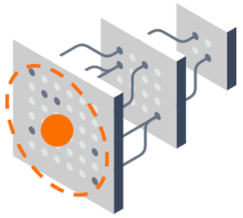
ZOOMING IN

TensorFlow

Carnegie Mellon University
School of Computer Science



Easy model building

Build and train ML models easily using intuitive high-level APIs like Keras with eager execution, which makes for immediate model iteration and easy debugging.

Robust ML production anywhere

Easily train and deploy models in the cloud, on-prem, in the browser, or on-device no matter what language you use.

Powerful experimentation for research

A simple and flexible architecture to take new ideas from concept to code, to state-of-the-art models, and to publication faster.

We'll be using TensorFlow 2.0 in this class

Carnegie Mellon University
School of Computer Science

TENSORFLOW

Tensors

Let’s have a look at **tensors** in TensorFlow!

0-dimensional tensors:

```
sport = tf.constant("Tennis", tf.string)
number = tf.constant(1.41421356237, tf.float64)

print("`sport` is a {}-d Tensor".format(tf.rank(sport).numpy())) print("`number` is a {}-d
Tensor".format(tf.rank(number).numpy()))
```

OUTPUT:

`sport` is a 0-d Tensor
`number` is a 0-d Tensor

TENSORFLOW

Tensors

Let's have a look at **tensors** in TensorFlow!

1-dimensional tensors (vectors):

```
sports = tf.constant(["Tennis", "Basketball"], tf.string)
numbers = tf.constant([3.141592, 1.414213, 2.71821], tf.float64)

print("`sports` is a {}-d Tensor with shape: {}".format(tf.rank(sports).numpy(), tf.shape(sports)))
print("`numbers` is a {}-d Tensor with shape: {}".format(tf.rank(numbers).numpy(), tf.shape(numbers)))
```

OUTPUT:

`sports` is a 1-d Tensor with shape: [2]
`numbers` is a 1-d Tensor with shape: [3]

TENSORFLOW

Tensors

Let’s have a look at **tensors** in TensorFlow!

2-dimensional tensors (matrices):

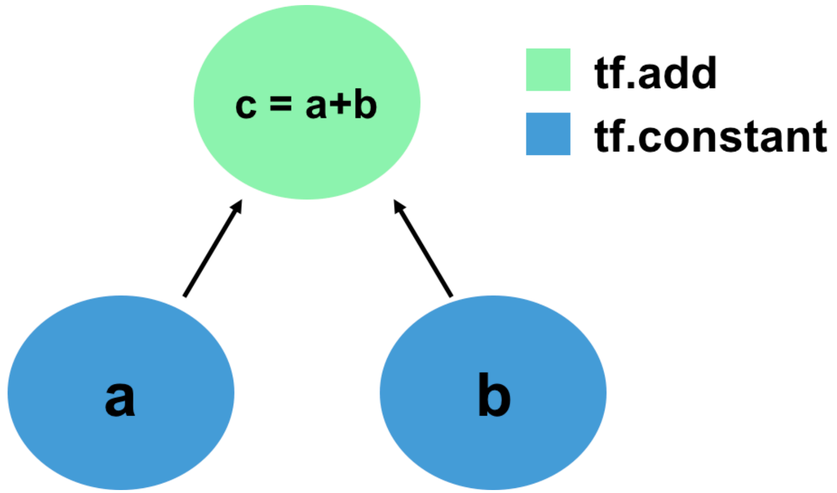
```
matrix = tf.constant([[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]])

print("`matrix` is a {}-d Tensor with shape: {}".format(tf.rank(matrix).numpy(), tf.shape(matrix)))
```

OUTPUT:
`matrix` is a 2-d Tensor with shape: [2 4]

Computations on Tensors

Computations can be thought of in TensorFlow as **graphs**.

We can define this graph in terms of Tensors, which hold data, and the mathematical operations that act on these Tensors in some order. Let's look at a simple example, and define this computation using TensorFlow:

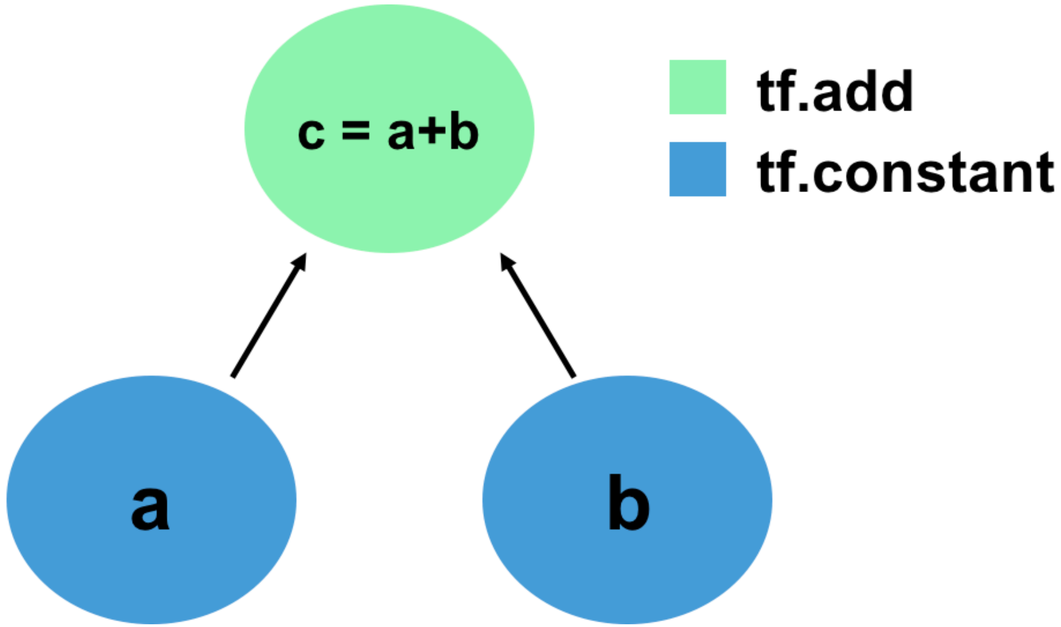


TENSORFLOW

Computations on Tensors

```
# Create the nodes in the graph, and
# initialize values
a = tf.constant(15)
b = tf.constant(61)

# Add them!
c1 = tf.add(a,b)
c2 = a + b

# TensorFlow overrides the "+" operation
# so that it is able to act on Tensors
print(c1)
print(c2)
```

```
tf.Tensor(76, shape=(), dtype=int32) tf.Tensor(76, shape=(),
dtype=int32)
```

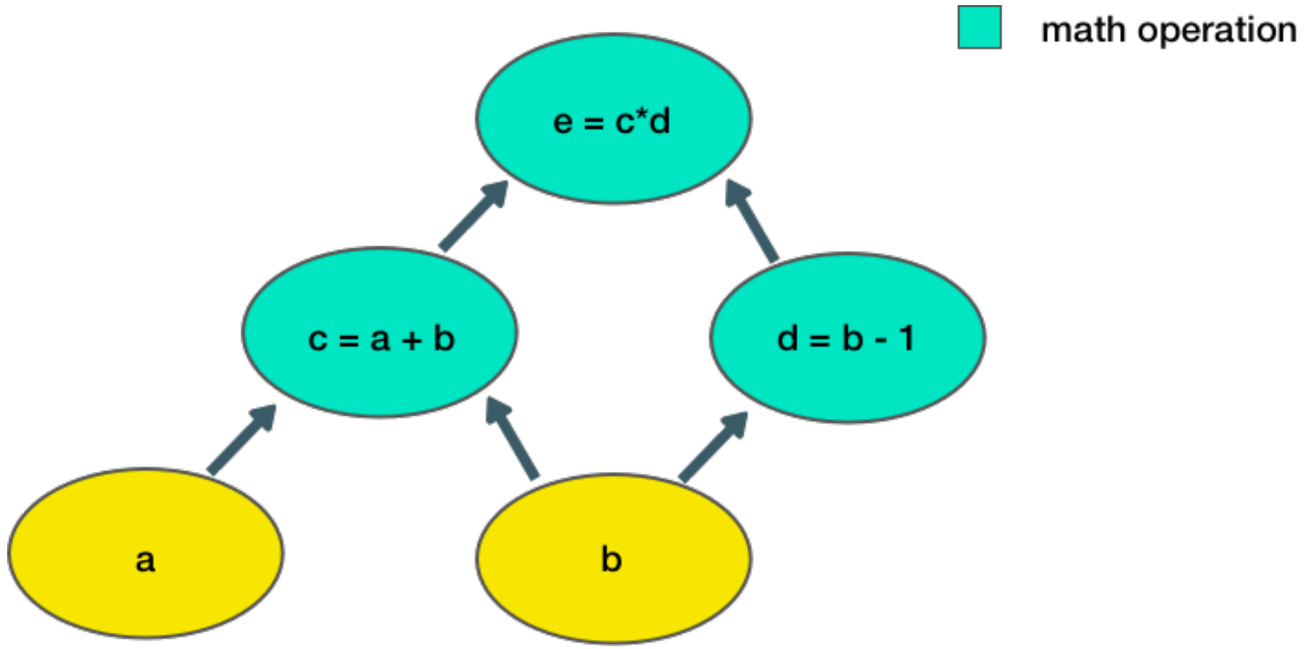


TENSORFLOW

Computations on Tensors

Lets look at another example. Here, we take two inputs, a, b, and compute an output e.

Each node in the graph represents an operation that takes some input, does some computation, and passes its output to another node.



TENSORFLOW

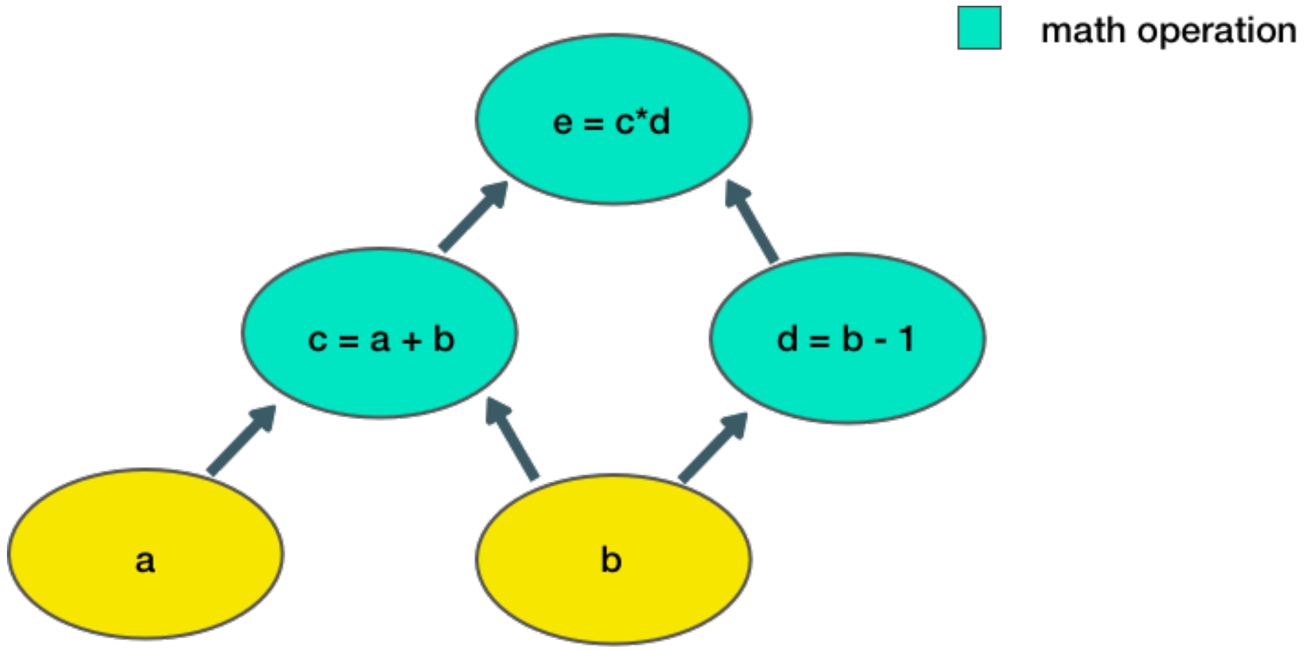
Computations on Tensors

```
# Construct a simple computation function
def func(a,b):
  '''Define the operation for c, d, e'''
  '''(use tf.add, tf.subtract, tf.multiply).'''
  c = tf.add(a, b)
  d = tf.subtract(b, 1)
  e = tf.multiply(c, d)
  return e
```

Now, we can call this function to execute the computation graph given some inputs a,b…

```
# Consider example values for a,b
a, b = 1.5, 2.5
# Execute the computation e_out = func(a,b) print(e_out)
```

tf.Tensor(6.0, shape=(), dtype=float32)

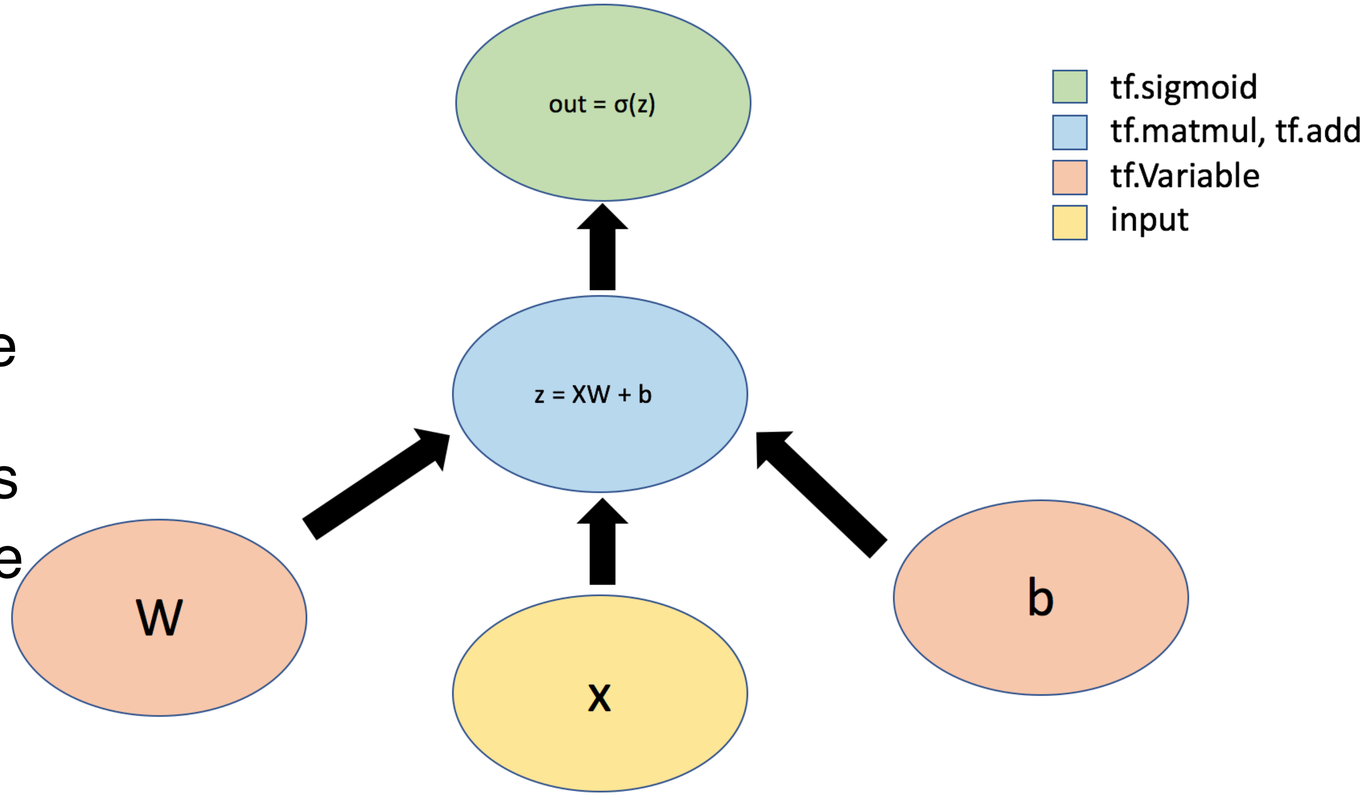


TENSORFLOW

Neural Networks in TensorFlow

How does this transfer to neural networks?

Let's consider the example of a simple perceptron defined by just one dense layer: $y=\sigma(Wx+b)$, where W represents a matrix of weights, b is a bias, x is the input, σ is the sigmoid activation function, and y is the output.



We can also visualize this operation using a graph ->

TENSORFLOW

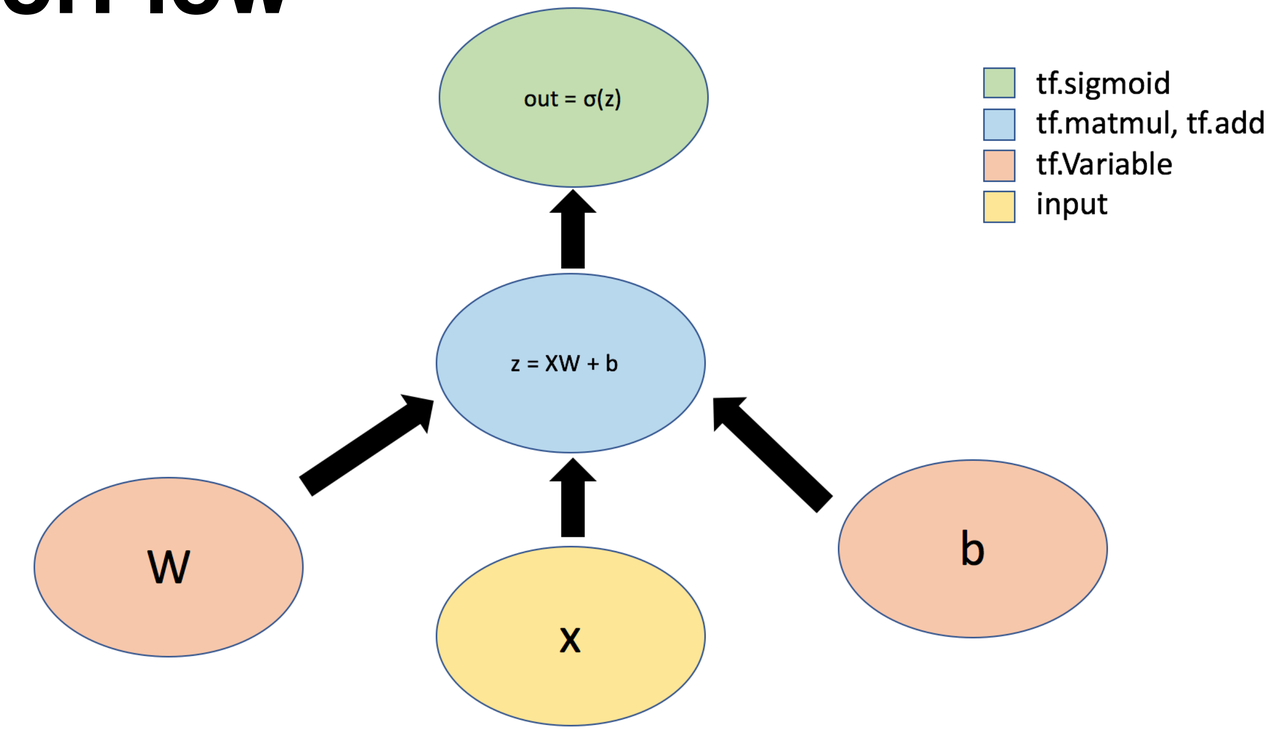
Neural Networks in TensorFlow

Tensors can flow through abstract types called **Layers** -- the building blocks of neural networks.

Layers implement common neural networks operations, and are used to:

- update weights,
- compute losses,
- and define inter-layer connectivity.

We will first define a Layer to implement our simple perceptron.

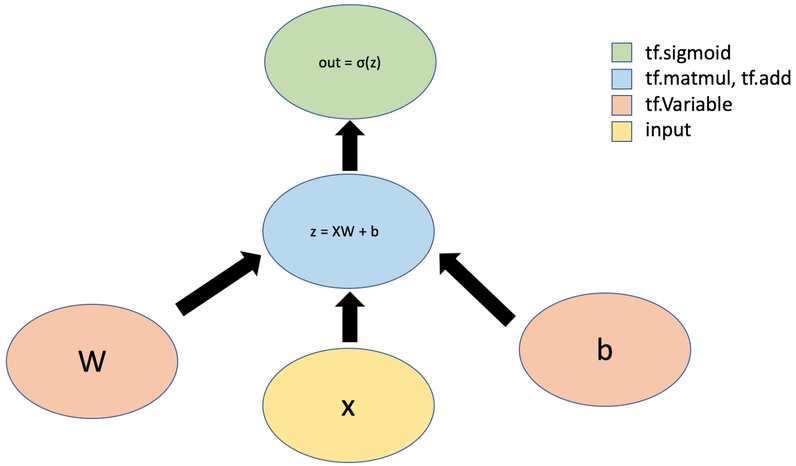


TENSORFLOW

Neural Networks in TensorFlow

The custom layer approach

```
# n_output_nodes: number of output nodes
# input_shape: shape of the input
# x: input to the layer

class OurDenseLayer(tf.keras.layers.Layer):
  def __init__(self, n_output_nodes):
    super(OurDenseLayer, self).__init__()
    self.n_output_nodes = n_output_nodes

  def build(self, input_shape):
    d = int(input_shape[-1])
    # Define and initialize parameters: a weight matrix W and bias b
    # Note that parameter initialization is random!
    self.W = self.add_weight("weight", shape=[d, self.n_output_nodes])
    self.b = self.add_weight("bias", shape=[1, self.n_output_nodes])

  def call(self, x):
    '''define the operation for z (hint: use tf.matmul)'''
    z = tf.matmul(x, self.W) + self.b

    '''define the operation for out (hint: use tf.sigmoid)'''
    y = tf.sigmoid(z)
    return y
```



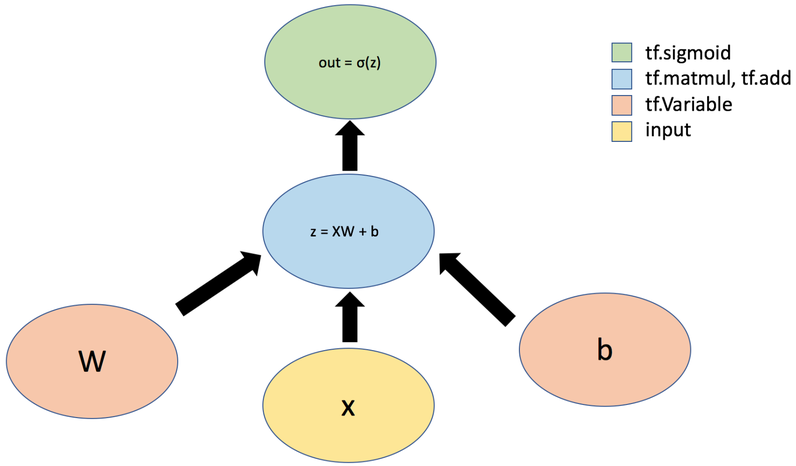

```
# Since layer parameters are initialized randomly, we will set a random
# seed for reproducibility
tf.random.set_seed(1)
layer = OurDenseLayer(3)
layer.build((1,2))
x_input = tf.constant([[1,2.]], shape=(1,2))
y = layer.call(x_input)

# test the output!
print(y.numpy())
mdl.lab1.test_custom_dense_layer_output(y)
```

```
[[0.26978594 0.45750412 0.66536945]]
[PASS] test_custom_dense_layer_output
True
```

TENSORFLOW

Neural Networks in TensorFlow

Using Keras's Sequential API



Conveniently, TensorFlow has defined a number of Layers that are commonly used in neural networks, for example a *Dense*.

Now, instead of using a single Layer to define our simple neural network, we'll use the Sequential model from Keras (standard in TensorFlow 2.0) and a single Dense layer to define our network.

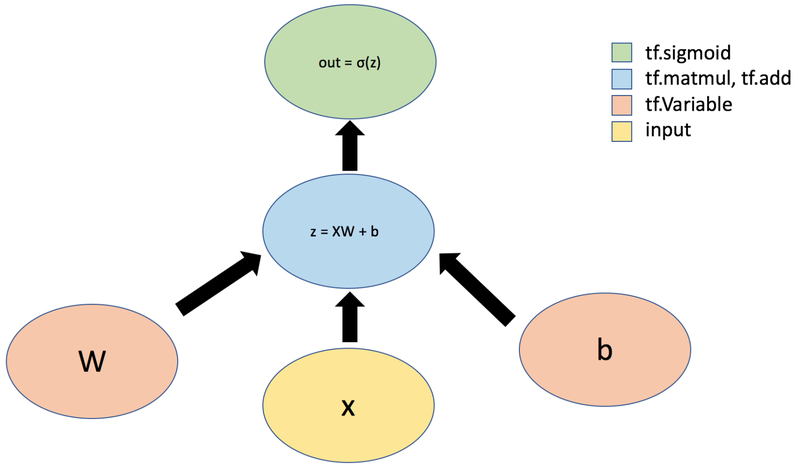
With the Sequential API, you can readily create neural networks by stacking together layers like building blocks.

TENSORFLOW

Neural Networks in TensorFlow

Using Keras's Sequential API

With the Sequential API, you can readily create neural networks by stacking together layers like building blocks.




```
### Defining a neural network using the Sequential API ###

# Import relevant packages
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# Define the number of outputs
n_output_nodes = 3

# First define the model
model = Sequential()

dense_layer = Dense(n_output_nodes, activation='sigmoid')

# Add the dense layer to the model
model.add(dense_layer)
```

We've defined our model with the Sequential API. Now, we can test it on an example input.

```
# Test model with example input
x_input = tf.constant([[1,2.]], shape=(1,2))

# feed input into the model and predict the output!
model_output = model(x_input).numpy()
# model_output = # TODO
print(model_output)
```

OUTPUT:
[[0.5607363 0.6566898 0.1249697]]

TENSORFLOW

Automatic Differentiation in TensorFlow

Automatic differentiation is one of the most important parts of TensorFlow and is the backbone of training with backpropagation.

We will use the TensorFlow GradientTape tf.GradientTape to trace operations for computing gradients

What's a GradientTape?

Tensorflow "records" all operations executed inside the context of a tf.GradientTape onto a "tape".

Tensorflow then uses that tape and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation using reverse mode differentiation.

Automatic Differentiation in TensorFlow

What's a GradientTape?

When a forward pass is made through the network, all forward-pass operations get recorded to a "tape"; then, to compute the gradient, the tape is played backwards.

By default, the tape is discarded after it is played backwards; this means that a particular tf.GradientTape can only compute one gradient, and subsequent calls throw a runtime error.

However, we can compute multiple gradients over the same computation by creating a persistent gradient tape.

Automatic Differentiation in TensorFlow

First, we will look at how we can compute gradients using GradientTape and access them for computation.

We define the simple function $y=x^2$ and compute the gradient:

```
### Gradient computation with GradientTape ###
# y = x^2
# Example: x = 3.0
x = tf.Variable(3.0)

# Initiate the gradient tape
with tf.GradientTape() as tape:
  # Define the function
  y = x * x

# Access the gradient -- derivative of y with respect to x
dy_dx = tape.gradient(y, x)
assert dy_dx.numpy() == 6.0
```

Automatic Differentiation in TensorFlow

Now that we have a sense of how GradientTape can be used to compute and access derivatives, **let's look at an example where we use automatic differentiation and SGD to find the minimum of $L=(x-x_f)^2$.**

Here x_f is a variable for a desired value we are trying to optimize for; L represents a loss that we are trying to minimize.

TENSORFLOW

Automatic Differentiation in TensorFlow

Let's look at an example where we use automatic differentiation and SGD to find the minimum of $L=(x-x_f)^2$.

```
### Function minimization with automatic differentiation and SGD ###

# Initialize a random value for our initial x
x = tf.Variable([tf.random.normal([1])])
print("Initializing x={}".format(x.numpy()))

learning_rate = 1e-2 # learning rate for SGD
history = []
# Define the target value
x_f = 4

# We will run SGD for a number of iterations. At each iteration, we compute the loss,
#   compute the derivative of the loss with respect to x, and perform the SGD update.
for i in range(500):
  with tf.GradientTape() as tape:
    loss = (x - x_f)**2 # "forward pass": record the current loss on the tape

  # loss minimization using gradient tape
  grad = tape.gradient(loss, x) # compute the derivative of the loss with respect to x
  new_x = x - learning_rate*grad # sgd update
  x.assign(new_x) # update the value of x
  history.append(x.numpy()[0])
```

TENSORFLOW

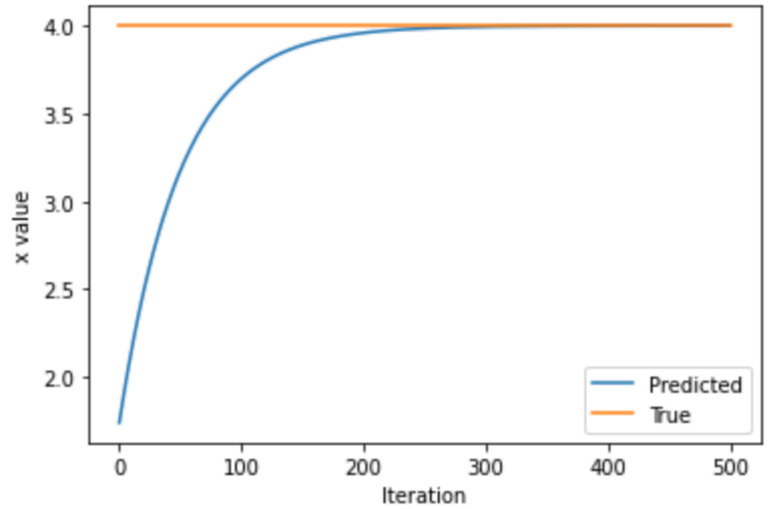
Automatic Differentiation in TensorFlow

Let's look at an example where we use automatic differentiation and SGD to find the minimum of $L=(x-x_f)^2$.

```
### Function minimization with automatic differentiation and SGD ###

# Initialize a random value for our initial x
x = tf.Variable([tf.random.normal([1])])
print("Initializing x={}".format(x.numpy()))

learning_rate = 1e-2 # learning rate for SGD
history = []
# Define the target value
x_f = 4

# We will run SGD for a number of iterations. At each iteration, we compute the loss,
#   compute the derivative of the loss with respect to x, and perform the SGD update.
for i in range(500):
  with tf.GradientTape() as tape:
    loss = (x - x_f)**2 # "forward pass": record the current loss on the tape

  # loss minimization using gradient tape
  grad = tape.gradient(loss, x) # compute the derivative of the loss with respect to x
  new_x = x - learning_rate*grad # sgd update
  x.assign(new_x) # update the value of x
  history.append(x.numpy()[0])
```

```
# Plot the evolution of x as we optimize towards x_f!
plt.plot(history)
plt.plot([0, 500],[x_f,x_f])
plt.legend(('Predicted', 'True'))
plt.xlabel('Iteration')
plt.ylabel('x value')
```

```
Initializing x=[[1.6940167]]
Text(0, 0.5, 'x value')
```



WRAPPING UP ...

TensorFlow

- We will use TensorFlow (2.0) in HW5
- For additional resources on TensorFlow, please check out:

  - Quickstart: https://www.tensorflow.org/tutorials/quickstart/beginner
  - Guide: https://www.tensorflow.org/guide
  - Tutorials: https://www.tensorflow.org/tutorials