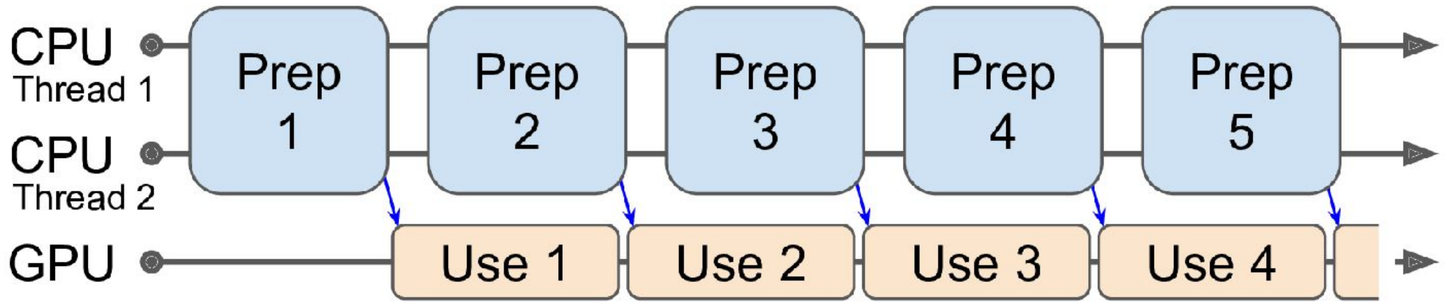
Without prefetching



With prefetching



With prefetching + multithreaded loading & preprocessing



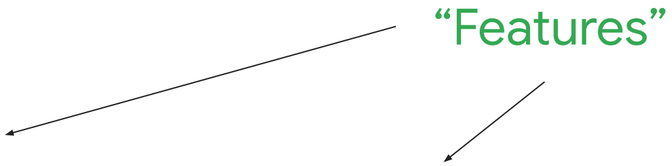
The real benefit of Dataset is that you can do more than just ingest data

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

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

Feature columns tell the model what inputs to expect

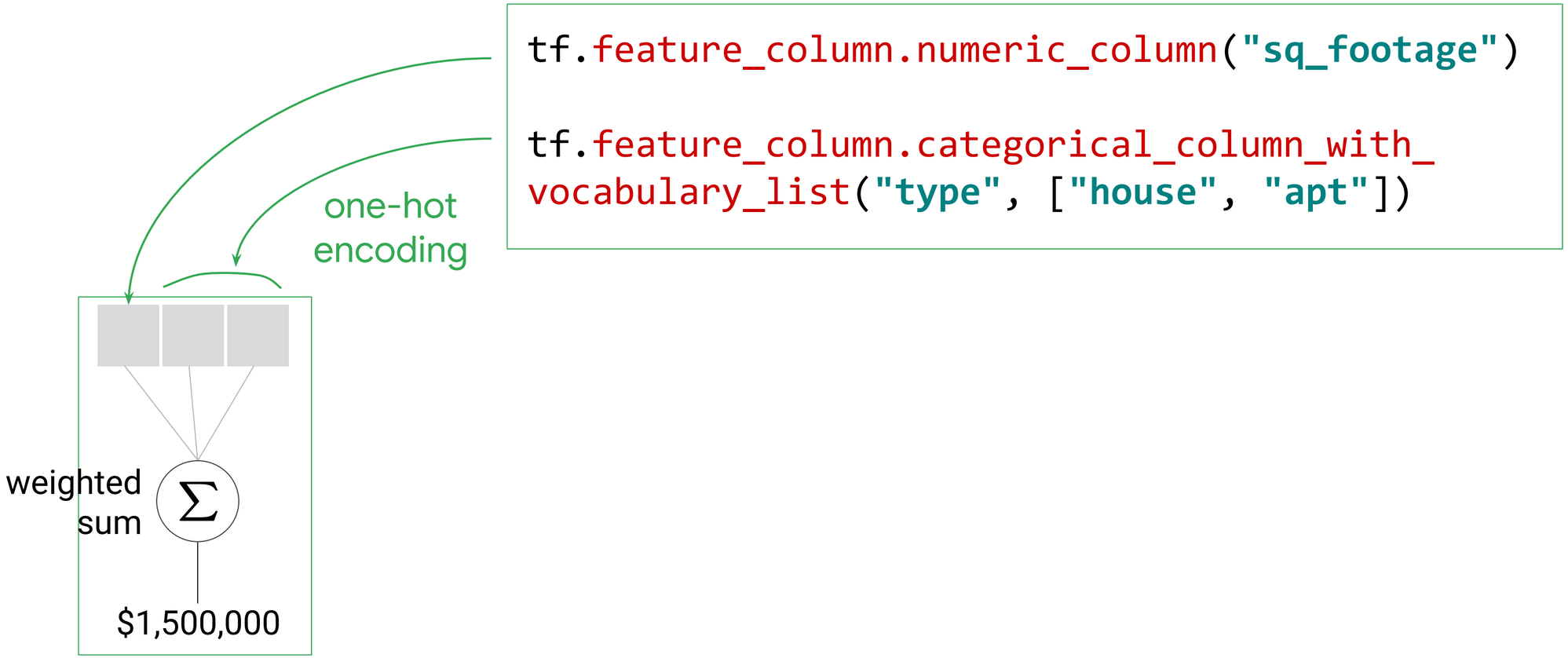
```
import tensorflow as tf

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

...
```



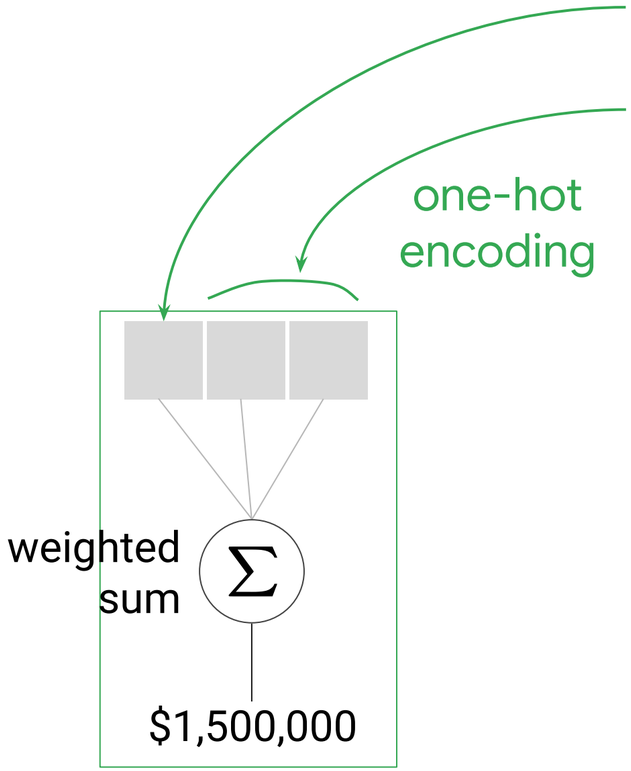
Under the hood: Feature columns take care of packing the inputs into the input vector of the model

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

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

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



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



one-hot
encoding

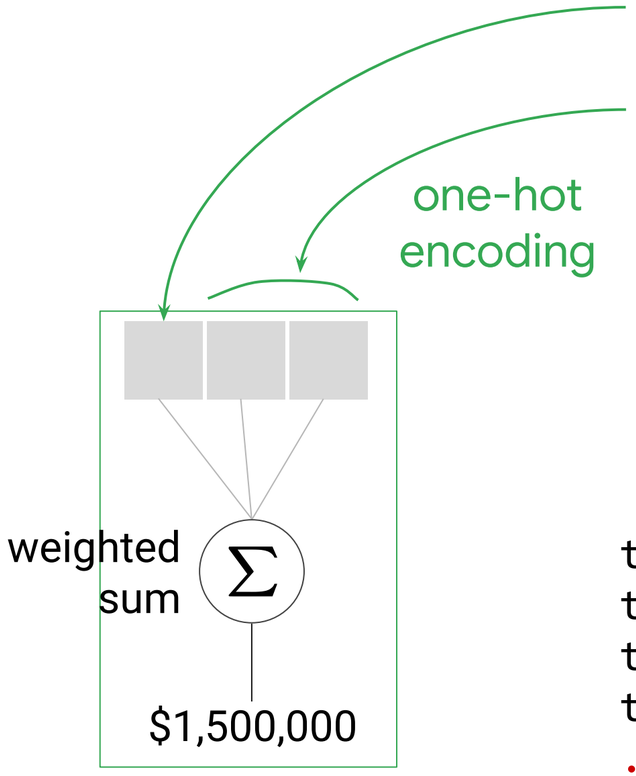
```
tf.feature_column.numeric_column("sq_footage")

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

“ house” -> 1, 0

“ apt” -> 0, 1

Many more...

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




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

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

“ house” -> 1, 0

“ apt” -> 0, 1

Many more...

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

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

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

set up numeric ranges

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

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

...
```

create bucketized columns for pickup latitude and pickup 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):

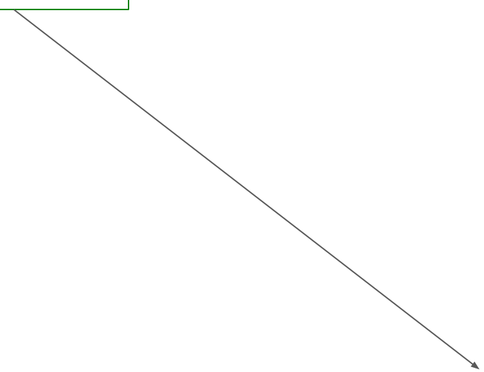
```
tf.feature_column.categorical_column_with_identity('schoolsRatings',
    num_buckets = 2)
```

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

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

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

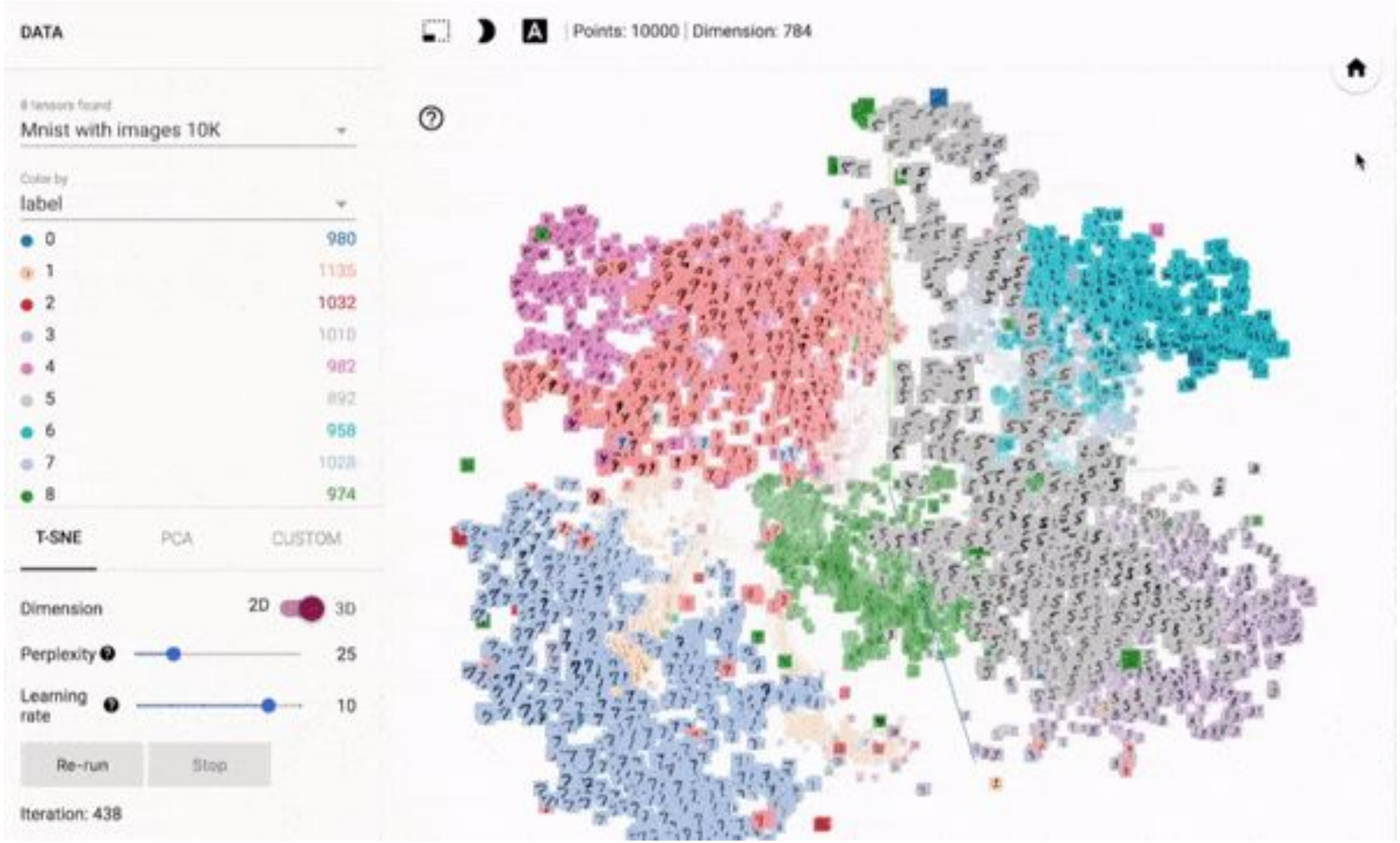
```
fc_ploc = fc.embedding_column(categorical_column=fc_crossed_ploc,
                              dimension=3)
```



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

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

Embeddings are everywhere in modern machine learning



How do you recommend movies to customers?

☺	2							4
☺				5			2	
☺						3		
☺			4					
☺					4		5	
☺		5						

1 million customers





500,000 movies

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

Average age of viewers

Shrek

Incredibles

The Triplets of Belleville

Harry Potter

Star Wars

Bleu

The Dark Knight Rises

Memento

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

Shrek

Incredibles

Harry Potter

Star Wars

The Dark Knight Rises

The Triplets of Belleville

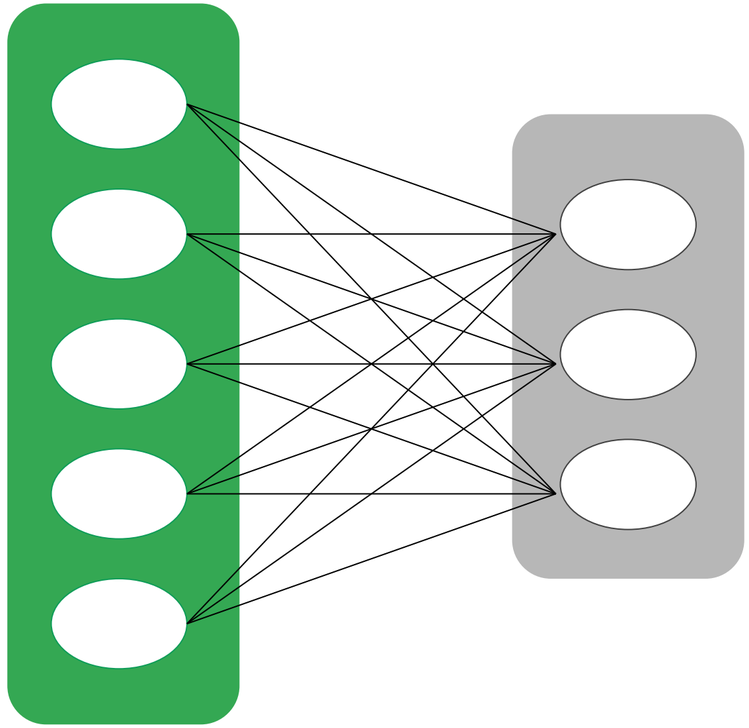
Bleu

Memento

Gross ticket sales

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

Input has N dimensions.



Each input is reduced to a d-dimensional point.

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.

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

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

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

Training input data requires dictionary of features and a label

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

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

Create input pipeline using tf.data

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

    return dataset
```

Use DenseFeatures layer to input feature columns to the Keras model

```
feature_columns = [...]


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


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

...
```

What about compiling and training the Keras model?

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

model.fit()

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

Agenda

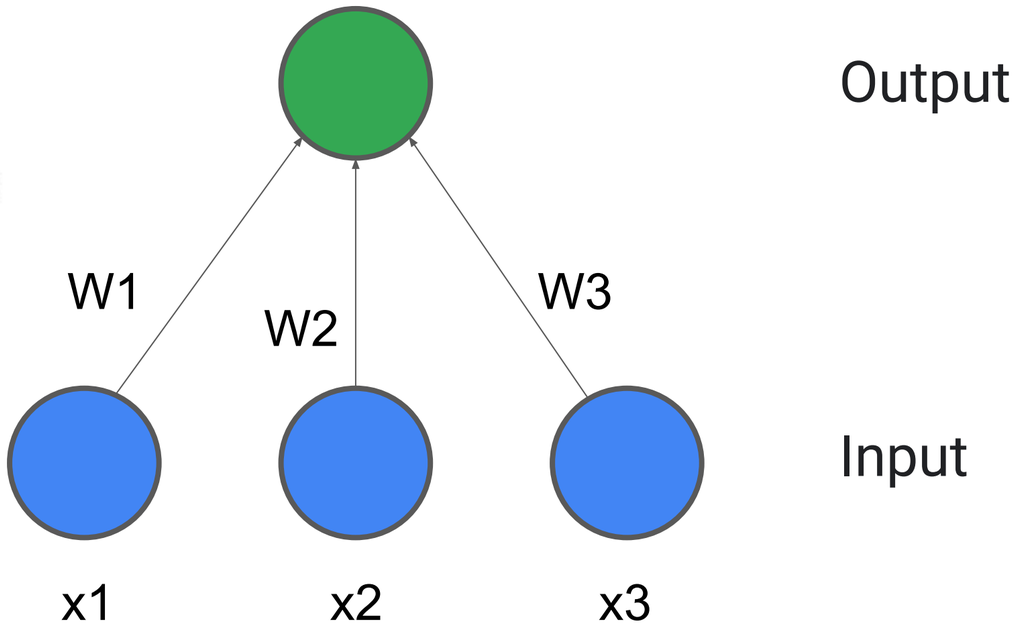
- Activation Functions
- Neural Networks with TF 2 and Keras
- Regularization

Agenda

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

A Linear Model can be represented as nodes and edges

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



Add Complexity: Non-Linear?

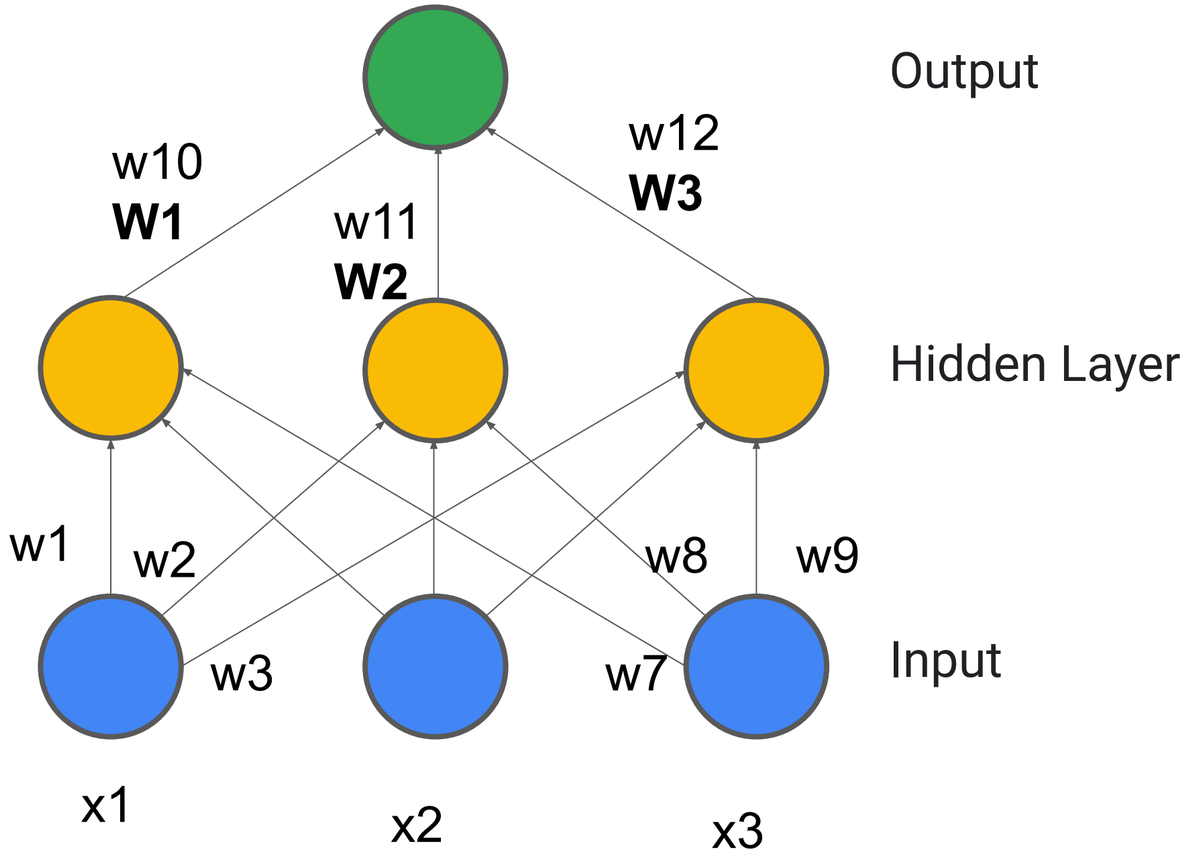
$$output = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

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

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

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

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



Add Complexity: Non-Linear?

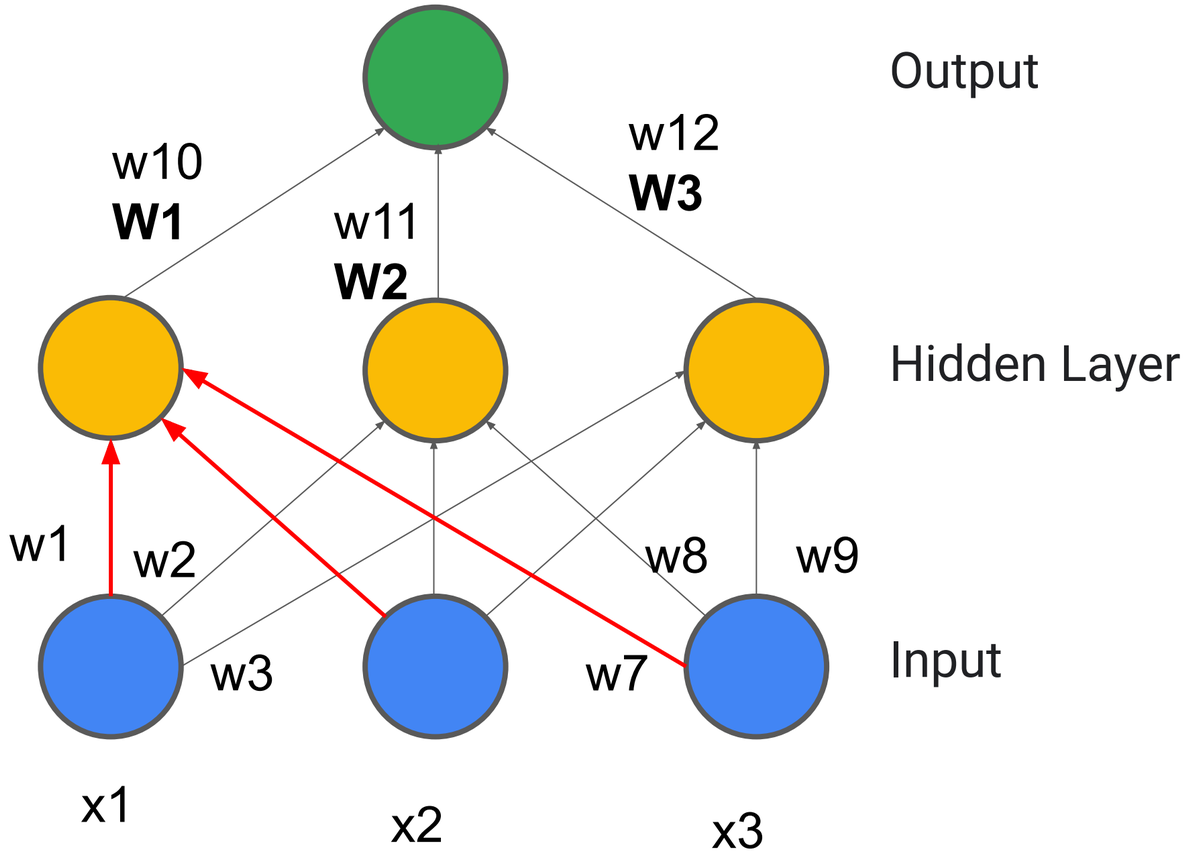
$$output = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

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

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

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

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



Add Complexity: Non-Linear?

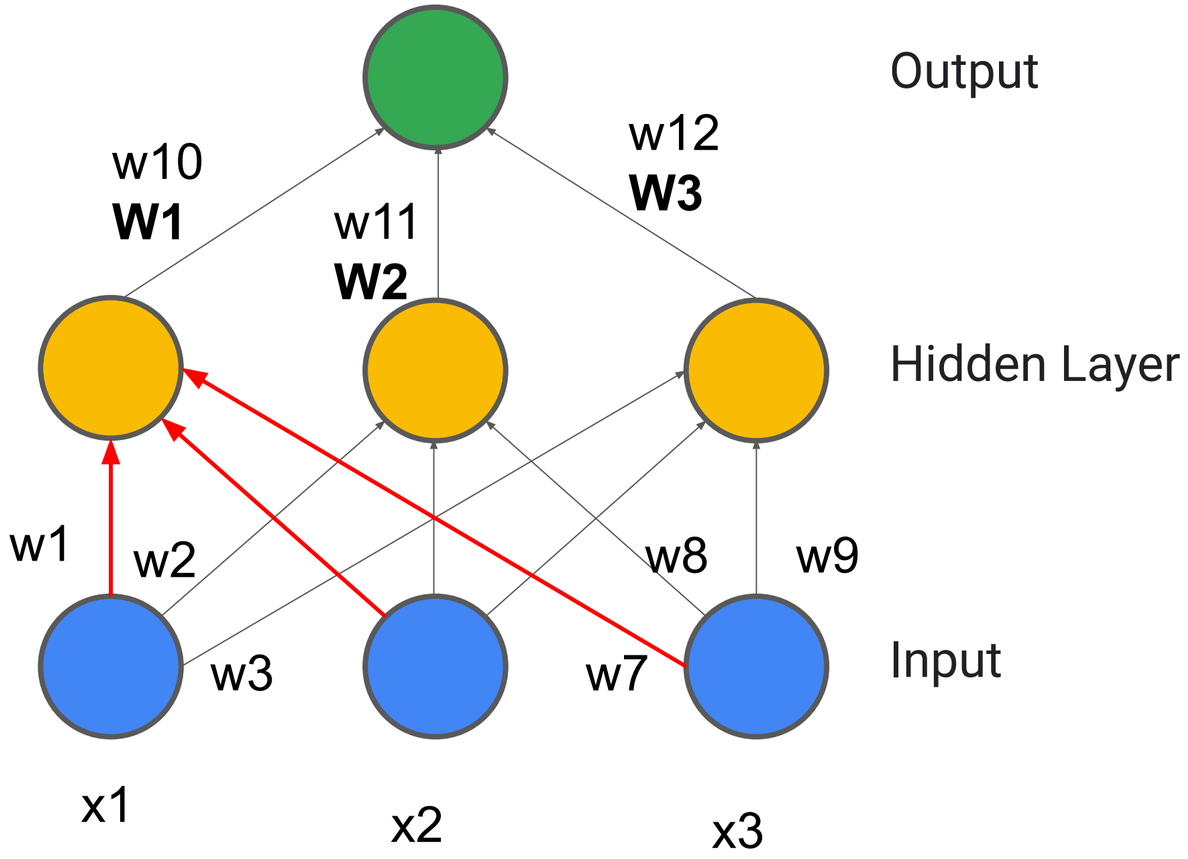
$$output = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

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

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

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

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



Add Complexity: Non-Linear?

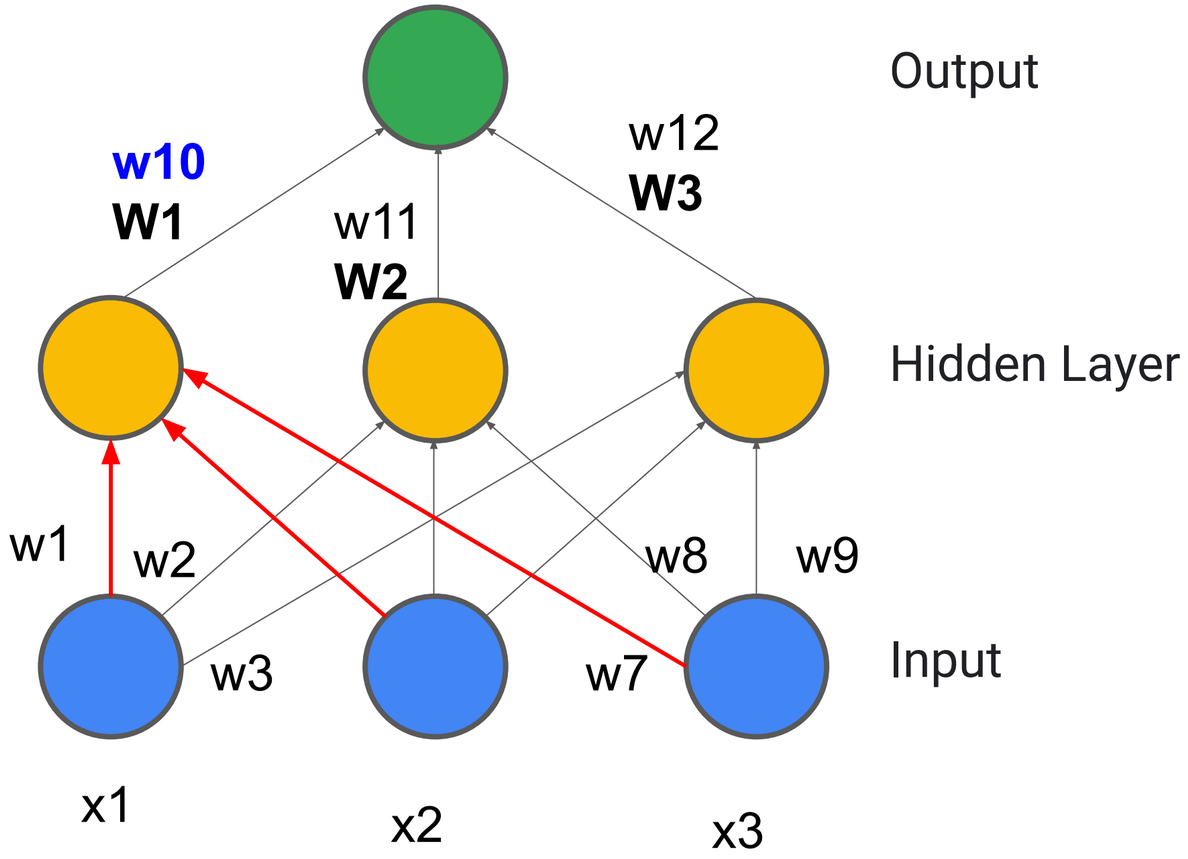
$$output = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

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

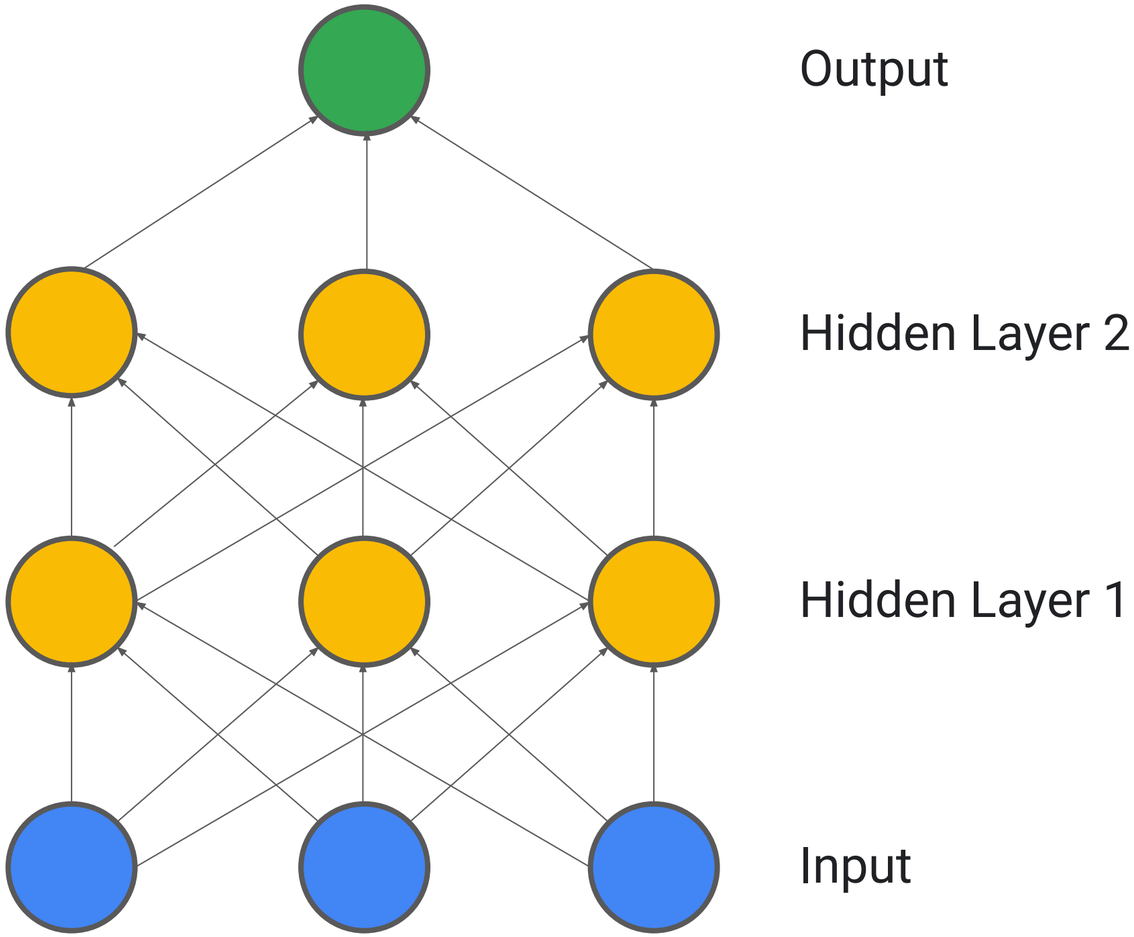
+ ((w11 * w1) + (w11 * w4) + (w11 * w7)) * x2

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

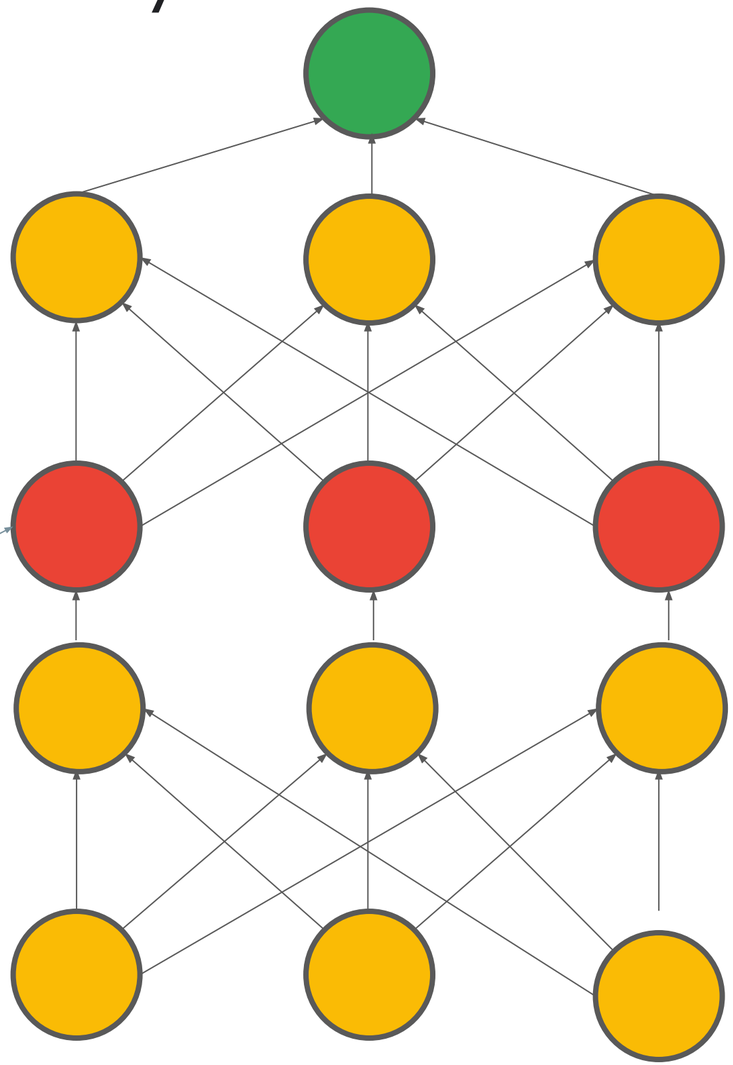
$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



Add Complexity: Non-Linear?



Adding a Non-Linearity



Output

Hidden Layer 2

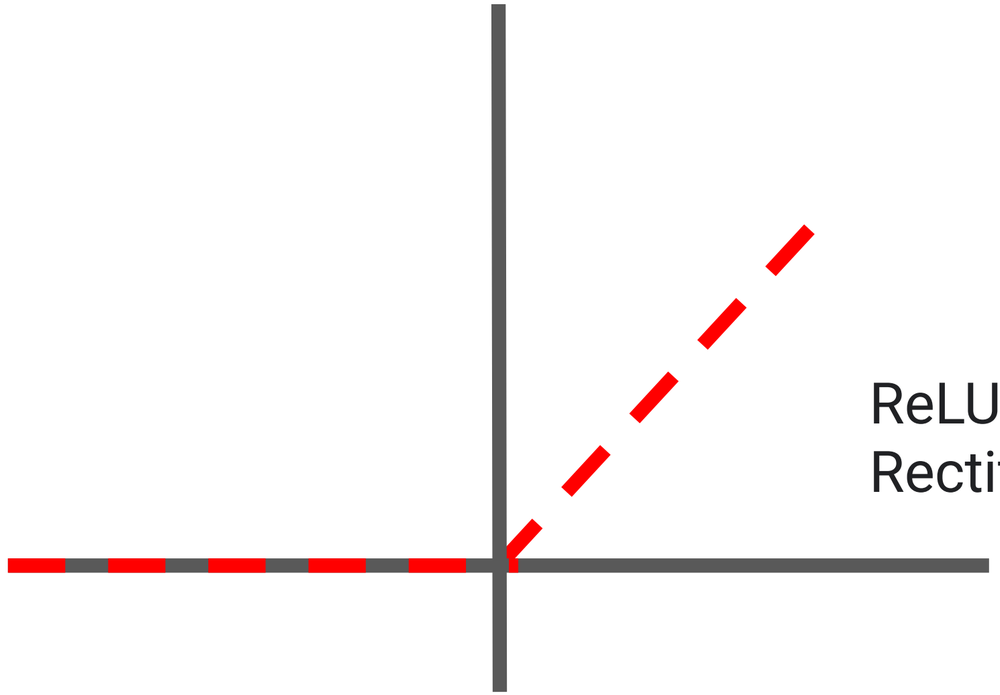
Non-Linear
Transformation Layer
aka Activation Function

Hidden Layer 1

Input

We usually don't draw Non-Linear Transforms.

Our favorite non-linearity is the Rectified Linear Unit

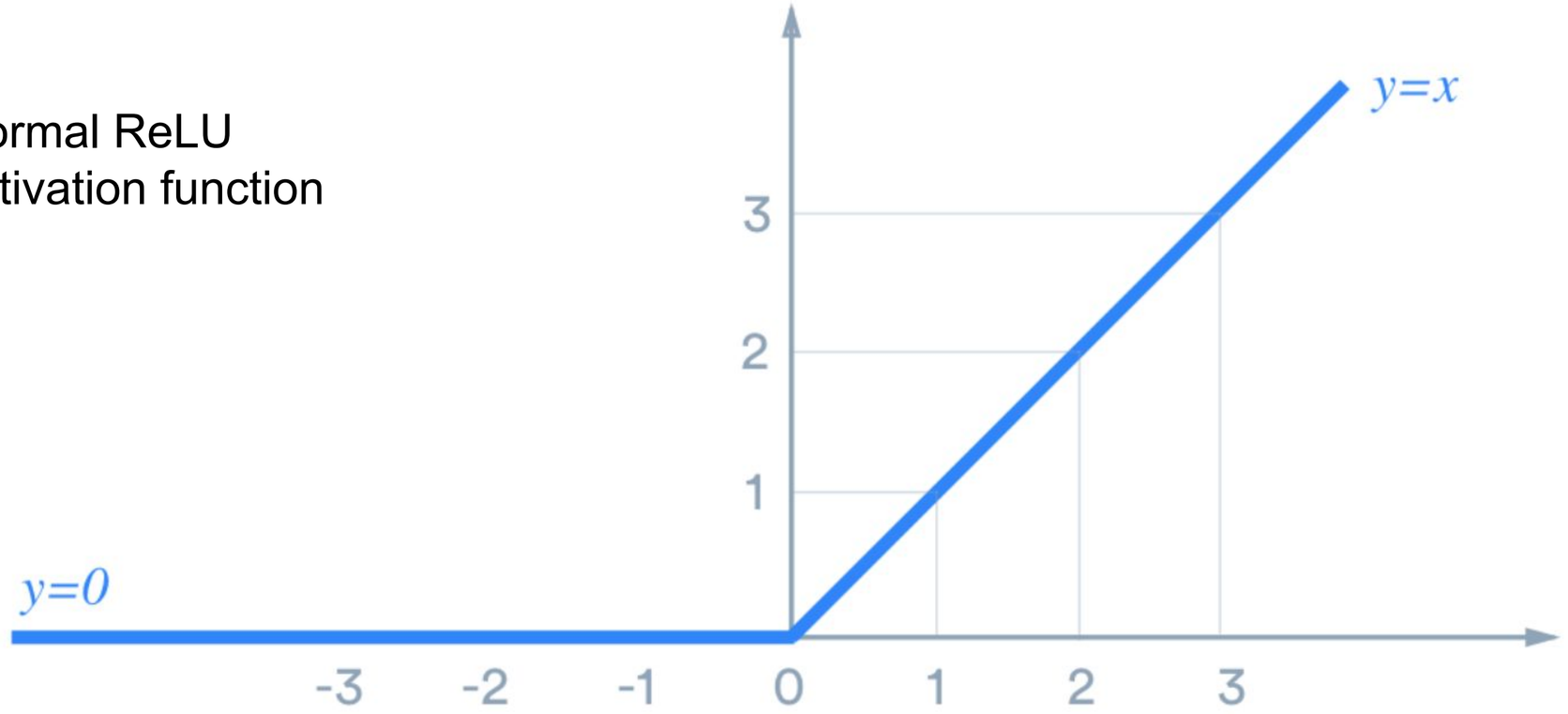


ReLU
Rectified Linear Unit

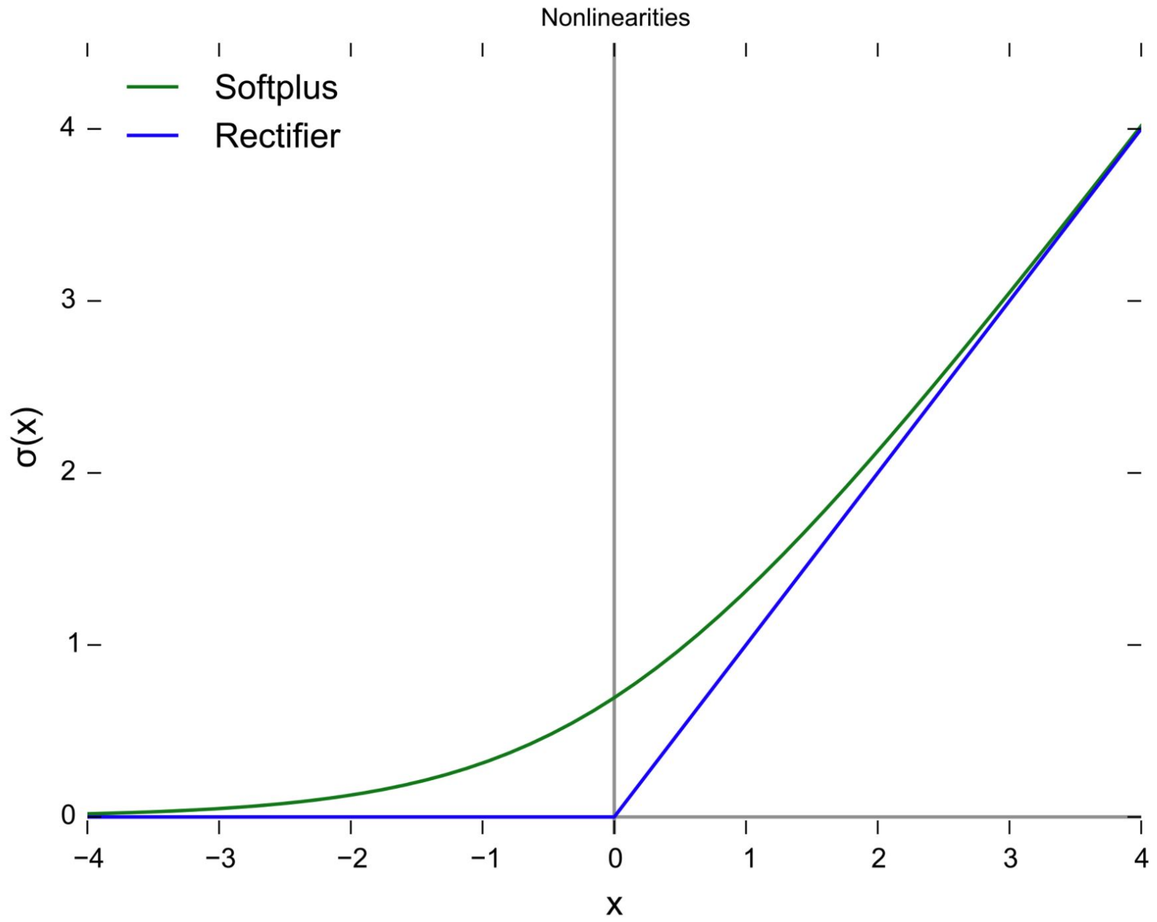
$$f(x) = max(0, x)$$

There are many different ReLU variants

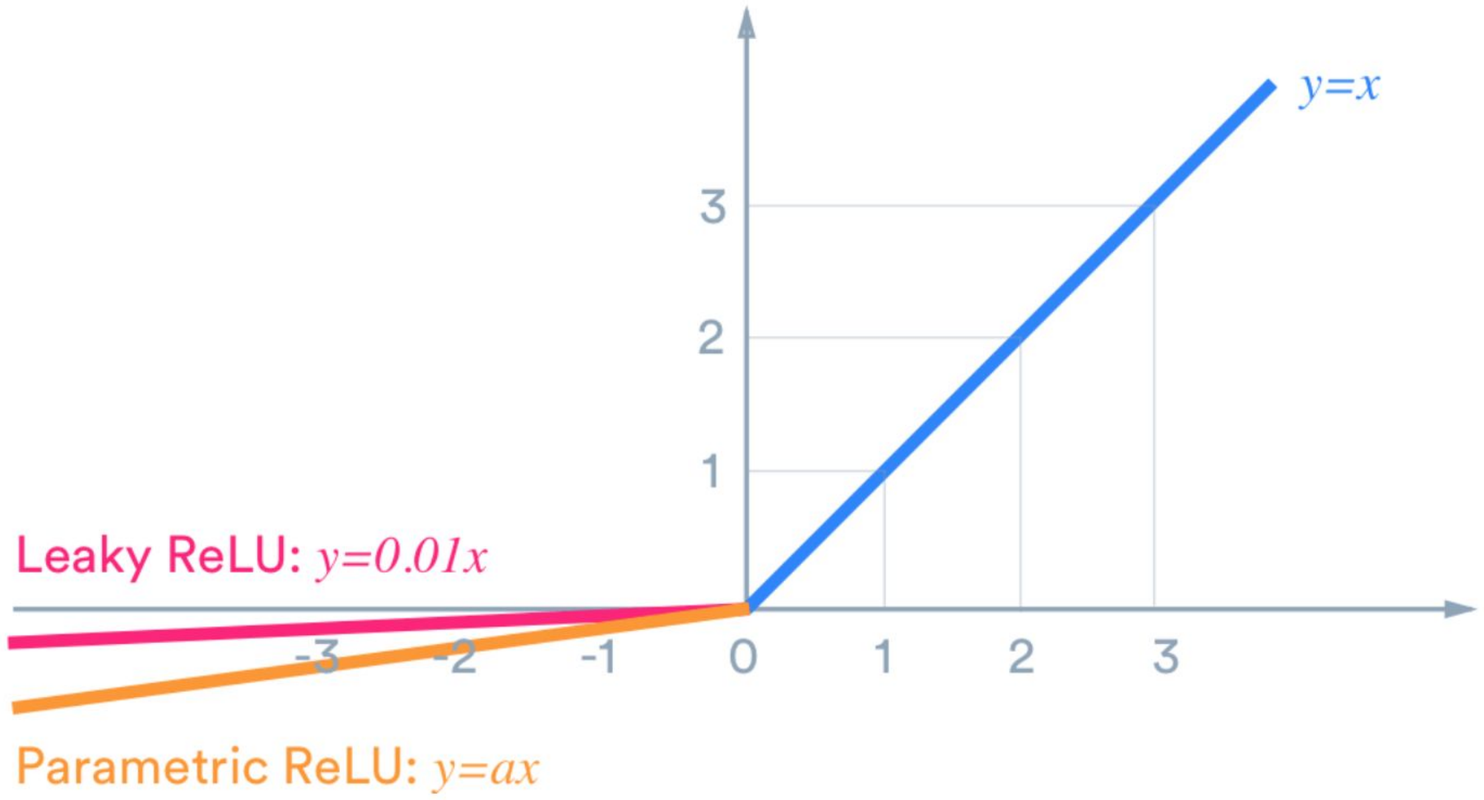
Normal ReLU
activation function



There are many different ReLU variants

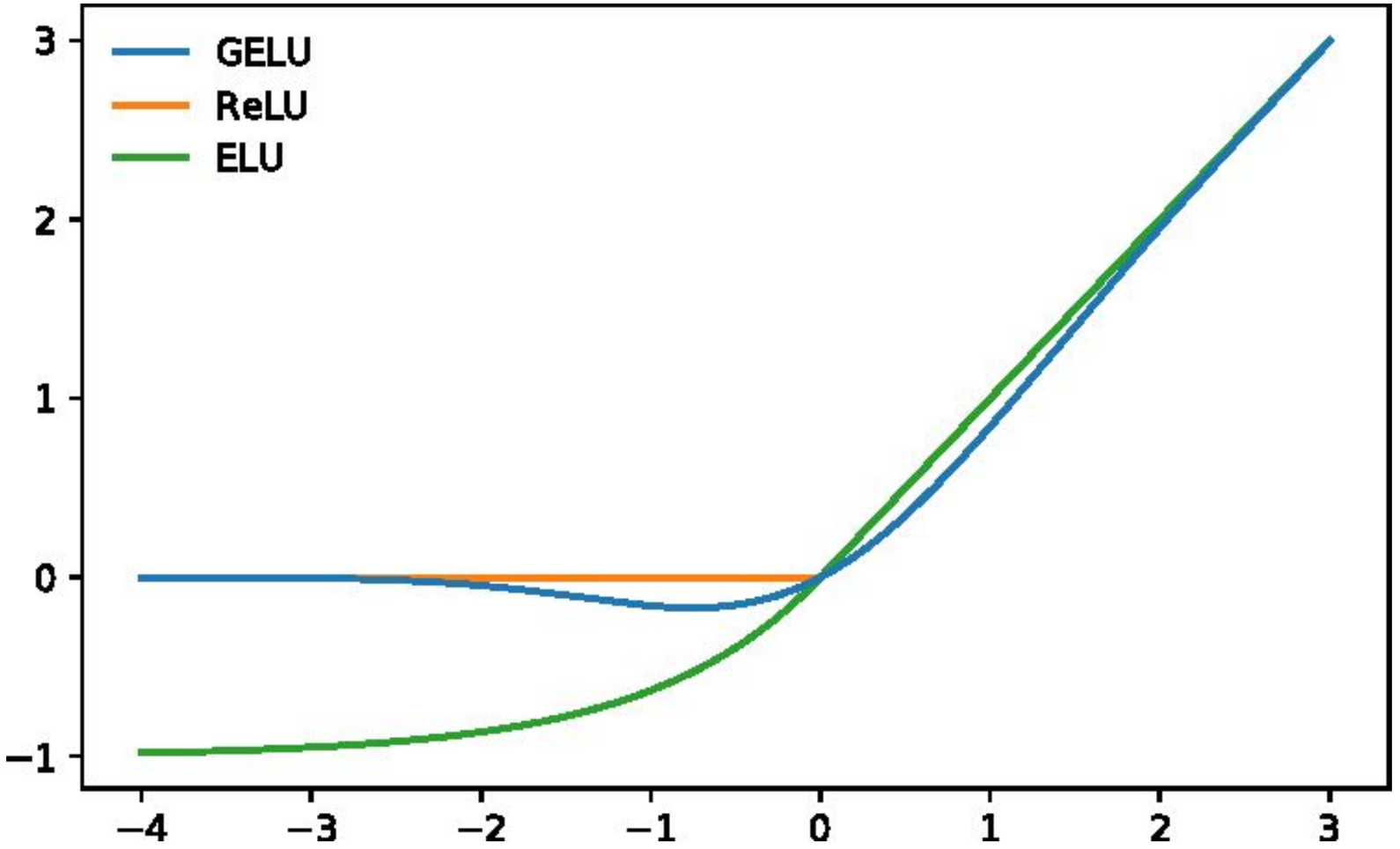


There are many different ReLU variants

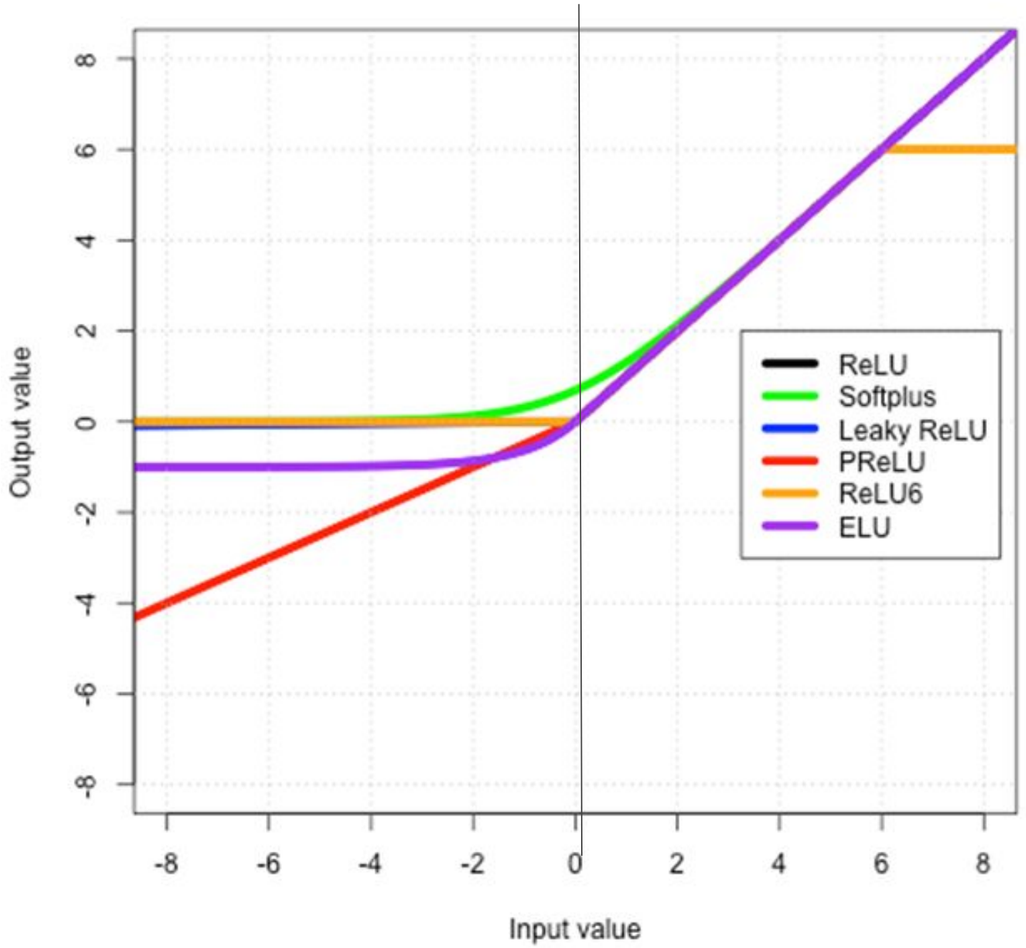


Leaky ReLU

There are many different ReLU variants



There are many different ReLU variants



Three common failure modes for gradient descent

#1 Gradients can vanish — Problem

Each additional layer
can successively reduce
signal vs. noise — Insight

**Using ReLu instead of
sigmoid/tanh can help** — Solution

Three common failure modes for gradient descent

Gradients can vanish	**#2 Gradients can explode**	Problem
Each additional layer can successively reduce signal vs. noise	Learning rates are important here	Insight
Using ReLu instead of sigmoid/tanh can help	**Batch normalization (useful knob) can help**	Solution

Three common failure modes for gradient descent

Gradients can vanish	**Gradients can explode**	**#3 ReLu layers can die**	Problem
Each additional layer can successively reduce signal vs. noise	Learning rates are important here	Monitor fraction of zero weights in TensorBoard	Insight
Using ReLu instead of sigmoid/tanh can help	**Batch normalization (useful knob) can help**	**Lower your learning rates**	Solution

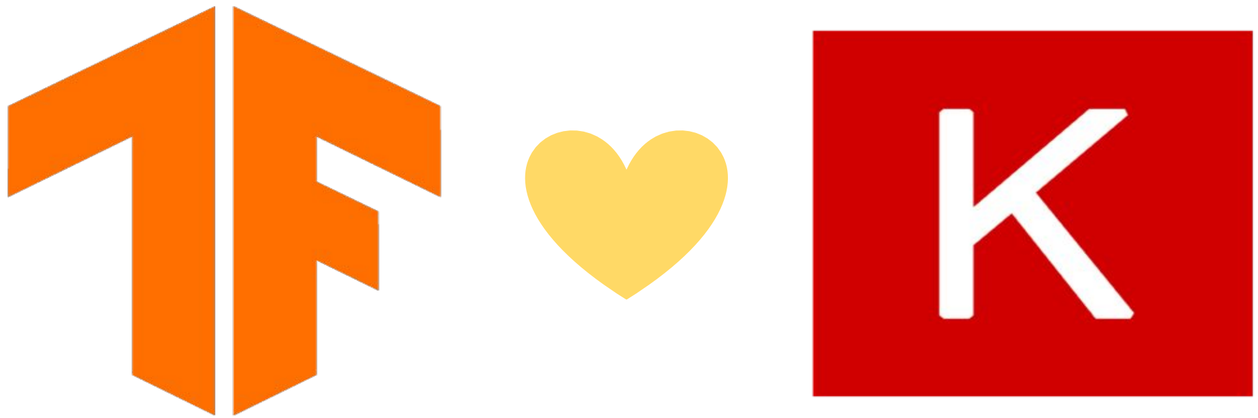
Agenda

Activation Functions

Neural Networks with TF 2 and Keras

Regularization

Keras is built-in to TF 2.x



Stacking layers with Keras Sequential model

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


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

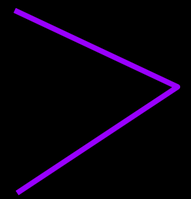
The batch size is omitted. Here the model expects batches of vectors with 64 components.

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

```
%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')
])
```

```
%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()
```

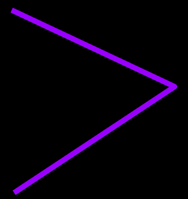
```
# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(10, activation='softmax')
])
```



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

```
%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()
```

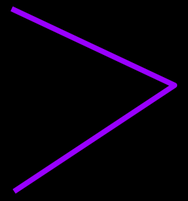
```
# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```



A neural network with one hidden layer

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

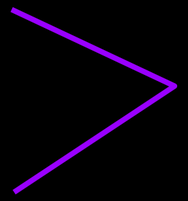
```
# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```



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

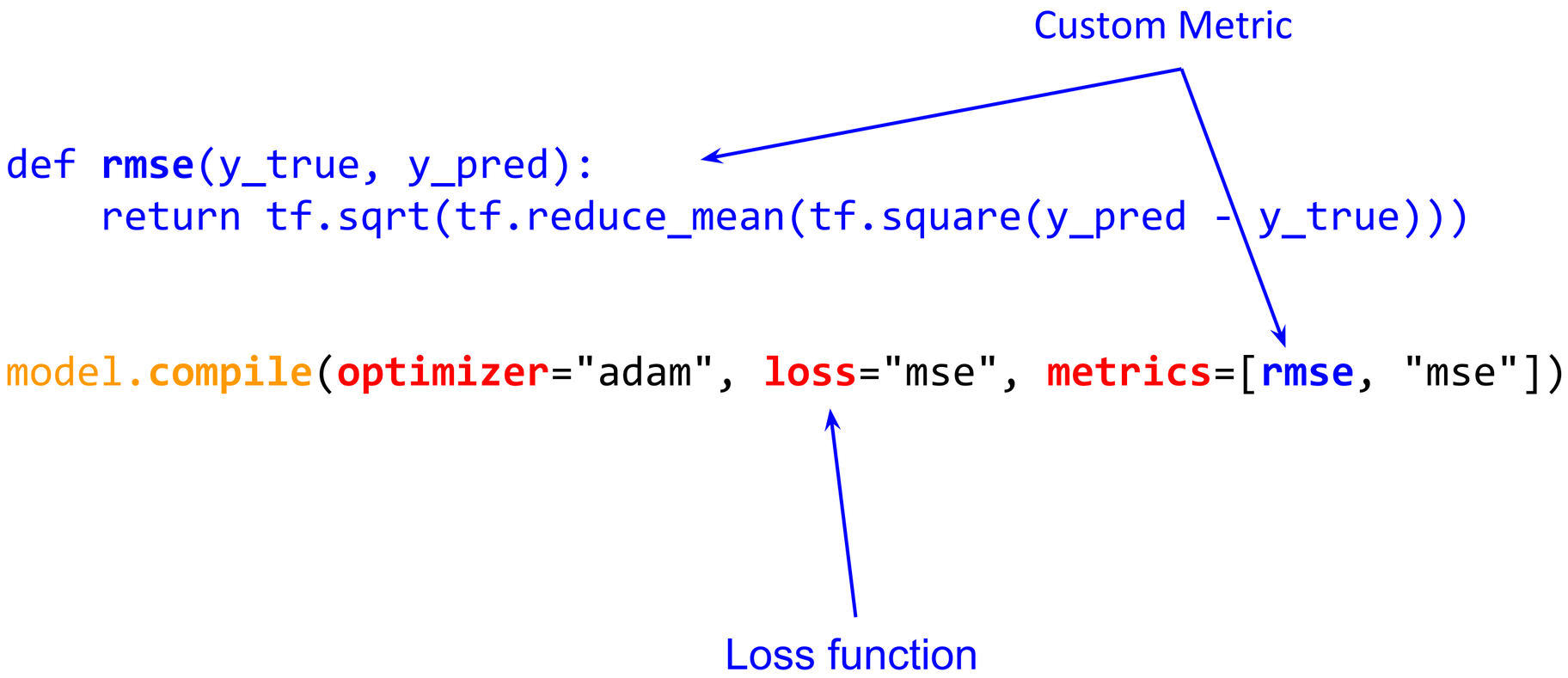
```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

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



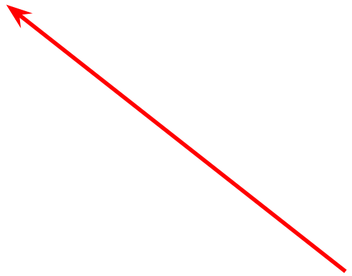
A deeper neural network

Compiling a Keras model



Compiling a Keras model

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


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



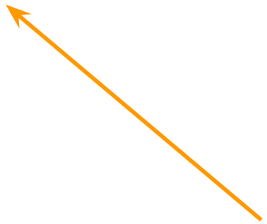
Optimizer

Compiling a Keras model

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


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

Training a Keras model

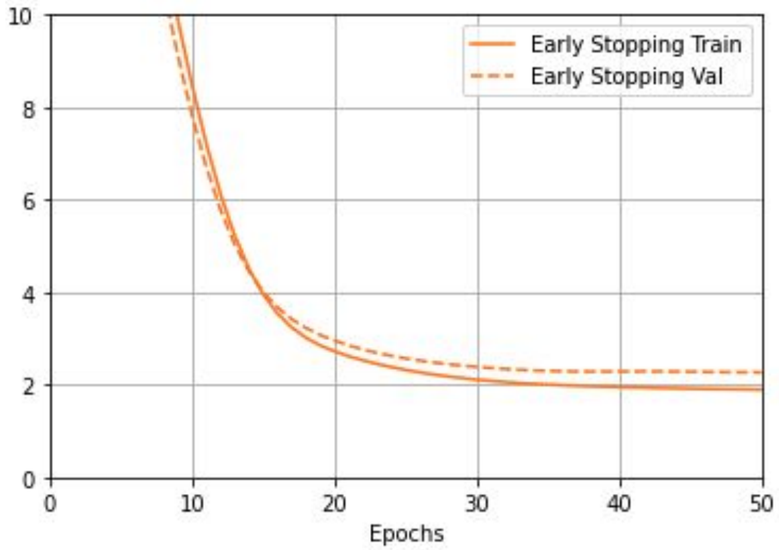
```
from tensorflow.keras.callbacks import TensorBoard

steps_per_epoch = NUM_TRAIN_EXAMPLES // (TRAIN_BATCH_SIZE * NUM_EVALS)

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



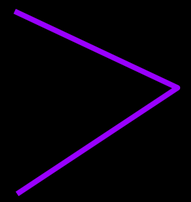
This is a trick so that we have control on the total number of examples the model trains on (NUM_TRAIN_EXAMPLES) and the total number of evaluation we want to have during training (NUM_EVALS).

Training a Keras model

```
from tensorflow.keras.callbacks import TensorBoard

steps_per_epoch = NUM_TRAIN_EXAMPLES // (TRAIN_BATCH_SIZE * NUM_EVALS)

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



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

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

# Configure and train
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```



A deeper neural network

Once trained, the model can be used for prediction

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

returns a Numpy array of predictions

With the `predict()` method you can pass

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

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

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

SavedModel is the universal serialization format for Tensorflow models

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

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

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

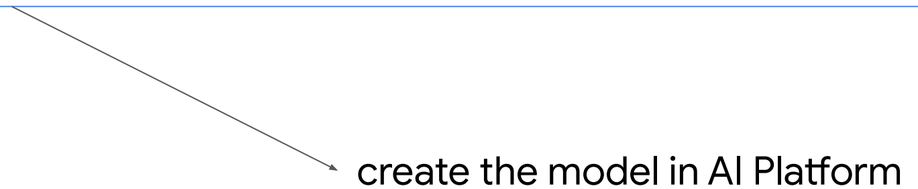
the directory in which to write the SavedModel

a trackable object such as a trained keras model

exports a model object to a SavedModel format

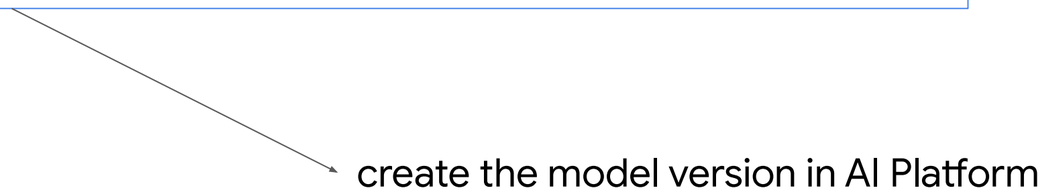
Create a model object in Cloud AI Platform

```
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

...
```



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
```



Deploy SavedModel using `gcloud ai-platform`

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

specify the model name and version

`EXPORT_PATH` denotes location of SavedModel directory

Make predictions using `gcloud ai-platform`

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

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

specify the name and version of the deployed model

json file for prediction

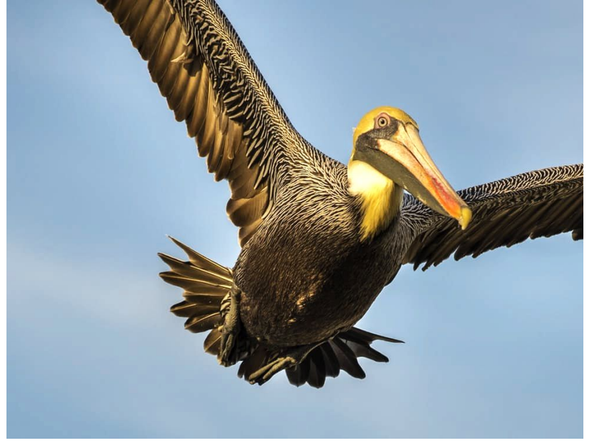
Google Cloud

DNNs with the Keras Functional API

Seagulls
can fly.

Pigeons
can fly.

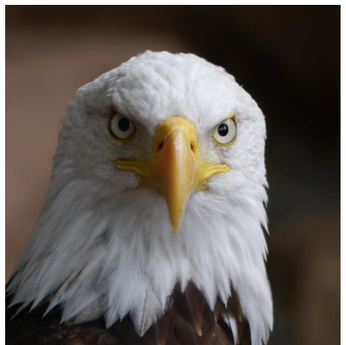










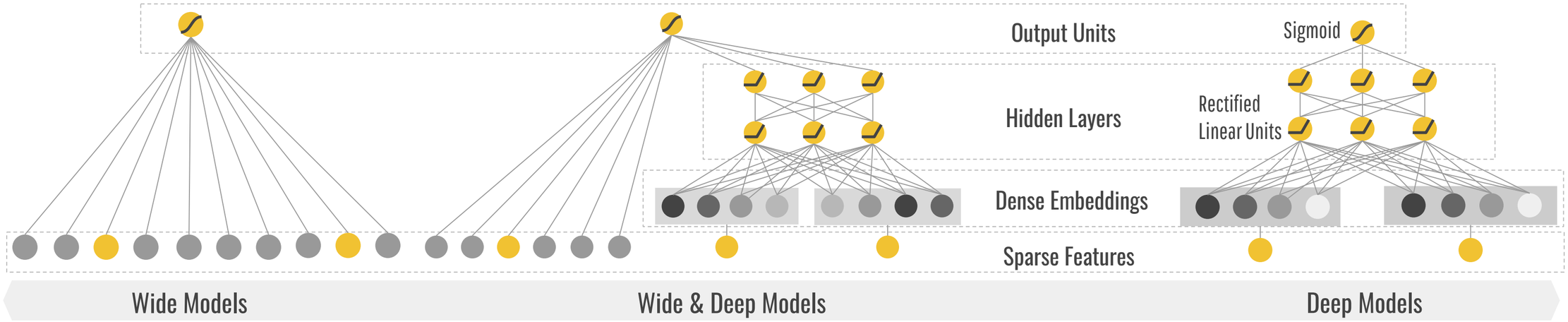



Animals with wings can fly.

Penguins...

Usings Wide and Deep learning

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



Memorization + Generalization

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

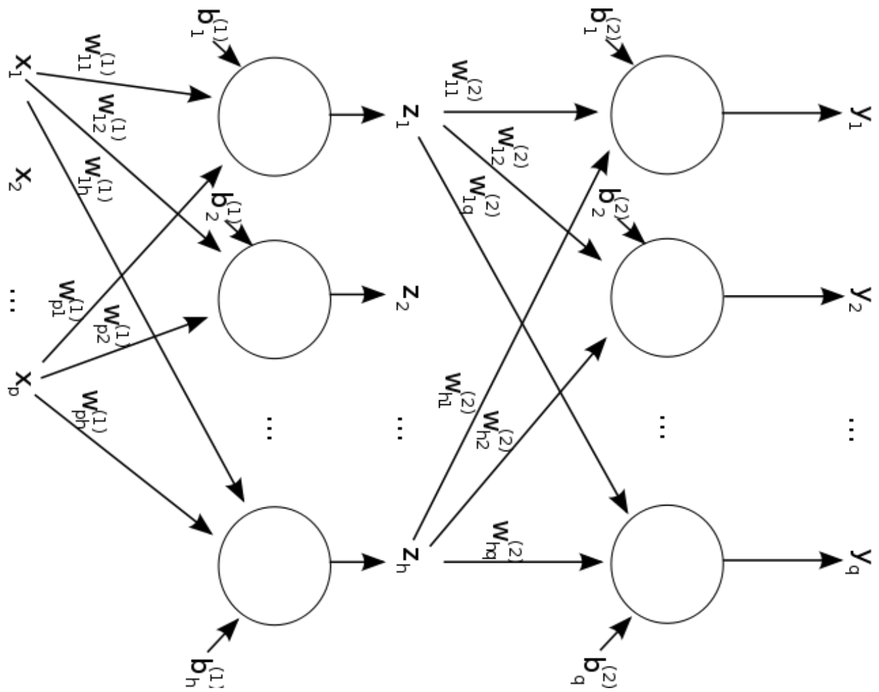
Linear models are good for sparse, independent features

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

DNNs are good for dense, highly correlated features

pixel_values (

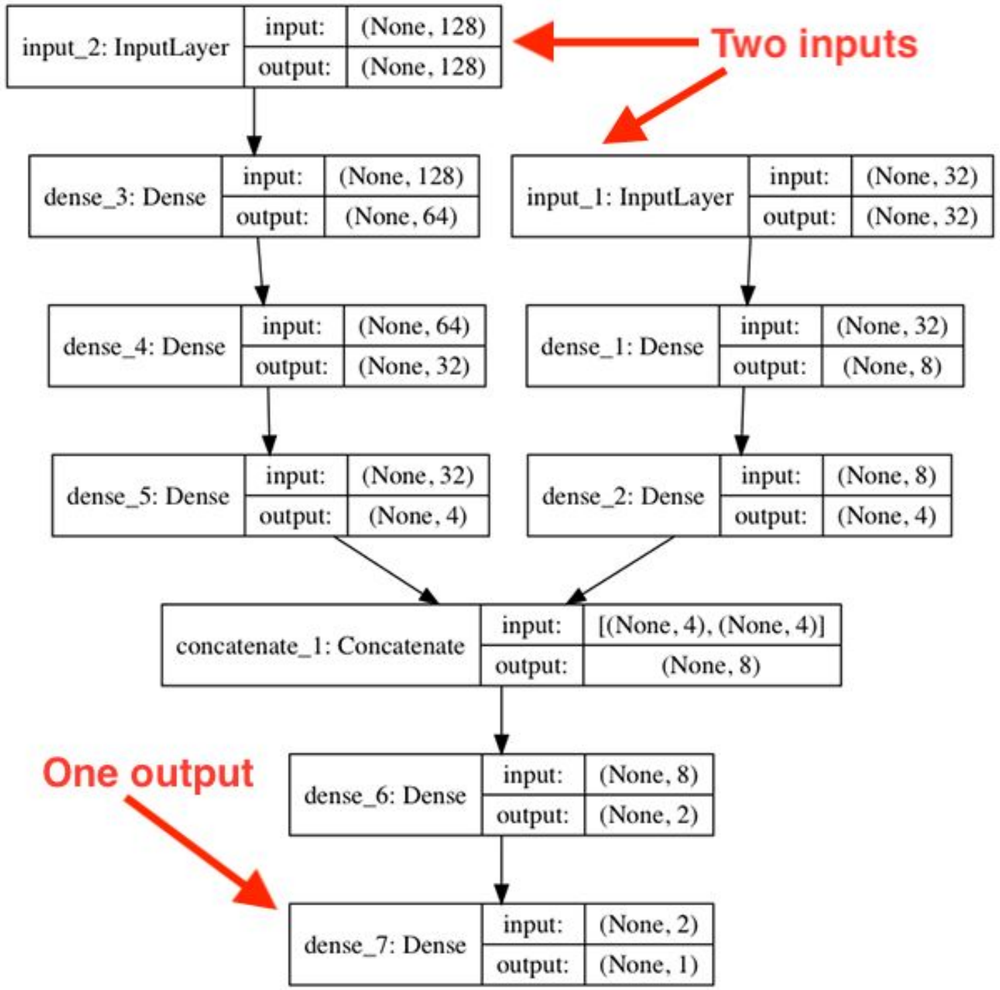
)



1024^2 input nodes

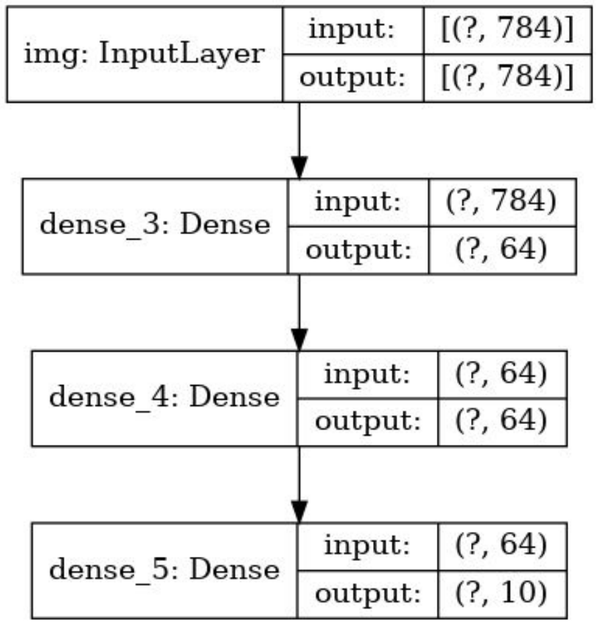
10 hidden nodes -> 10 image features

Wide and deep networks using Keras Functional API



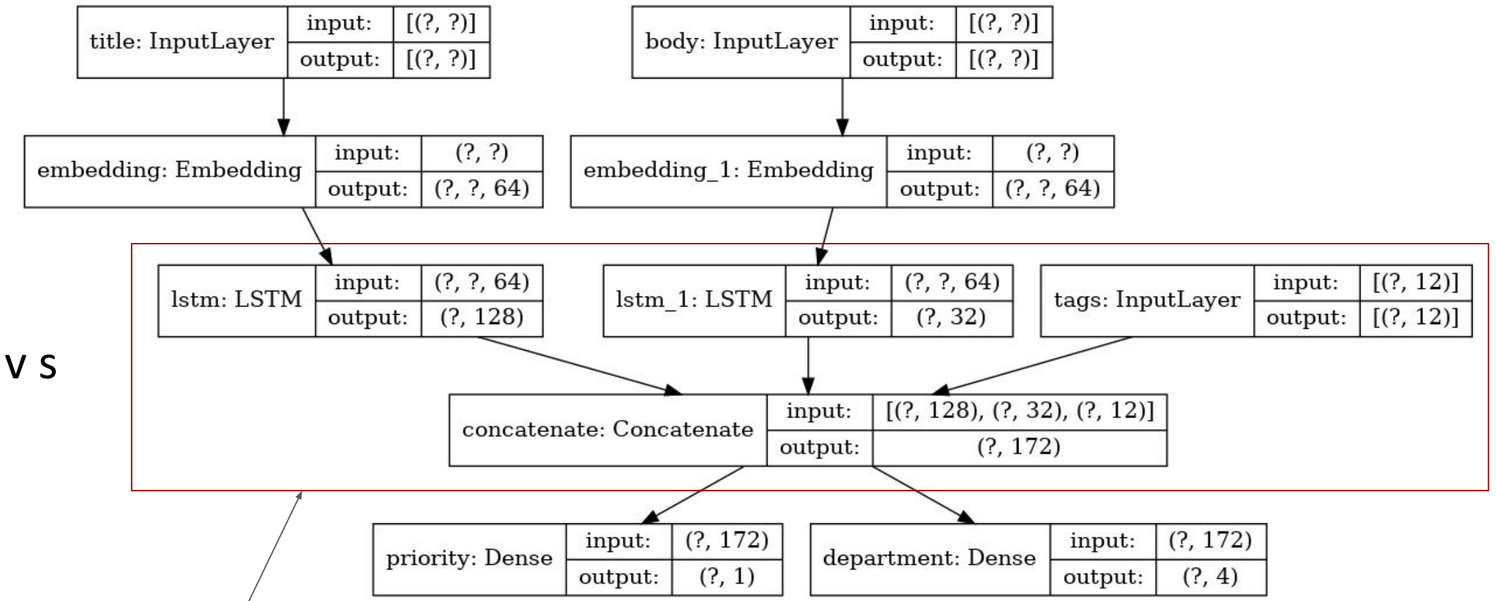
Functional API is more flexible than Sequential API

Sequential model in Keras



v s

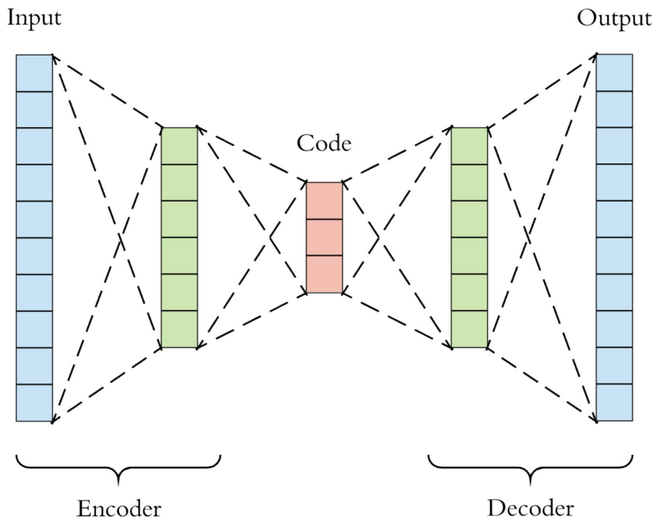
Functional model in Keras



Multiple inputs / shared layer

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

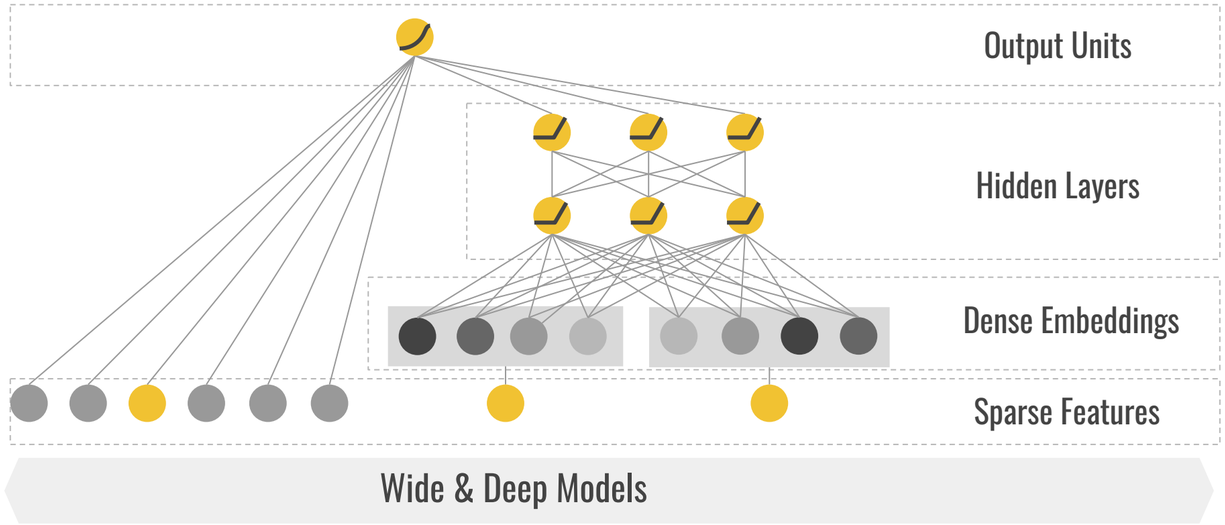
```
encoder_input = keras.Input(shape=(28, 28, 1), name='img')
x = layers.Dense(16, activation='relu')(encoder_input)
x = layers.Dense(10, activation='relu')(x)
x = layers.Dense(5, activation='relu')(x)
encoder_output = layers.Dense(3, activation='relu')(x)

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




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

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

Creating a Wide and Deep model in Keras

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

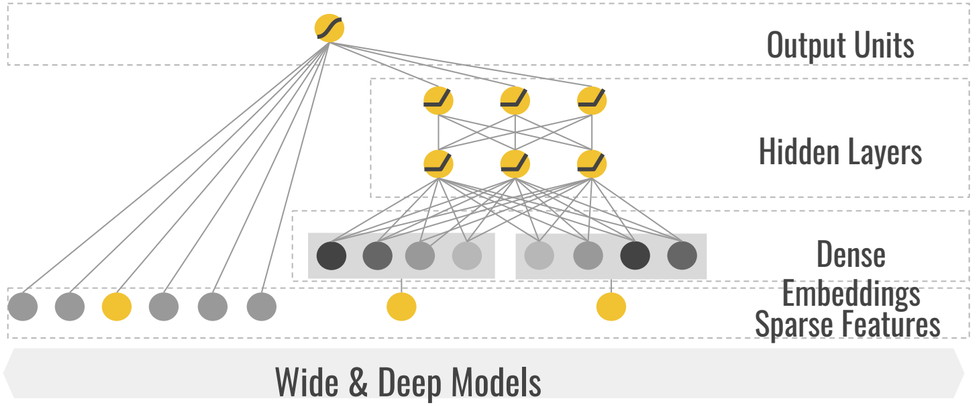



```
# Prepare input feature columns
inputs = {colname : layers.Input(name=colname, shape=(), dtype='float32')
          for colname in INPUT_COLS
}
```

...

Creating a Wide and Deep model in Keras

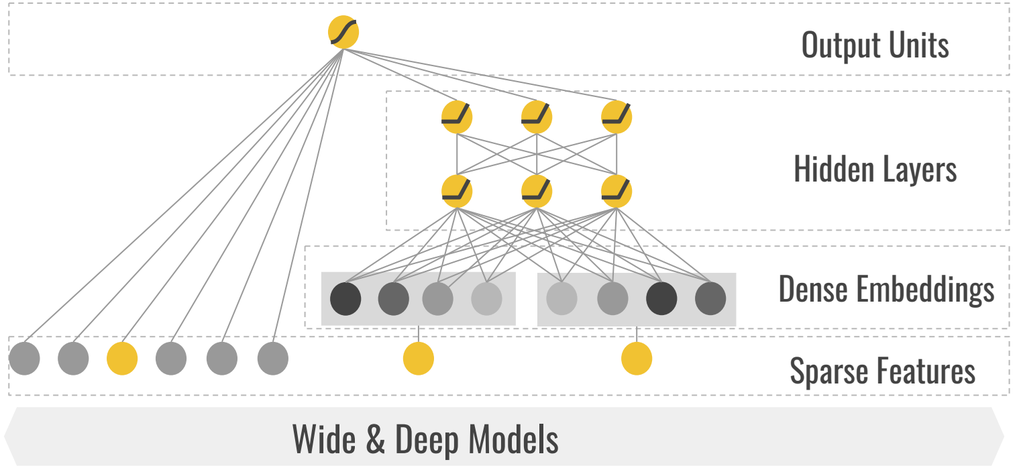
```
# Create deep columns
deep_columns = [
    # Embedding_column to "group" together ...
    fc.embedding_column(fc_crossed_pd_pair, 10),

    # 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

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)
]
```

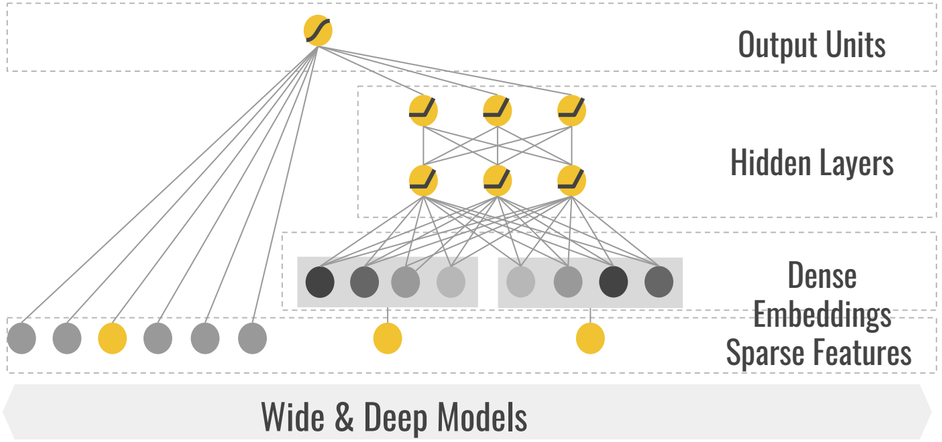



```
# Create the wide part of model

wide = layers.DenseFeatures(wide_columns, name='wide_inputs')(inputs)
```

created in the previous slide

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

Strengths and weaknesses of the Functional API

Strengths

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

Strengths and weaknesses of the Functional API

Strengths

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

Weaknesses

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

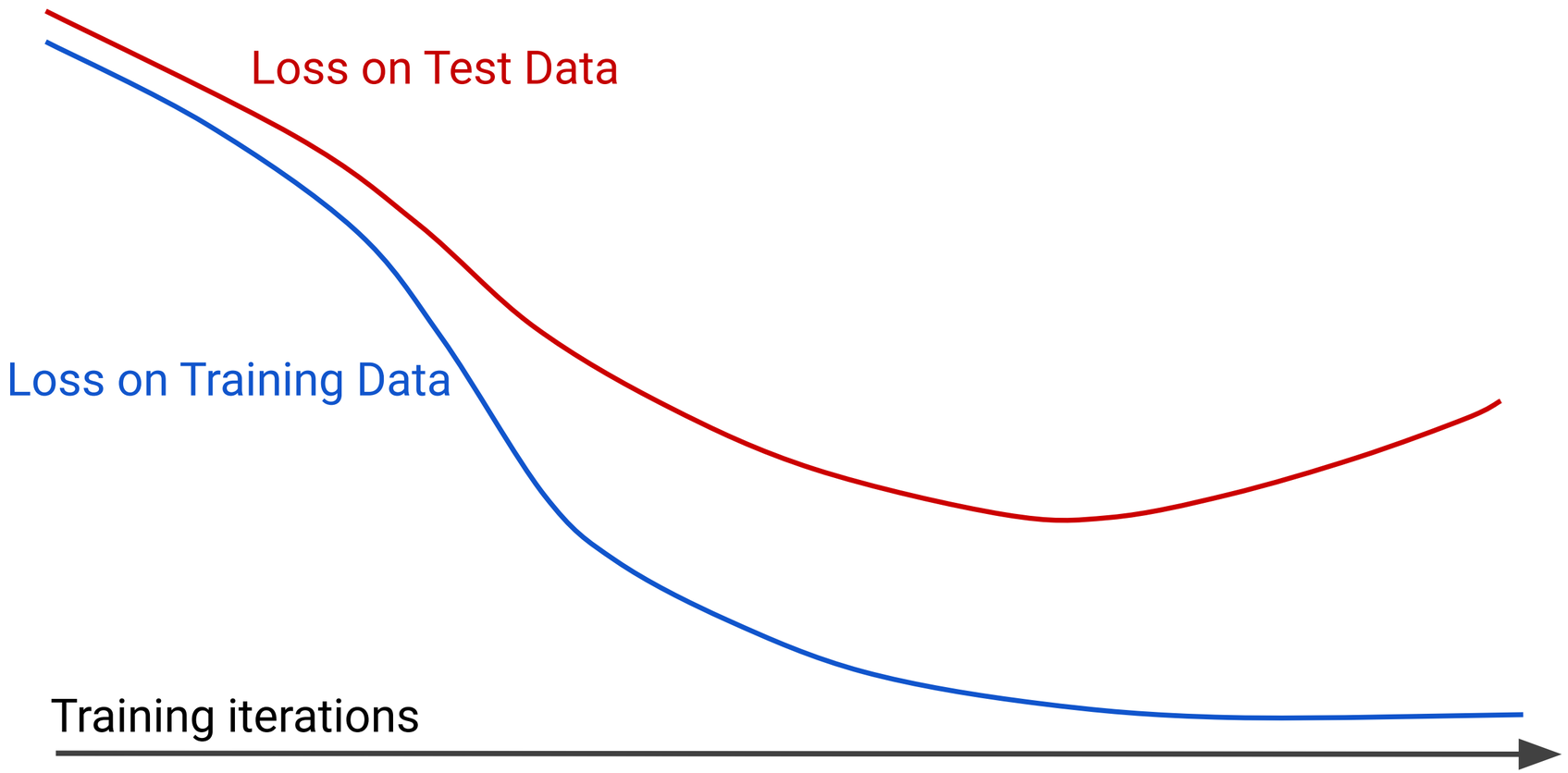
Agenda

Activation Functions

Neural Networks with TF 2 and Keras

Regularization

What's happening here? How can we address this?



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?

The simpler the better

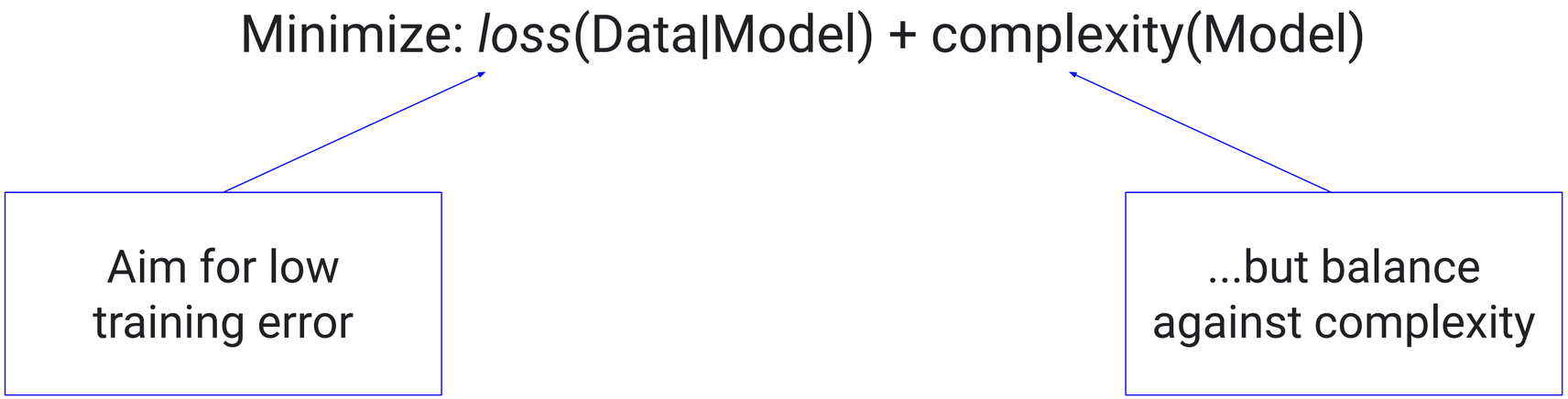
Don’t cook with every spice in the spice rack!

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

Factor in model complexity when calculating error



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

Regularization is a major field of ML research

Early Stopping

Parameter Norm Penalties

- L1 regularization
- L2 regularization
- Max-norm regularization

Dataset Augmentation

Noise Robustness

Sparse Representations

. . .

We'll look into these methods.

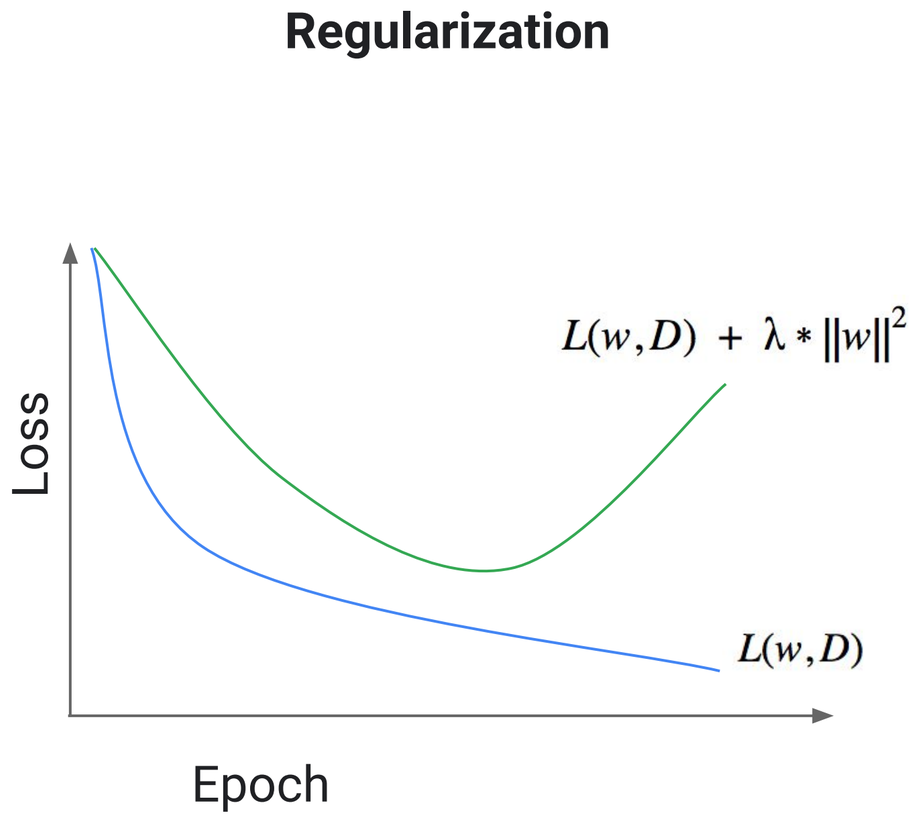
Zeroing out coefficients can help with performance, especially with large models and sparse inputs

Action	Impact
Fewer coefficients to store/load.	Reduce memory, model size.
Fewer multiplications needed.	Increase prediction speed.

$$L(w, D) + \lambda \sum^{n} |w|$$

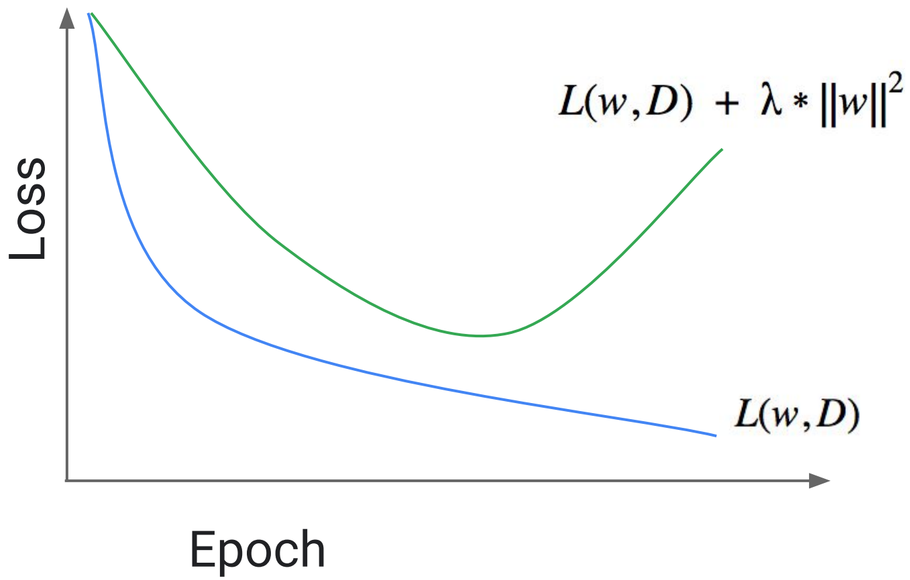
L2 regularization only makes weights small, not zero.

Often we do both regularization and early stopping to counteract overfitting

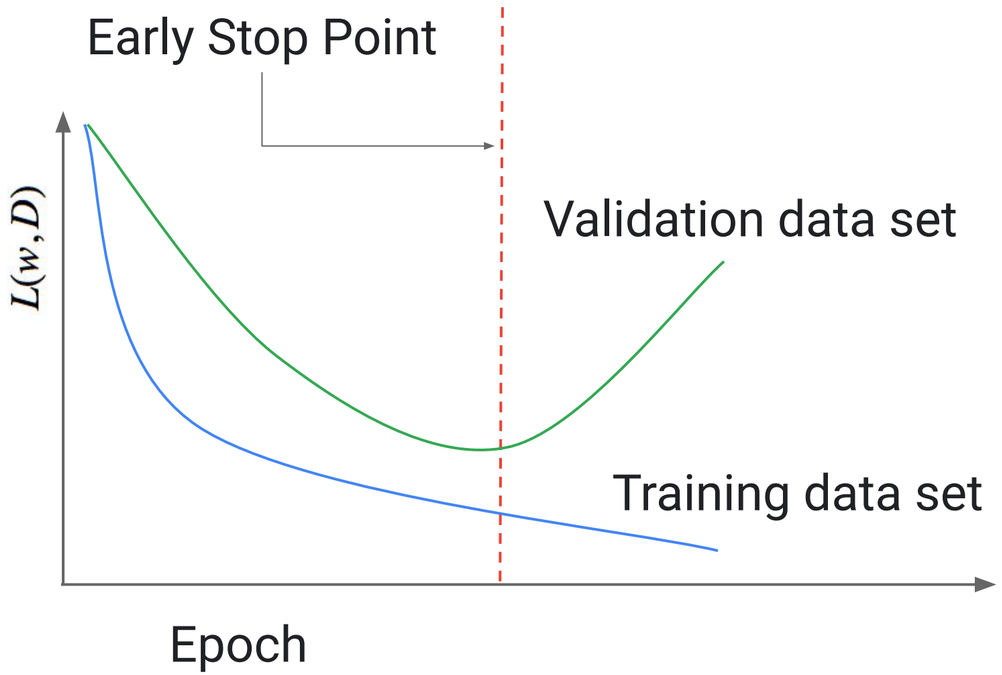


Often we do both regularization and early stopping to counteract overfitting

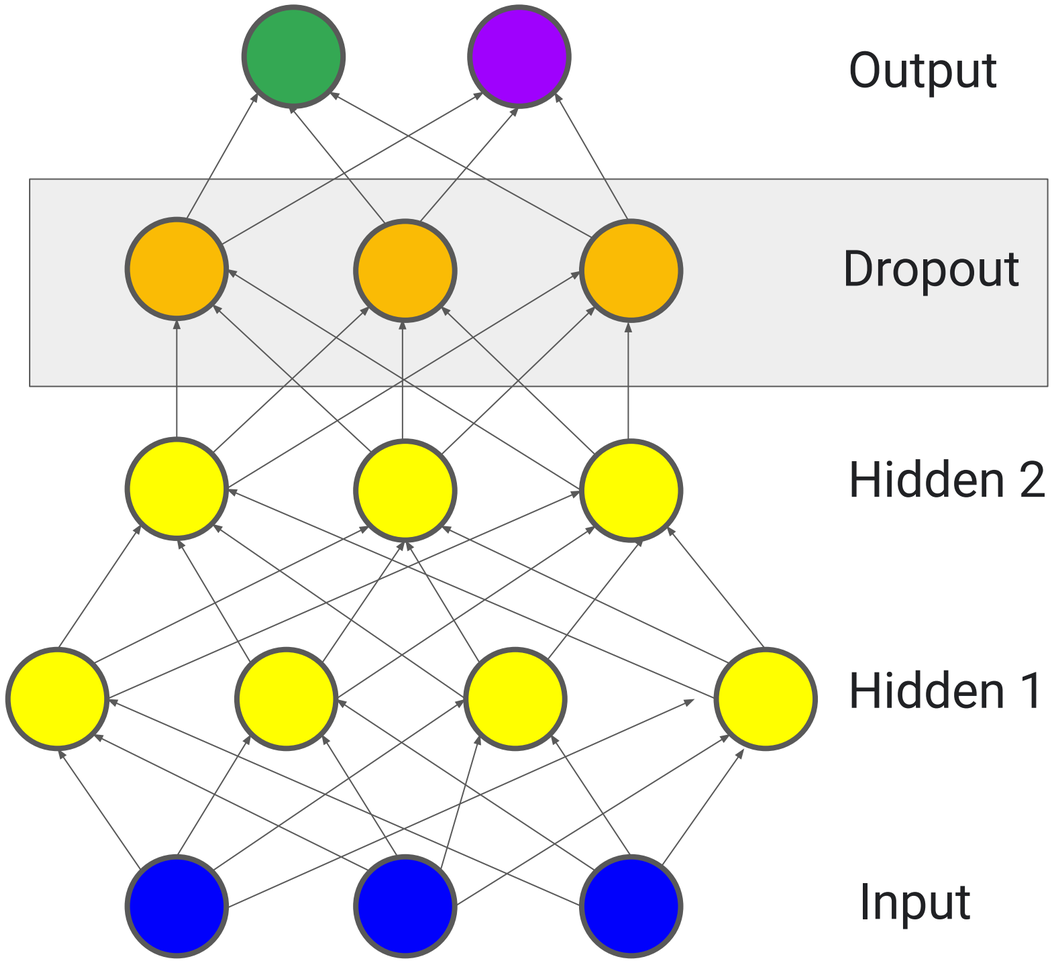
Regularization



Early stopping

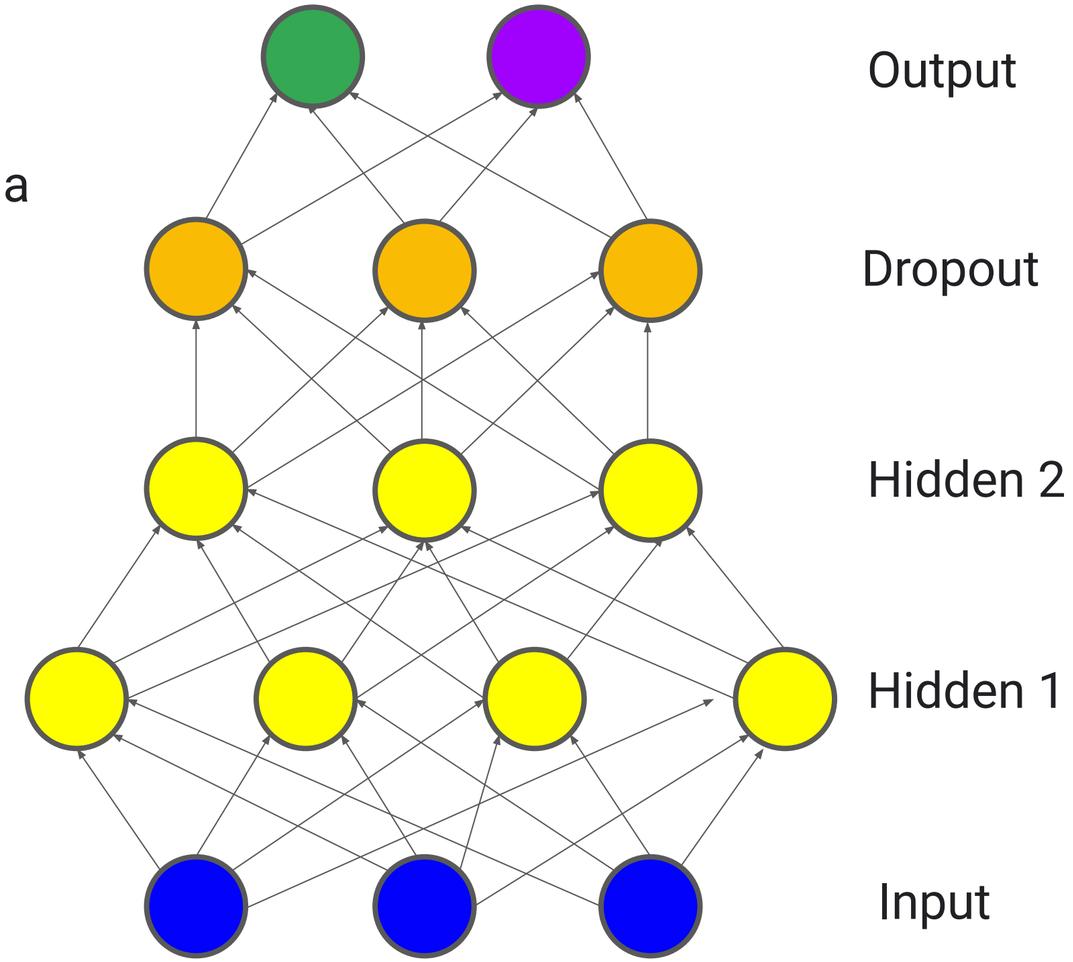


Dropout layers are a form of regularization



Dropout layers are a form of regularization

Dropout works by randomly “dropping out” unit activations in a network for a single gradient step.

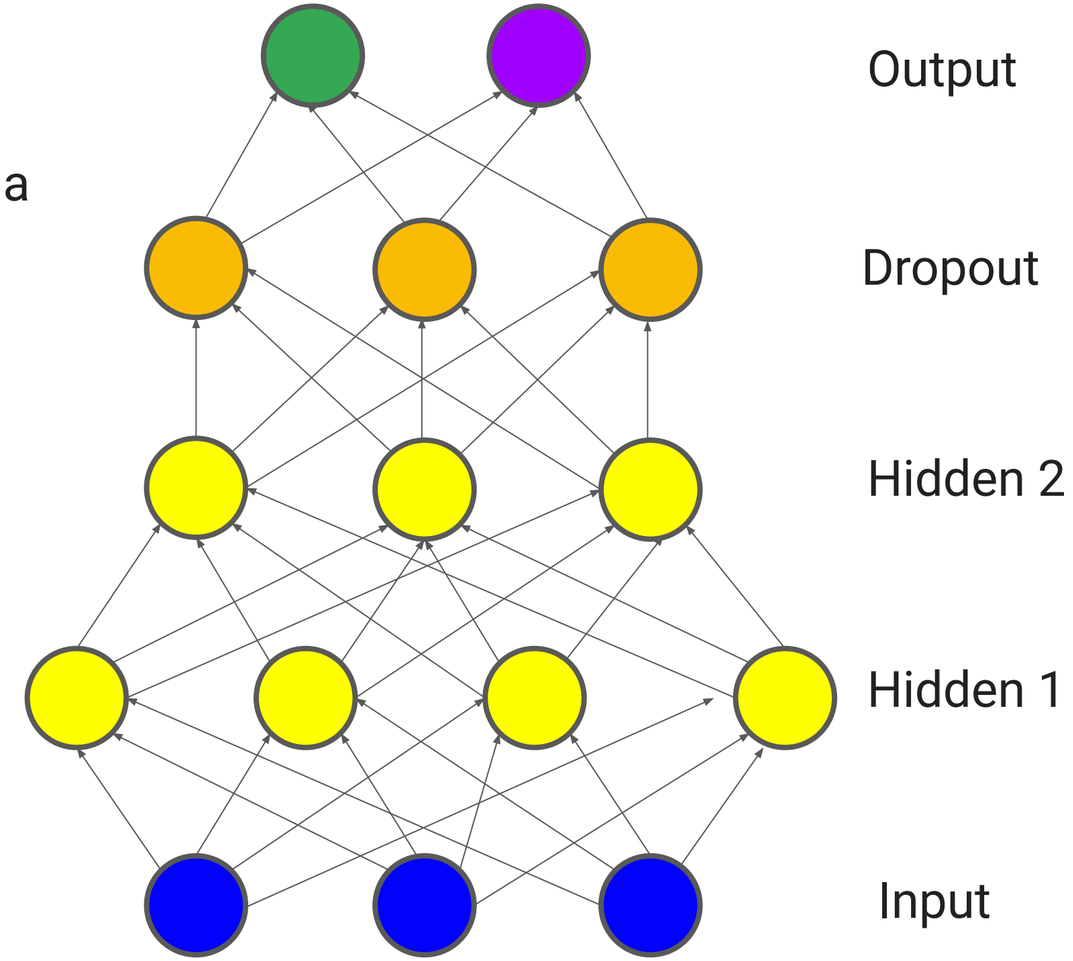


Dropout layers are a form of regularization

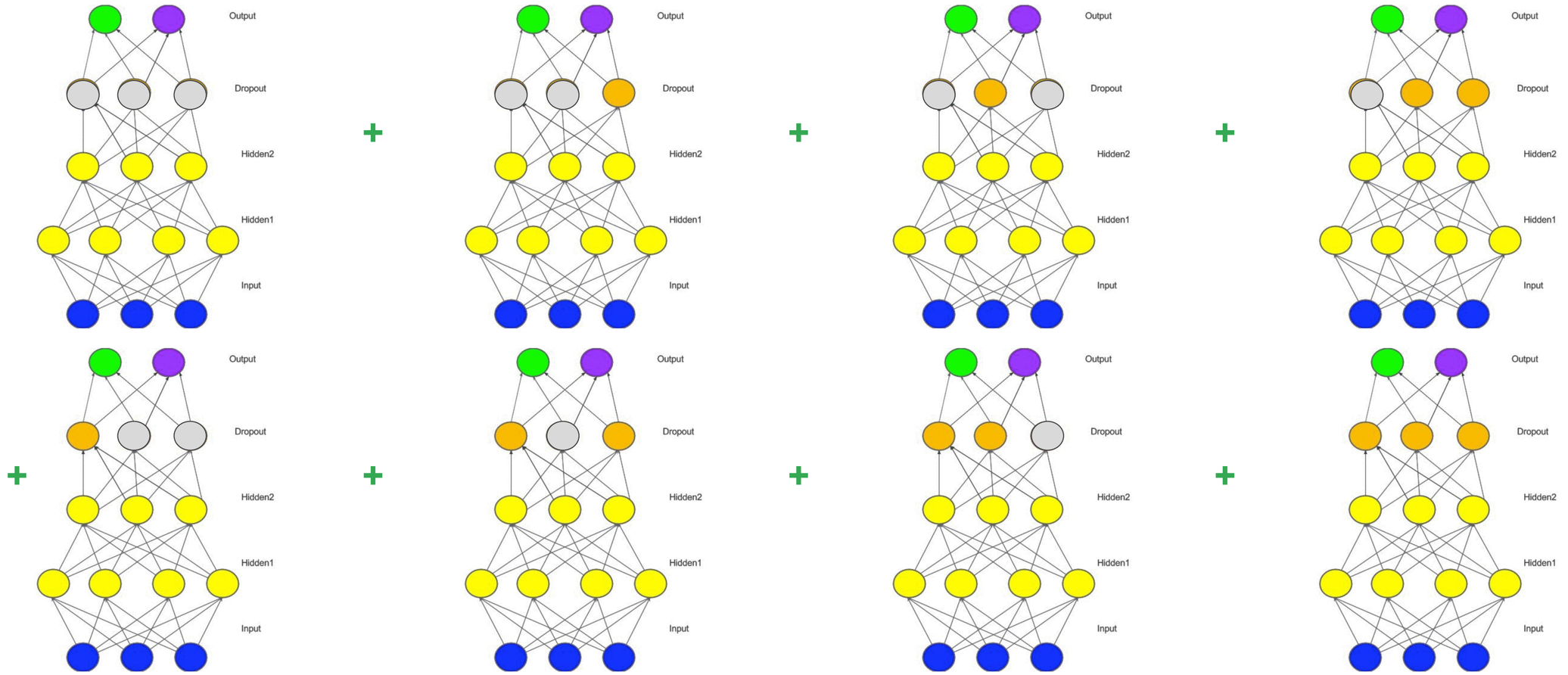
Dropout works by randomly “dropping out” unit activations in a network for a single gradient step.

During training only! In prediction all nodes are kept.

Helps learn “multiple paths” -- think: ensemble models, random forests.



Dropout simulates ensemble learning



The more you drop out, the stronger the regularization

0.0 = no dropout regularization

Intermediate values more useful, a value of dropout=0.2 is typical

The more you drop out, the stronger the regularization

0.0 = no dropout regularization

0.0

Intermediate values more useful, a value of dropout=0.2 is typical

1.0 = drop everything out! learns nothing

1.0

The more you drop out, the stronger the regularization