Introduction to Tensorflow in Python

Monday, 10 August 2020 3:45 PM

Introduction to Tensorflow

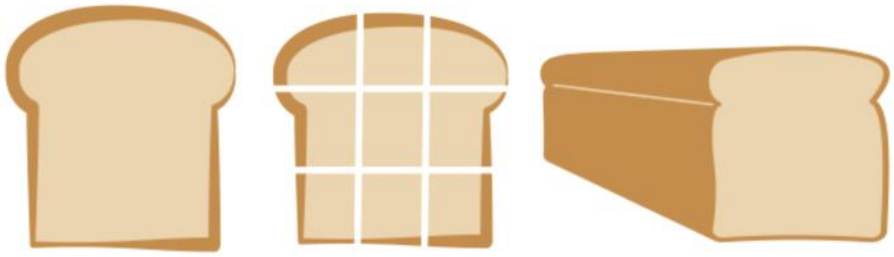
What is TensorFlow?

- **Open-source library for graph-based numerical computation**
  - Developed by the Google Brain Team
- **Low and high level APIs**
  - Addition, multiplication, differentiation
  - Machine learning models
- **Important changes in TensorFlow 2.0**
  - Eager execution by default
  - Model building with Keras and Estimators

What is a tensor?

- Generalization of vectors and matrices
- Collection of numbers
- Specific shape

What is a tensor?



Source: Public Domain Vectors

A collection of 3 pieces that form a row or column is a 1-dimensional tensor.

All 9 pieces together are a 2-dimensional tensor.

And the whole loaf, which contains many slices, is a 3-dimensional tensor.

Defining tensors in TensorFlow

```
import tensorflow as tf

# 0D Tensor
d0 = tf.ones((1,))
```

```
# 1D Tensor
d1 = tf.ones((2,))
```

```
# 2D Tensor
d2 = tf.ones((2, 2))
```

```
# 3D Tensor
d3 = tf.ones((2, 2, 2))
```

We will then define 0-, 1-, 2-, and 3-dimensional tensors.

Note that each object will be a tf dot Tensor object.

Defining tensors in TensorFlow

```
# Print the 3D tensor
print(d3.numpy())
```

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

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

Defining constants in TensorFlow

- A constant is the simplest category of tensor
  - Not trainable
  - Can have any dimension

```
from tensorflow import constant

# Define a 2x3 constant.
a = constant(3, shape=[2, 3])
```

```
# Define a 2x2 constant.
b = constant([1, 2, 3, 4], shape=[2, 2])
```

Using convenience functions to define constants

Operation	Example
`tf.constant()`	`constant([1, 2, 3])`
`tf.zeros()`	`zeros([2, 2])`
`tf.zeros_like()`	`zeros_like(input_tensor)`
`tf.ones()`	`ones([2, 2])`
`tf.ones_like()`	`ones_like(input_tensor)`
`tf.fill()`	`fill([3, 3], 7)`

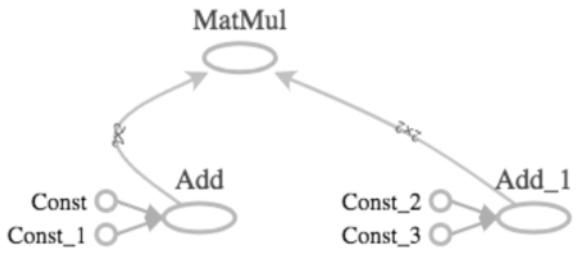
Defining and initializing variables

```
import tensorflow as tf

# Define a variable
a0 = tf.Variable([1, 2, 3, 4, 5, 6], dtype=tf.float32)
a1 = tf.Variable([1, 2, 3, 4, 5, 6], dtype=tf.int16)
```

```
# Define a constant
b = tf.constant(2, tf.float32)
```

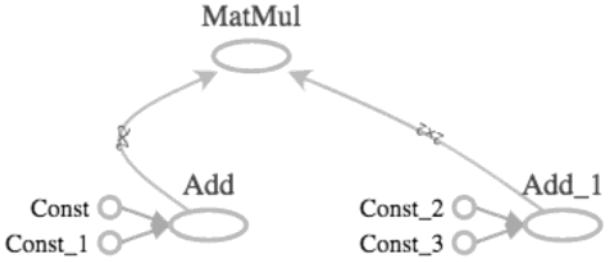
```
# Compute their product
c0 = tf.multiply(a0, b)
c1 = a0*b
```

What is a TensorFlow operation?



A TensorFlow graph contains edges and nodes, where the edges are tensors and the nodes are operations.

Another two tensors are then summed using the add operation.



Finally, the resulting matrices are multiplied together with the matmul operation.

Applying the addition operator

```
#Import constant and add from tensorflow
from tensorflow import constant, add

# Define 0-dimensional tensors
A0 = constant([1])
B0 = constant([2])
```

```
# Define 1-dimensional tensors
A1 = constant([1, 2])
B1 = constant([3, 4])
```

```
# Define 2-dimensional tensors
A2 = constant([[1, 2], [3, 4]])
B2 = constant([[5, 6], [7, 8]])
```

Applying the addition operator

```
# Perform tensor addition with add()
C0 = add(A0, B0)
C1 = add(A1, B1)
C2 = add(A2, B2)
```

Performing tensor addition

- The `add()` operation performs **element-wise addition** with two tensors
- Element-wise addition requires both tensors to have the same shape:
  - Scalar addition: $1 + 2 = 3$
  - Vector addition: $[1, 2] + [3, 4] = [4, 6]$
  - Matrix addition: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$
- The `add()` operator is overloaded

How to perform multiplication in TensorFlow

- **Element-wise multiplication** performed using `multiply()` operation
  - The tensors multiplied must have the same shape
  - E.g. [1,2,3] and [3,4,5] or [1,2] and [3,4]
- **Matrix multiplication** performed with `matmul()` operator
  - The `matmul(A,B)` operation multiplies A by B
  - Number of columns of A must equal the number of rows of B

Applying the multiplication operators

```
# Import operators from tensorflow
from tensorflow import ones, matmul, multiply

# Define tensors
A0 = ones(1)
A31 = ones([3, 1])
A34 = ones([3, 4])
A43 = ones([4, 3])
```

- What types of operations are valid?
  - `multiply(A0, A0)`, `multiply(A31, A31)`, and `multiply(A34, A34)`
  - `matmul(A43, A34)`, but not `matmul(A43, A43)`

Summing over tensor dimensions

- The `reduce_sum()` operator sums over the dimensions of a tensor
  - `reduce_sum(A)` sums over all dimensions of A
  - `reduce_sum(A, i)` sums over dimension i

```
# Import operations from tensorflow
from tensorflow import ones, reduce_sum

# Define a 2x3x4 tensor of ones
A = ones([2, 3, 4])
```

Summing over tensor dimensions

```
# Sum over all dimensions
B = reduce_sum(A)

# Sum over dimensions 0, 1, and 2
B0 = reduce_sum(A, 0)
B1 = reduce_sum(A, 1)
B2 = reduce_sum(A, 2)
```

Overview of advanced operations

- We have covered basic operations in TensorFlow
  - `add()`, `multiply()`, `matmul()`, and `reduce_sum()`
- In this lesson, we explore advanced operations
  - `gradient()`, `reshape()`, and `random()`

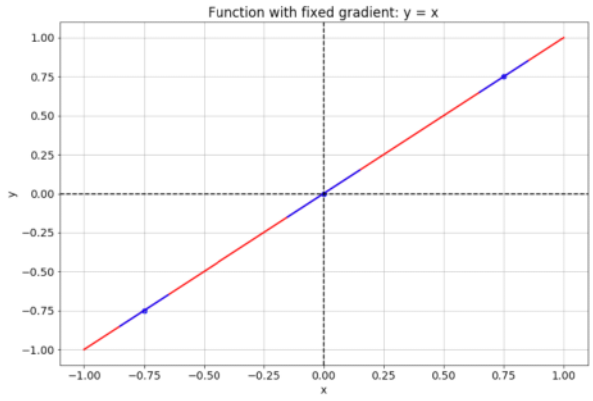
Overview of advanced operations

Operation	Use
`gradient()`	Computes the slope of a function at a point
`reshape()`	Reshapes a tensor (e.g. 10x10 to 100x1)
`random()`	Populates tensor with entries drawn from a probability distribution

Finding the optimum

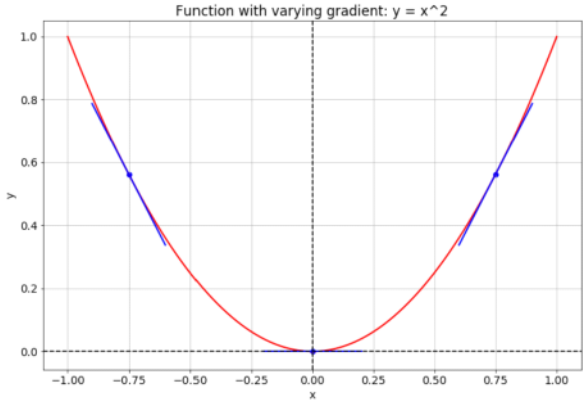
- In many problems, we will want to find the optimum of a function.
  - **Minimum**: Lowest value of a loss function.
  - **Maximum**: Highest value of objective function.
- We can do this using the `gradient()` operation.
  - **Optimum**: Find a point where gradient = 0.
  - **Minimum**: Change in gradient > 0
  - **Maximum**: Change in gradient < 0

Calculating the gradient



If we increase x by 1 unit, y also increases by 1 unit.

Calculating the gradient



Gradients in TensorFlow

```
# Import tensorflow under the alias tf
import tensorflow as tf

# Define x
x = tf.Variable(-1.0)
```

```
# Define y within instance of GradientTape
with tf.GradientTape() as tape:
    tape.watch(x)
    y = tf.multiply(x, x)
```

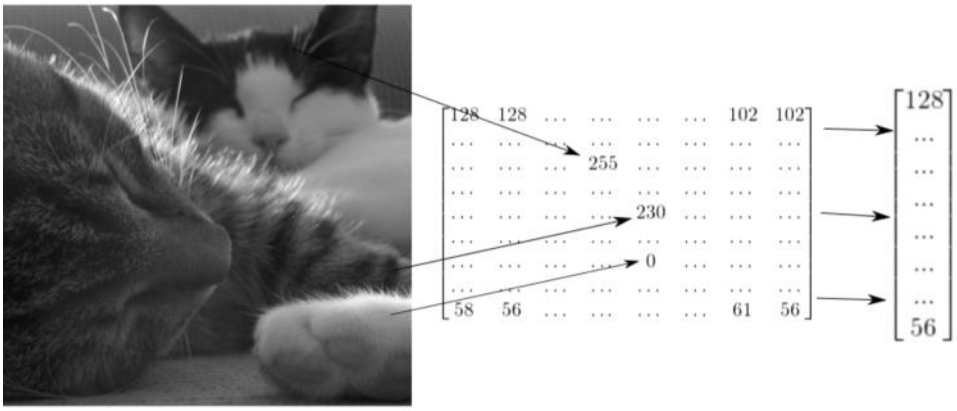
```
# Evaluate the gradient of y at x = -1
g = tape.gradient(y, x)
print(g.numpy())
```

```
-2.0
```

APIs; however, gradient tape remains an invaluable tool for building advanced and custom models.

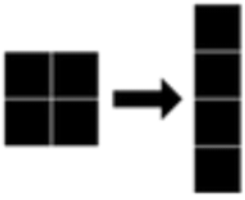
-2.0

Images as tensors



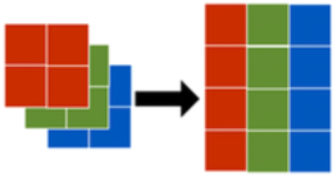
How to reshape a grayscale image

```
# Import tensorflow as alias tf
import tensorflow as tf

# Generate grayscale image
gray = tf.random.uniform([2, 2], maxval=255, dtype='int32')

# Reshape grayscale image
gray = tf.reshape(gray, [2*2, 1])
```



How to reshape a color image

```
# Import tensorflow as alias tf
import tensorflow as tf

# Generate color image
color = tf.random.uniform([2, 2, 3], maxval=255, dtype='int32')

# Reshape color image
color = tf.reshape(color, [2*2, 3])
```



Linear Models

NUMERIC DATA

price	bedrooms	bathrooms	sqft_living
221900.0	3	1.00	1180
538000.0	3	2.25	2570
180000.0	2	1.00	770
604000.0	4	3.00	1960
510000.0	3	2.00	1680
1225000.0	4	4.50	5420
257500.0	3	2.25	1715
291850.0	3	1.50	1060
229500.0	3	1.00	1780
323000.0	3	2.50	1890
662500.0	3	2.50	3560
468000.0	2	1.00	1160

IMAGE DATA

```
[[164, 161, 159, ...,  79,  87, 131],
 [161, 162, 164, ...,  98, 117, 146],
 [147, 151, 151, ..., 155, 165, 167],  93, 137],
 ...,                                  123, 152],
 [178, 172, 176, ..., 178, 175, 123], 172, 174],  91, 135],
 [ 84,  82,  86, ..., 168, 192, 175],           119, 148],
 [157, 158, 162, ..., 157, 179, 174]] 180, 128],  165, 167],
        [ 88,  87,  88, ..., 169, 194, 177],
        [159, 158, 161, ..., 156, 180, 175]] 176, 122],
               [ 87,  83,  85, ..., 164, 189, 172],
               [158, 156, 159, ..., 151, 174, 169]]
```

TEXT DATA

King County is one of three Washington counties that are included in the Seattle–Tacoma–Bellevue metropolitan statistical area. (The others are Snohomish County to the north, and Pierce County to the south.) About two-thirds of King County's population lives in Seattle's suburbs.

```
0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 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 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Importing data for use in TensorFlow

- **Data can be imported using** `tensorflow`
  - Useful for managing complex pipelines
  - Not necessary for this chapter
- **Simpler option used in this chapter**
  - Import data using `pandas`
  - Convert data to `numpy` array
  - Use in `tensorflow` without modification

How to import and convert data

```
# Import numpy and pandas
import numpy as np
import pandas as pd

# Load data from csv
housing = pd.read_csv('kc_housing.csv')

# Convert to numpy array
housing = np.array(housing)
```

- We will focus on data stored in csv format in this chapter
- Pandas also has methods for handling data in other formats
  - E.g. `read_json()` , `read_html()` , `read_excel()`

Parameters of read_csv()

Parameter	Description	Default
`filepath_or_buffer`	Accepts a file path or a URL.	`None`
`sep`	Delimiter between columns.	`,`
`delim_whitespace`	Boolean for whether to delimit whitespace.	`False`
`encoding`	Specifies encoding to be used if any.	`None`

Using mixed type datasets

date	price	bedrooms
20141013T000000	221900	3
20141209T000000	538000	3
20150225T000000	180000	2
20141209T000000	604000	4
20150218T000000	510000	3
20140627T000000	257500	3
20150115T000000	291850	3
20150415T000000	229500	3

floors	waterfront	view
1	0	0
2	0	0
1	1	0
1	0	0
1	0	2
2	0	0
1	0	4
1	0	0

Setting the data type

```
# Load KC dataset
housing = pd.read_csv('kc_housing.csv')

# Convert price column to float32
price = np.array(housing['price'], np.float32)

# Convert waterfront column to Boolean
waterfront = np.array(housing['waterfront'], np.bool)
```

Setting the data type

```
# Load KC dataset
housing = pd.read_csv('kc_housing.csv')

# Convert price column to float32
price = tf.cast(housing['price'], tf.float32)

# Convert waterfront column to Boolean
waterfront = tf.cast(housing['waterfront'], tf.bool)
```

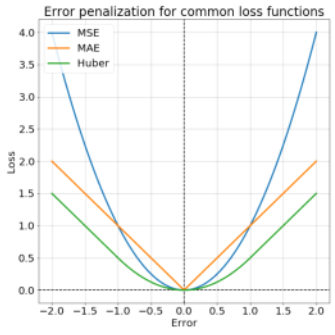
Introduction to loss functions

- **Fundamental `tensorflow` operation**
  - Used to train a model
  - Measure of model fit
- **Higher value -> worse fit**
  - Minimize the loss function

Common loss functions in TensorFlow

- **TensorFlow has operations for common loss functions**
  - Mean squared error (MSE)
  - Mean absolute error (MAE)
  - Huber error
- **Loss functions are accessible from `tf.keras.losses()`**
  - `tf.keras.losses.mse()`
  - `tf.keras.losses.mae()`
  - `tf.keras.losses.Huber()`

Why do we care about loss functions?



- MSE
  - Strongly penalizes outliers
  - High (gradient) sensitivity near minimum
- MAE
  - Scales linearly with size of error
  - Low sensitivity near minimum
- Huber
  - Similar to MSE near minimum
  - Similar to MAE away from minimum

Defining a loss function

```
# Import TensorFlow under standard alias
import tensorflow as tf

# Compute the MSE loss
loss = tf.keras.losses.mse(targets, predictions)
```

Defining a loss function

```
# Define a linear regression model
def linear_regression(intercept, slope = slope, features = features):
    return intercept + features*slope
```

```
# Define a loss function to compute the MSE
def loss_function(intercept, slope, targets = targets, features = features):
    # Compute the predictions for a linear model
    predictions = linear_regression(intercept, slope)

    # Return the loss
    return tf.keras.losses.mse(targets, predictions)
```

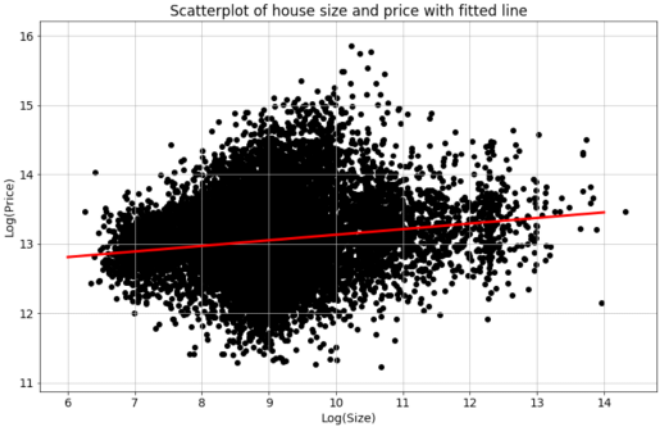
Defining the loss function

```
# Compute the loss for test data inputs
loss_function(intercept, slope, test_targets, test_features)
```

```
10.77
```

```
# Compute the loss for default data inputs
loss_function(intercept, slope)
```

What is a linear regression?



The linear regression model

- A linear regression model assumes a linear relationship:
  - $price = intercept + size * slope + error$
- This is an example of a univariate regression.
  - There is only one feature, `size` .
- Multiple regression models have more than one feature.
  - E.g. `size` and `location`

```
# Define the targets and features
price = np.array(housing['price'], np.float32)
size = np.array(housing['sqft_living'], np.float32)

# Define the intercept and slope
intercept = tf.Variable(0.1, np.float32)
slope = tf.Variable(0.1, np.float32)
```

```
# Define a linear regression model
def linear_regression(intercept, slope, features = size):
    return intercept + features*slope
```

```
# Compute the predicted values and loss
def loss_function(intercept, slope, targets = price, features = size):
    predictions = linear_regression(intercept, slope)
    return tf.keras.losses.mse(targets, predictions)
```

Linear regression in TensorFlow

```
# Define an optimization operation
opt = tf.keras.optimizers.Adam()
```

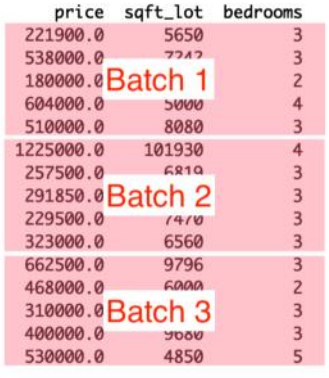
```
# Minimize the loss function and print the loss
for j in range(1000):
    opt.minimize(lambda: loss_function(intercept, slope),\
    var_list=[intercept, slope])
    print(loss_function(intercept, slope))
```

```
tf.Tensor(10.909373, shape=(), dtype=float32)
...
tf.Tensor(0.15479447, shape=(), dtype=float32)
```

```
# Print the trained parameters
print(intercept.numpy(), slope.numpy())
```

What is batch training?

price	sqft_lot	bedrooms
221900.0	5650	3
538000.0	7242	3
180000.0	10000	2
604000.0	5000	4
510000.0	8080	3
1225000.0	101930	4
257500.0	6819	3
291850.0	9711	3
229500.0	7470	3
323000.0	6560	3
662500.0	9796	3
468000.0	6000	2
310000.0	19901	3
400000.0	9680	3
530000.0	4850	5



Beyond alleviating memory constraints, batch training will also allow you to update

model weights and optimizer parameters after each batch, rather than at the end of the epoch.

The chunksize parameter

- `pd.read_csv()` allows us to load data in batches
  - Avoid loading entire dataset
  - `chunksize` parameter provides batch size

```
# Import pandas and numpy
import pandas as pd
import numpy as np

# Load data in batches
for batch in pd.read_csv('kc_housing.csv', chunksize=100):
    # Extract price column
    price = np.array(batch['price'], np.float32)

    # Extract size column
    size = np.array(batch['size'], np.float32)
```

Training a linear model in batches

```
# Import tensorflow, pandas, and numpy
import tensorflow as tf
import pandas as pd
import numpy as np
```

```
# Define trainable variables
intercept = tf.Variable(0.1, tf.float32)
slope = tf.Variable(0.1, tf.float32)
```

```
# Define the model
def linear_regression(intercept, slope, features):
    return intercept + features*slope
```

Training a linear model in batches

```
# Compute predicted values and return loss function
def loss_function(intercept, slope, targets, features):
    predictions = linear_regression(intercept, slope, features)
    return tf.keras.losses.mse(targets, predictions)
```

```
# Define optimization operation
opt = tf.keras.optimizers.Adam()
```

Training a linear model in batches

```
# Load the data in batches from pandas
for batch in pd.read_csv('kc_housing.csv', chunksize=100):
    # Extract the target and feature columns
    price_batch = np.array(batch['price'], np.float32)
    size_batch = np.array(batch['lot_size'], np.float32)

    # Minimize the loss function
    opt.minimize(lambda: loss_function(intercept, slope, price_batch, size_batch),
                 var_list=[intercept, slope])
```

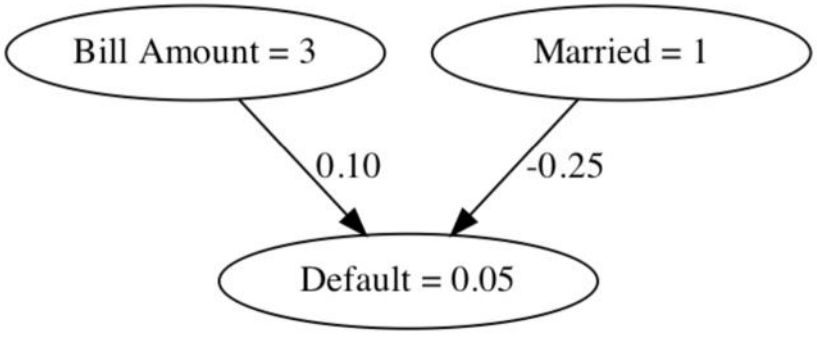
```
# Print parameter values
print(intercept.numpy(), slope.numpy())
```

Full sample versus batch training

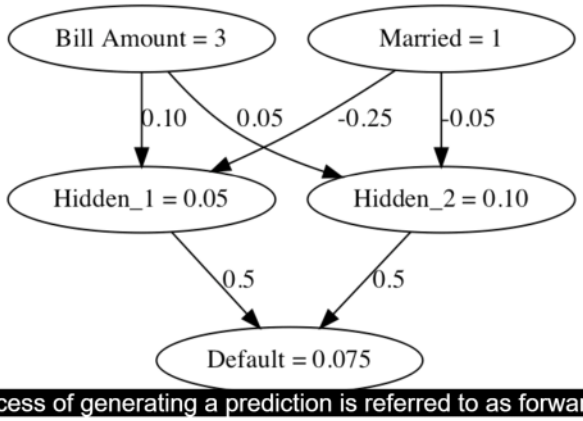
- **Full Sample**
  1. One update per epoch
  2. Accepts dataset without modification
  3. Limited by memory
- **Batch Training**
  1. Multiple updates per epoch
  2. Requires division of dataset
  3. No limit on dataset size

Neural Networks

The linear regression model

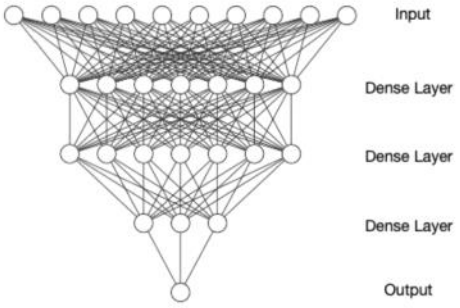


What is a neural network?



This entire process of generating a prediction is referred to as forward propagation.

What is a neural network?



- A dense layer applies weights to all nodes from the previous layer.

A simple dense layer

```
import tensorflow as tf
```

```
# Define inputs (features)
inputs = tf.constant([[1, 35]])
```

```
# Define weights
weights = tf.Variable([[-0.05], [-0.01]])
```

```
# Define the bias
bias = tf.Variable([0.5])
```

We'll first define a constant tensor that contains the marital status and age data as the input layer.

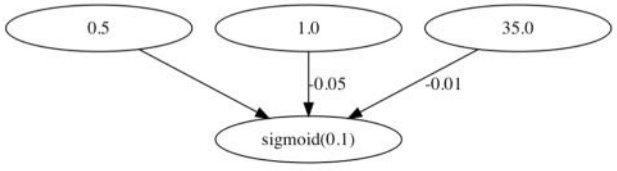
We then initialize weights as a variable, since we will train those weights to predict the output from the inputs.

We also define a bias, which will play a similar role to the intercept in the linear regression model.

A simple dense layer

```
# Multiply inputs (features) by the weights
product = tf.matmul(inputs, weights)
```

```
# Define dense layer
dense = tf.keras.activations.sigmoid(product+bias)
```



Defining a complete model

```
import tensorflow as tf
```

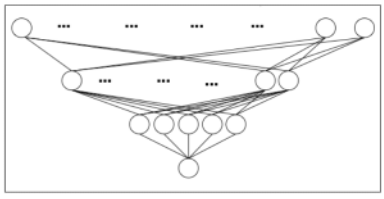
```
# Define input (features) layer
inputs = tf.constant(data, tf.float32)
```

```
# Define first dense layer
dense1 = tf.keras.layers.Dense(10, activation='sigmoid')(inputs)
```

Defining a complete model

```
# Define second dense layer
dense2 = tf.keras.layers.Dense(5, activation='sigmoid')(dense1)
```

```
# Define output (predictions) layer
outputs =  tf.keras.layers.Dense(1, activation='sigmoid')(dense2)
```



High-level versus low-level approach

- **High-level approach**
  - High-level API operations

```
dense = keras.layers.Dense(10,\
 activation='sigmoid')
```

- **Low-level approach**
  - Linear-algebraic operations

```
prod = matmul(inputs, weights)
dense = keras.activations.sigmoid(prod)
```

What is an activation function?

- **Components of a typical hidden layer**
  - **Linear**: Matrix multiplication
  - **Nonlinear**: Activation function

A simple example

```
import numpy as np
import tensorflow as tf

# Define example borrower features
young, old = 0.3, 0.6
low_bill, high_bill = 0.1, 0.5
```

```
# Apply matrix multiplication step for all feature combinations
young_high = 1.0*young + 2.0*high_bill
young_low = 1.0*young + 2.0*low_bill
old_high = 1.0*old + 2.0*high_bill
old_low = 1.0*old + 2.0*low_bill
```

Let's look at a simple example, where we assume that the weight on age is 1 and the weight on the bill amount is 2.

A simple example

```
# Difference in default predictions for young
print(young_high - young_low)

# Difference in default predictions for old
print(old_high - old_low)
```

```
0.8
0.8
```

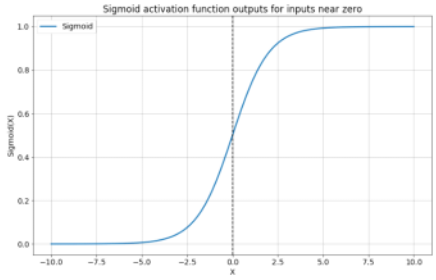
A simple example

```
# Difference in default predictions for young
print(tf.keras.activations.sigmoid(young_high).numpy() -
tf.keras.activations.sigmoid(young_low).numpy())

# Difference in default predictions for old
print(tf.keras.activations.sigmoid(old_high).numpy() -
tf.keras.activations.sigmoid(old_low).numpy())
```

```
0.16337568
0.14204389
```

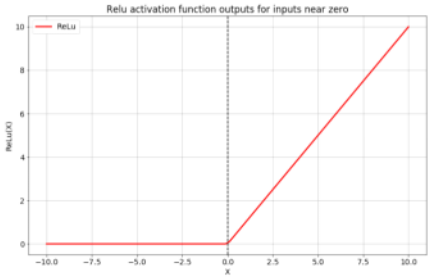
The sigmoid activation function

- **Sigmoid activation function**
  - Binary classification
  - Low-level: `tf.keras.activations.sigmoid()`
  - High-level: `sigmoid`



The relu activation function

- **ReLu activation function**
  - Hidden layers
  - Low-level: `tf.keras.activations.relu()`
  - High-level: `relu`



The softmax activation function

- **Softmax activation function**
  - Output layer (>2 classes)
  - Low-level: `tf.keras.activations.softmax()`
  - High-level: `softmax`

Activation functions in neural networks

```
import tensorflow as tf
```

```
# Define input layer
inputs = tf.constant(borrower_features, tf.float32)
```

```
# Define dense layer 1
dense1 = tf.keras.layers.Dense(16, activation='relu')(inputs)
```

```
# Define dense layer 2
dense2 = tf.keras.layers.Dense(8, activation='sigmoid')(dense1)
```

```
# Define output layer
outputs = tf.keras.layers.Dense(4, activation='softmax')(dense2)
```

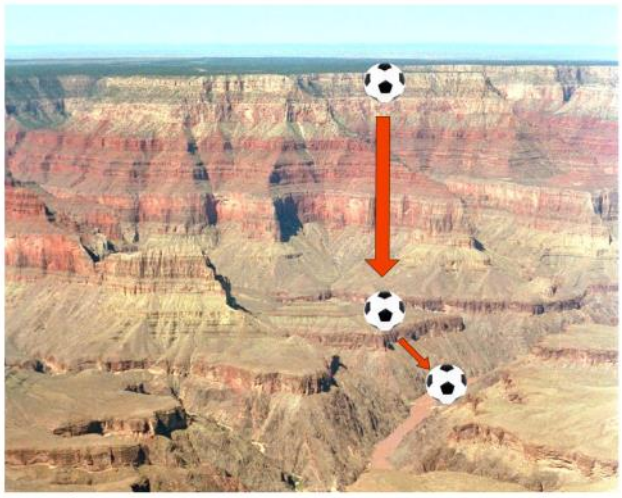
How to find a minimum

How to find a minimum

Let's start by picking a point and measuring the elevation.

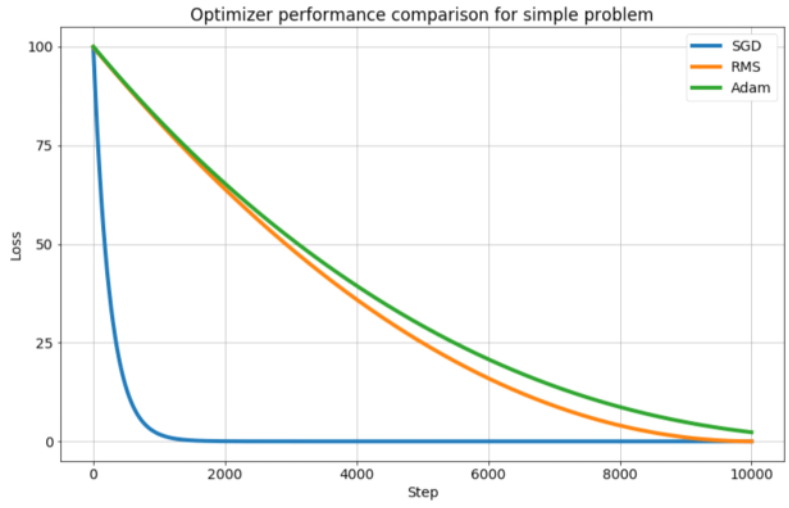
[1] Source: U.S. National Park Service

How to find a minimum



Source: U.S. National Park Service

Let's say you pick a different spot.



Stochastic gradient descent or SGD is an improved version of

The gradient descent optimizer

- Stochastic gradient descent (SGD) optimizer
  - `tf.keras.optimizers.SGD()`
  - `learning_rate`
- Simple and easy to interpret

The RMS prop optimizer

- Root mean squared (RMS) propagation optimizer
  - Applies different learning rates to each feature
  - `tf.keras.optimizers.RMSprop()`
  - `learning_rate`
  - `momentum`
  - `decay`
- Allows for momentum to both build and decay

The adam optimizer

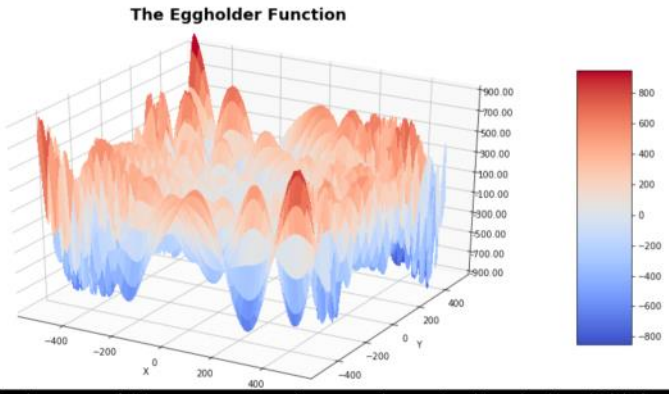
- Adaptive moment (adam) optimizer
  - `tf.keras.optimizers.Adam()`
  - `learning_rate`
  - `beta1`
- Performs well with default parameter values

A complete example

```
import tensorflow as tf

# Define the model function
def model(bias, weights, features = borrower_features):
    product = tf.matmul(features, weights)
    return tf.keras.activations.sigmoid(product+bias)
```

```
# Compute the predicted values and loss
def loss_function(bias, weights, targets = default, features = borrower_features):
    predictions = model(bias, weights)
    return tf.keras.losses.binary_crossentropy(targets, predictions)
```

```
# Minimize the loss function with RMS propagation
opt = tf.keras.optimizers.RMSprop(learning_rate=0.01, momentum=0.9)
opt.minimize(lambda: loss_function(bias, weights), var_list=[bias, weights])
```



We also saw that we could improve our chances by selecting better initial values for variables.

Random initializers

- **Often need to initialize thousands of variables**
  - `tf.ones()` may perform poorly
  - Tedious and difficult to initialize variables individually
- **Alternatively, draw initial values from distribution**
  - Normal
  - Uniform
  - Glorot initializer

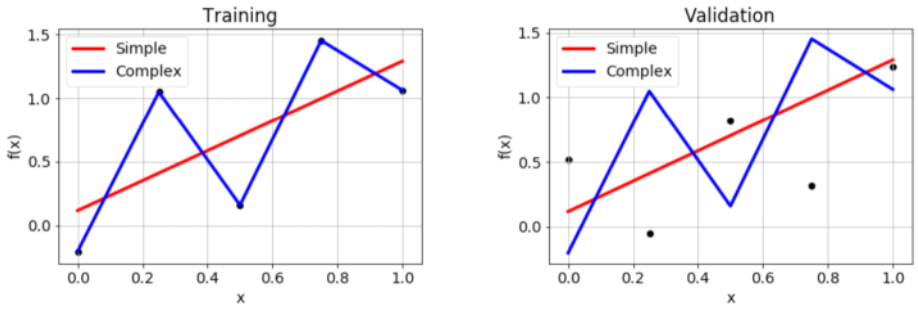
Initializing variables in TensorFlow

```
import tensorflow as tf

# Define 500x500 random normal variable
weights = tf.Variable(tf.random.normal([500, 500]))

# Define 500x500 truncated random normal variable
weights = tf.Variable(tf.random.truncated_normal([500, 500]))
```

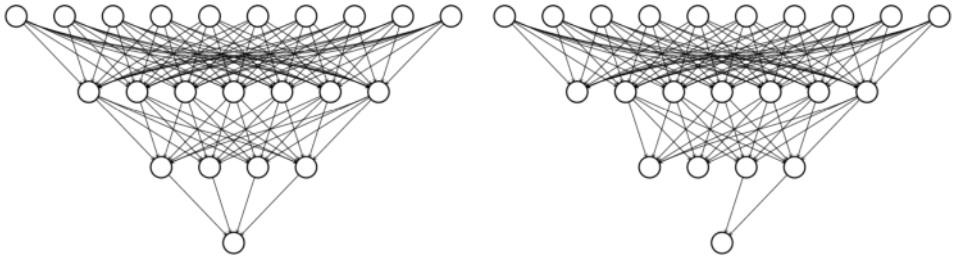
Initializing variables in TensorFlow

```
# Define a dense layer with the default initializer
dense = tf.keras.layers.Dense(32, activation='relu')

# Define a dense layer with the zeros initializer
dense = tf.keras.layers.Dense(32, activation='relu',\
    kernel_initializer='zeros')
```

Neural networks and overfitting



Applying dropout



Implementing dropout in a network

```
import numpy as np
import tensorflow as tf

# Define input data
inputs = np.array(borrower_features, np.float32)
```

```
# Define dense layer 1
dense1 = tf.keras.layers.Dense(32, activation='relu')(inputs)
```

Implementing dropout in a network

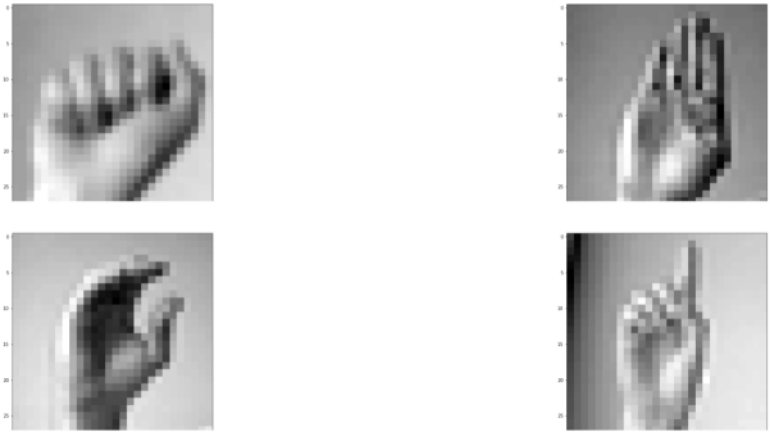
```
# Define dense layer 2
dense2 = tf.keras.layers.Dense(16, activation='relu')(dense1)
```

```
# Apply dropout operation
dropout1 = tf.keras.layers.Dropout(0.25)(dense2)
```

```
# Define output layer
outputs = tf.keras.layers.Dense(1, activation='sigmoid')(dropout1)
```

High Level API

Classifying sign language letters



The sequential API

- Input layer
- Hidden layers
- Output layer
- Ordered in sequence

Building a sequential model

```
# Import tensorflow
from tensorflow import keras

# Define a sequential model
model = keras.Sequential()
```

```
# Define first hidden layer
model.add(keras.layers.Dense(16, activation='relu', input_shape=(28*28,)))
```

Building a sequential model

```
# Define second hidden layer
model.add(keras.layers.Dense(8, activation='relu'))
```

```
# Define output layer
model.add(keras.layers.Dense(4, activation='softmax'))
```

```
# Compile the model
model.compile('adam', loss='categorical_crossentropy')
```

```
# Summarize the model
print(model.summary())
```

Using the functional API

```
# Import tensorflow
import tensorflow as tf

# Define model 1 input layer shape
model1_inputs = tf.keras.Input(shape=(28*28,))

# Define model 2 input layer shape
model2_inputs = tf.keras.Input(shape=(10,))
```

```
# Define layer 1 for model 1
model1_layer1 = tf.keras.layers.Dense(12, activation='relu')(model1_inputs)

# Define layer 2 for model 1
model1_layer2 = tf.keras.layers.Dense(4, activation='softmax')(model1_layer1)
```

Using the functional API

```
# Define layer 1 for model 2
model2_layer1 = tf.keras.layers.Dense(8, activation='relu')(model2_inputs)

# Define layer 2 for model 2
model2_layer2 = tf.keras.layers.Dense(4, activation='softmax')(model2_layer1)
```

```
# Merge model 1 and model 2
merged = tf.keras.layers.add([model1_layer2, model2_layer2])
```

```
# Define a functional model
model = tf.keras.Model(inputs=[model1_inputs, model2_inputs], outputs=merged)

# Compile the model
model.compile('adam', loss='categorical_crossentropy')
```

Overview of training and evaluation

1. Load and clean data
2. Define model
3. Train and validate model
4. Evaluate model

How to train a model

```
# Import tensorflow
import tensorflow as tf

# Define a sequential model
model = tf.keras.Sequential()
```

```
# Define the hidden layer
model.add(tf.keras.layers.Dense(16, activation='relu', input_shape=(784,)))
```

```
# Define the output layer
model.add(tf.keras.layers.Dense(4, activation='softmax'))
```

How to train a model

```
# Compile model
model.compile('adam', loss='categorical_crossentropy')
```

```
# Train model
model.fit(image_features, image_labels)
```

The fit() operation

- Required arguments
  - `features`
  - `labels`
- Many optional arguments
  - `batch_size`
  - `epochs`
  - `validation_split`

Batch size and epochs

Performing validation

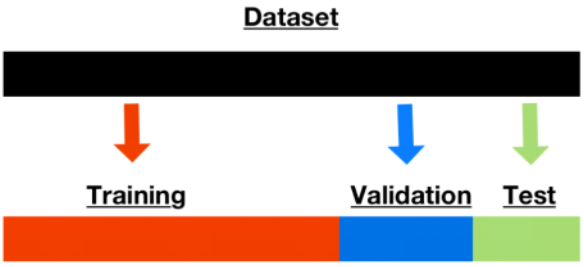
```
# Train model with validation split
model.fit(features, labels, epochs=10, validation_split=0.20)
```

Changing the metric

```
# Recomile the model with the accuracy metric
model.compile('adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
# Train model with validation split
model.fit(features, labels, epochs=10, validation_split=0.20)
```

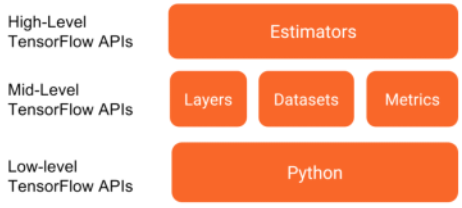
The evaluation() operation




```
# Evaluate the test set
model.evaluate(test)
```

What is the Estimators API?

- High level submodule
- Less flexible
- Enforces best practices
- Faster deployment
- Many premade models



Model specification and training

1. Define feature columns
2. Load and transform data
3. Define an estimator
4. Apply train operation

Defining feature columns

```
# Import tensorflow under its standard alias
import tensorflow as tf

# Define a numeric feature column
size = tf.feature_column.numeric_column("size")
```

```
# Define a categorical feature column
rooms = tf.feature_column.categorical_column_with_vocabulary_list("rooms",\
["1", "2", "3", "4", "5"])
```

Defining feature columns

```
# Create feature column list
features_list = [size, rooms]
```

```
# Define a matrix feature column
features_list = [tf.feature_column.numeric_column('image', shape=(784,))]
```

Loading and transforming data

```
# Define input data function
def input_fn():
    # Define feature dictionary
    features = {"size": [1340, 1690, 2720], "rooms": [1, 3, 4]}
    # Define labels
    labels = [221900, 538000, 180000]
    return features, labels
```

Define and train a regression estimator

```
# Define a deep neural network regression
model0 = tf.estimator.DNNRegressor(feature_columns=feature_list,\
    hidden_units=[10, 6, 6, 3])

# Train the regression model
model0.train(input_fn, steps=20)
```

Define and train a deep neural network

```
# Define a deep neural network classifier
model1 = tf.estimator.DNNClassifier(feature_columns=feature_list,
   hidden_units=[32, 16, 8], n_classes=4)

# Train the classifier
model1.train(input_fn, steps=20)
```

- https://www.tensorflow.org/guide/estimators