UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

SCS 3253 Analytics Techniques and Machine Learning

Module 10: Introduction to TensorFlow

Course Plan

Module Titles
Module 1 – Current Focus: Introduction to Machine Learning
Module 2 – End to End Machine Learning Project
Module 3 – Classification
Module 4 – Clustering and Unsupervised Learning
Module 5 – Training Models and Feature Selection
Module 6 – Support Vector Machines
Module 7 – Decision Trees and Ensemble Learning
Module 8 – Dimensionality Reduction
Module 9 – Introduction to Neural Networks & Deep Learning
Current Focus: Module 10 – Introduction to TensorFlow
Module 11 – Distributing TensorFlow, CNNs and RNNs
Module 12 – Final Assignment and Presentations (no content)

Learning Outcomes for this Module

- Understand low level TensorFlow for performing machine learning tasks
- Build custom TensorFlow models and customize training to build novel neural networks
- Use TensorBoard to visualize the training and computation

Topics in this Module

- **10.1** TensorFlow
- **10.2** Custom TensorFlow
- **10.3** Computing the Gradient
- **10.4** Training Loops
- **10.5** TensorBoard
- **10.6** Wrap-up

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 1

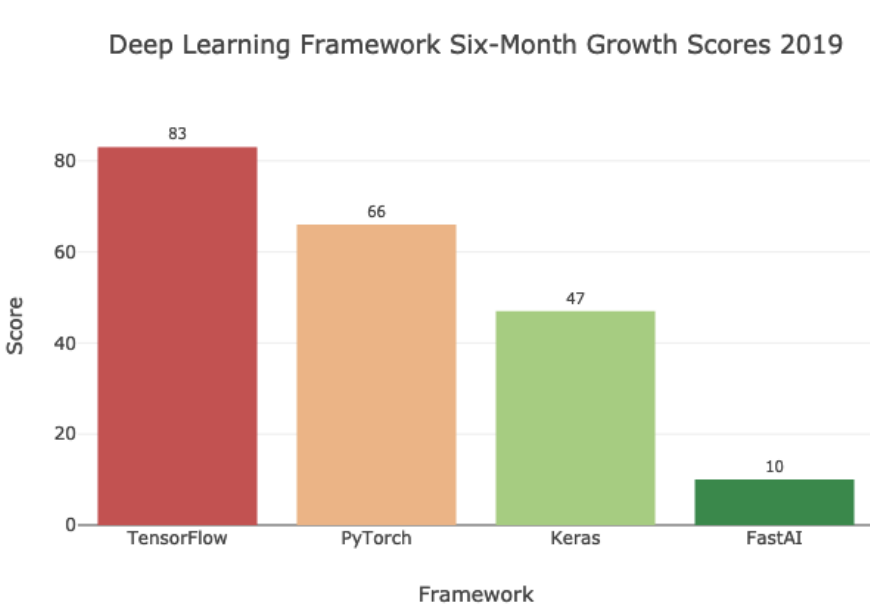
TensorFlow

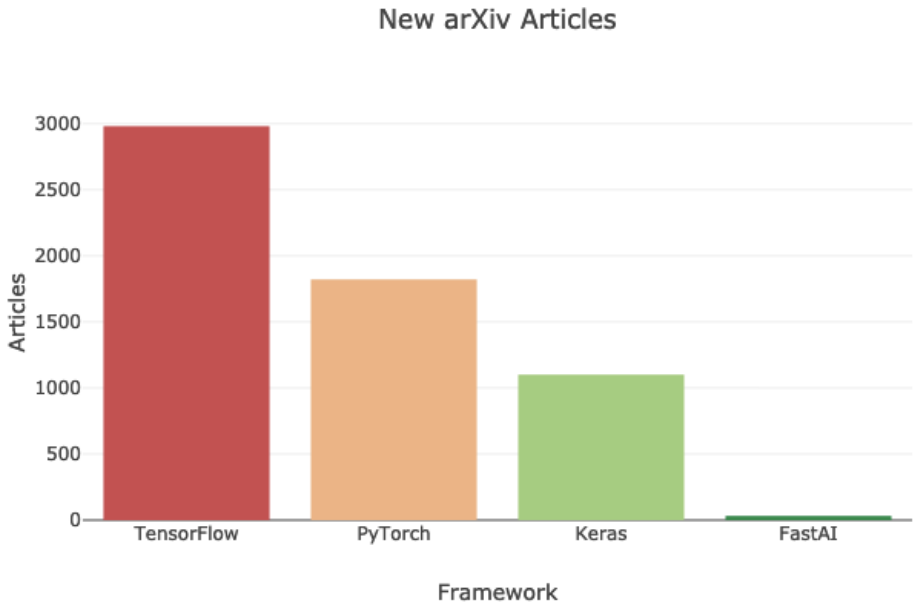
TensorFlow

- Open source machine learning and AI library from Google
- Works based on Graph in computer science to achieve higher efficiency
- Main Python API offers flexibility to perform diverse computations
- Includes highly efficient C++ implementations of many ML operations
- Provides several advanced optimizers to search for the parameters that minimize a cost function
- Comes with a great visualization tool called *TensorBoard*
- Several other high-level APIs have been built independently on top of TensorFlow, such as Keras, now the official recommended API
- Has a dedicated team of passionate and helpful developers including the team at Google, and a growing developer community
- One of the most popular open source projects on GitHub, and more and more great projects are being built on top of it

Other Libraries

- Great source of reference implementations: paperswithcode.com





https://towardsdatascience.com/which-deep-learning-framework-is-growing-fastest-3f77f14aa318

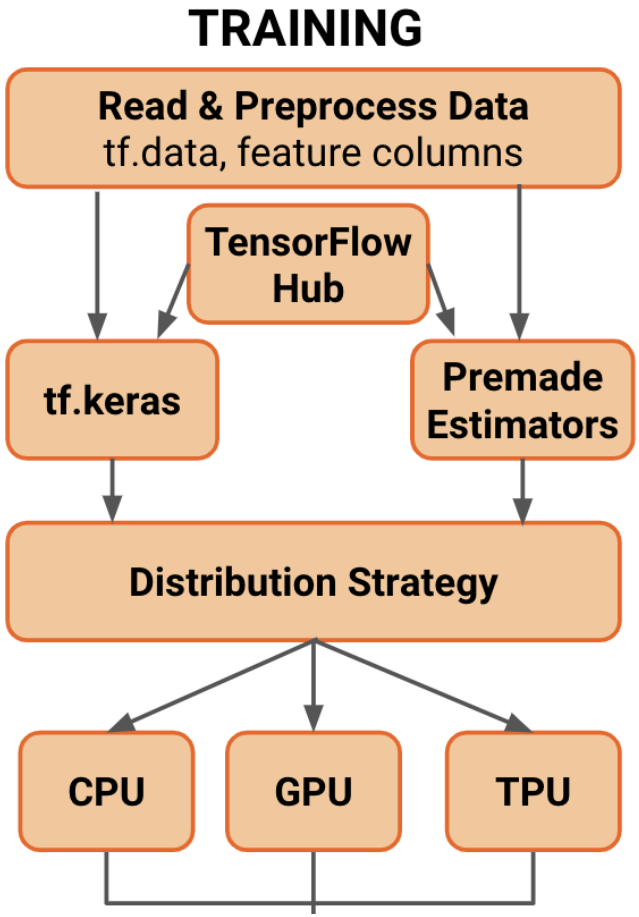
TensorFlow 1.0 - Deprecated

- Verbose Code
- Default Graph mode was difficult to debug code

```
n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()
```

TensorFlow 2.0

- New preprocessing tools with tf.data
- Transfer learning market with TensorFlow Hub
- Estimator library for pre-built models SKLearn style
- Keras with eager execution as default API
- Same performance graph core for training and deployment
- Support for CPUs, GPUs, and TPUs with the same code



https://medium.com/tensorflow/whats-coming-in-tensorflow-2-0-d3663832e9b8

What is a Tensor?

- Tensors are multi-dimensional arrays, including 0-dimensions: scalars
- Tensors are the output and inputs to every node in a neural networks, hence tensor flow
- Tensors work much like numpy ndarrays
- All the basic math operations are included as well

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
 tf.float32
```

Tensors

- Just like NumPy they can do:
  - Indexing
  - Math operations (+, -, *, /, @)
  - Equivalent methods like mean(), although some are renamed, e.g. .mean() is .reduce_mean() instead
- Can also use them interchangeably with NumPy functions and convert to NumPy using:

```
>>> t.numpy() # or np.array(t)
```

- **Warning**: tensors are sensitive to type so must use *tf.cast()* if doing an operation with another tensor type, e.g. *tf.float64* to *tf.float32*

Variable Tensors

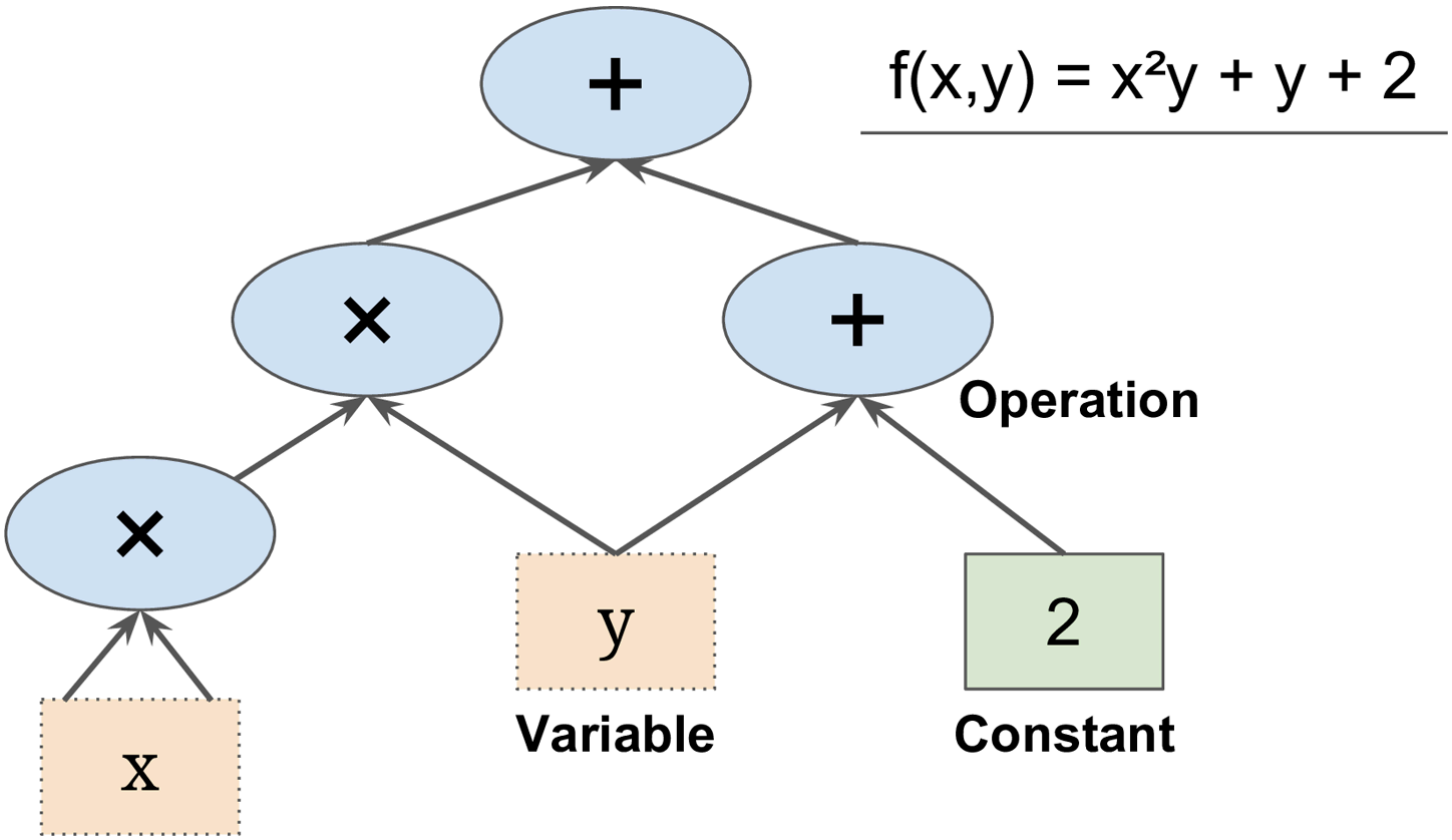
- Everything we have seen so far is immutable, it can't change, not suitable for model parameters!
- To make mutable tensors need to define *Variables*

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

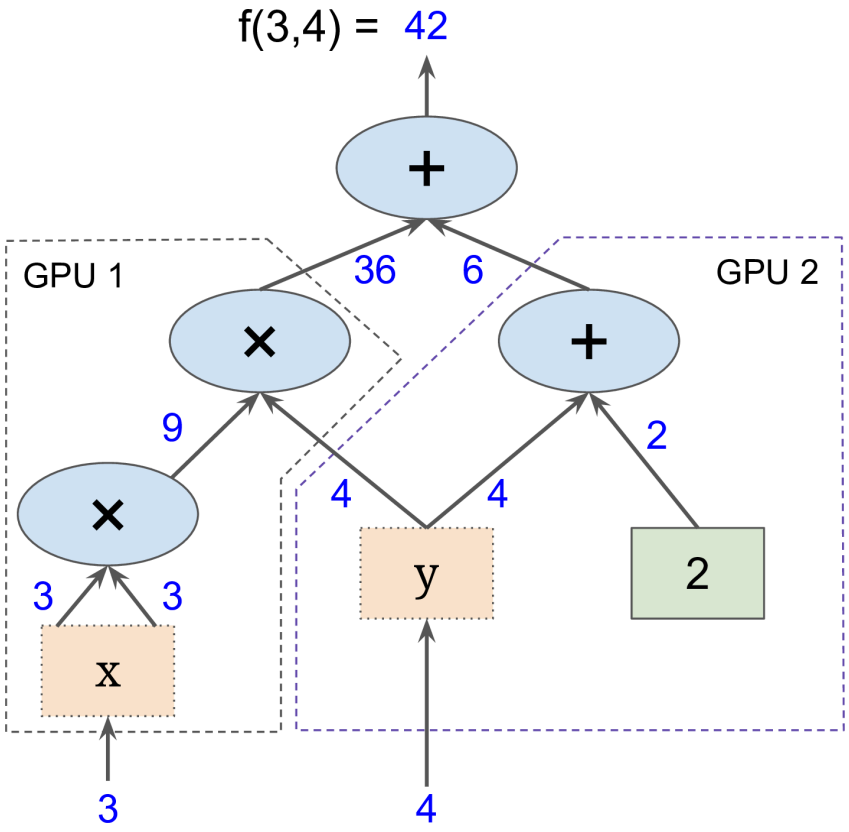
- They have an added assign method to allow changing their values

```
v.assign(2 * v)          # => [[2., 4., 6.], [8., 10., 12.]]
v[0, 1].assign(42)       # => [[2., 42., 6.], [8., 10., 12.]]
v[:, 2].assign([0., 1.]) # => [[2., 42., 0.], [8., 10., 1.]]
```

Tensorflow Graphs



Graphs - Parallel



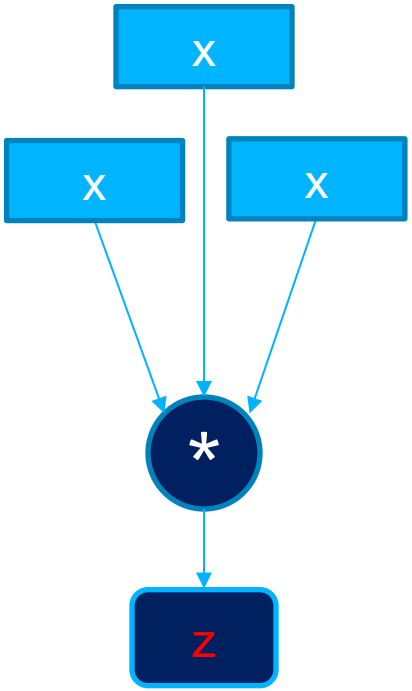
To confirm GPU computation run:

```
tf.config.experimental.list_physical_devices('GPU')
```

Defining a TensorFlow Graph

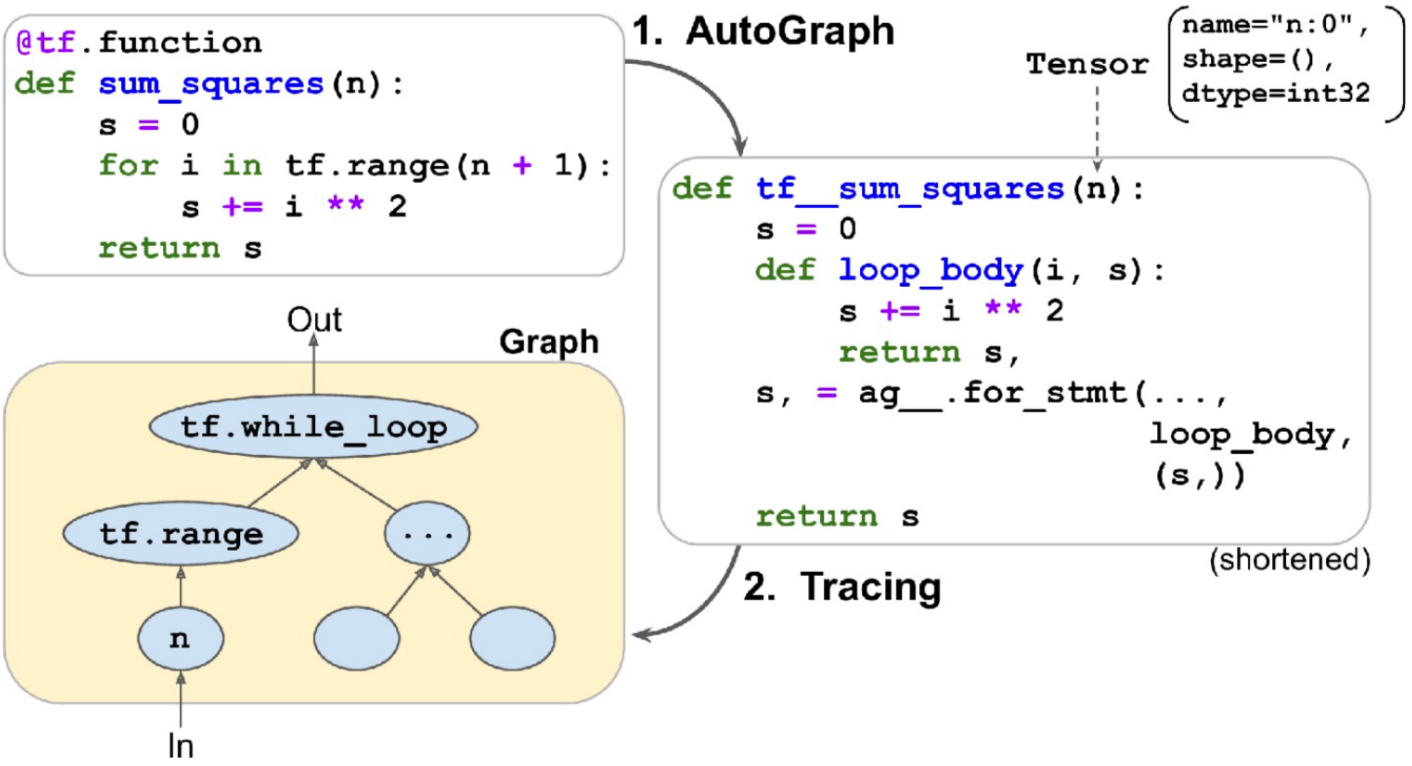
- Defining a graph in TF 2.0 uses AutoGraph
- Use @tf.function to executing function as a TF graph

```
@tf.function
def tf_cube(x):
    return x ** 3
```



- These TF functions always return Tensors
- This way regular python functions can be used to build the graph
- AutoGraph analyzes the python source code to build an equivalent graph for you
- We now have an efficient graph that can run on the GPU!

Defining a TensorFlow Graph



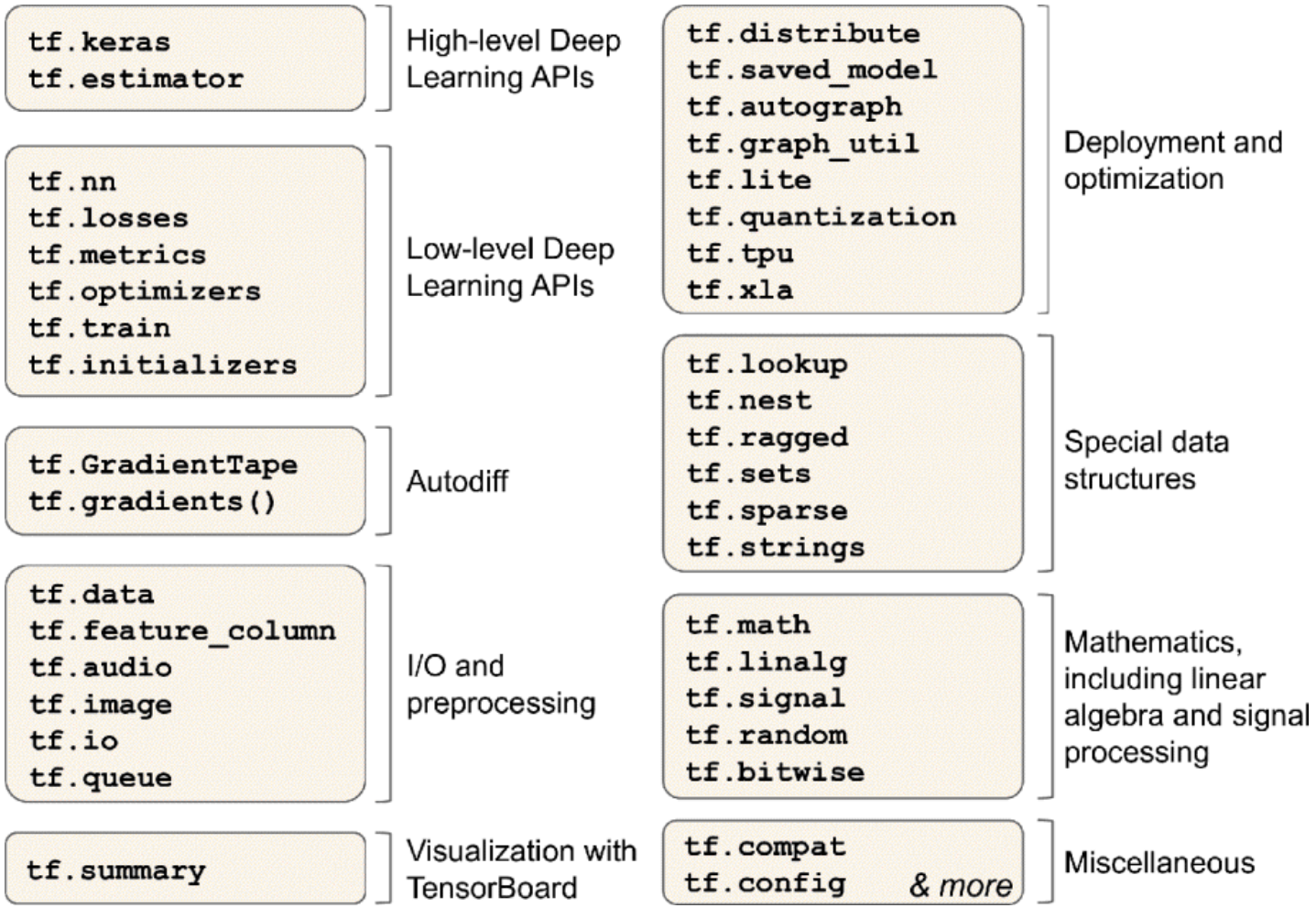
- In tf__sum_square function above, there is no for loop (loops are slow but graph is fast)
- You must use only TF methods and functions in order to see a benefit!

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

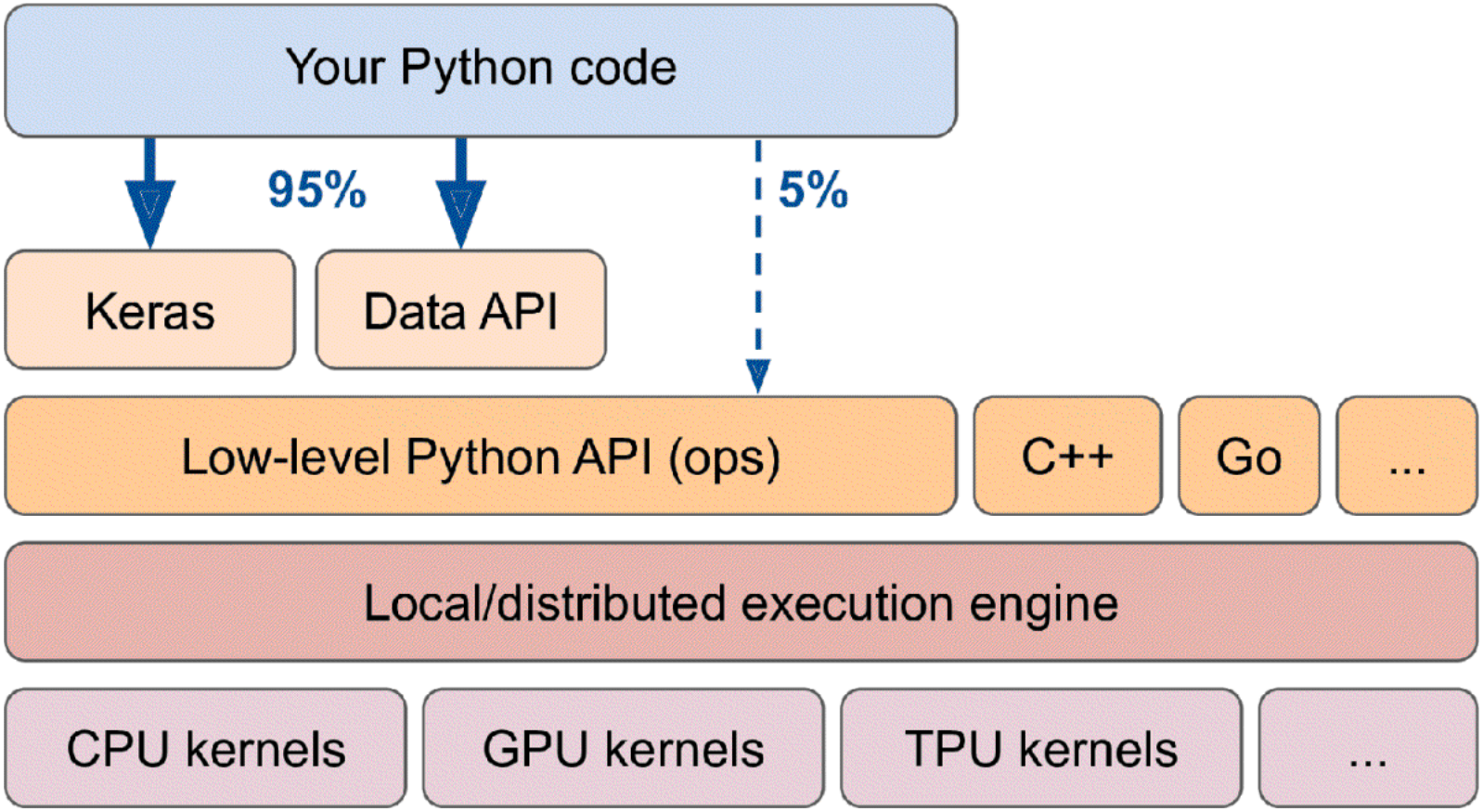
Module 10 – Section 2

Custom TensorFlow

Tensorflow's Python API



TensorFlow Architecture



Custom Loss Functions

- Defining a custom loss function is the same as defining any function in python
- However use vector operations and tensorflow functions as much as possible to take advantage of the performance boost of graphs

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss  = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```

- Use it in the same way as before:

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

Custom Model Components

- Adding other custom components like activations, initializers, regularizers and constraints is just as easy:

```
def my_softplus(z): # return value is just tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # return value is just tf.nn.relu(w
    return tf.where(weights < 0., tf.zeros_like(weights), weights)

layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)
```

Custom Metrics

- For metrics things are a bit different, remember losses are used to train the model so must be differentiable
- But metrics are used to evaluate the model so being human interpretable is more important
- Things get complicated when you need to track over multiple batches like for calculating of precision
- This is called the streaming metric and needs to be an object in order to keep track of state aka history

```
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

Custom Metrics

- Subclass* keras.metrics.Metric to create such an object
- You would then pass it to *compile()* like any other metric

Add_weight creates variable to track training over Epochs

```
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # handles base args (e.g., dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
    def result(self):
        return self.total / self.count
```

*Subclassing is a python way of inheritance in object-oriented programming model

Custom Layers

- Useful when groups of layers must be repeated
- Can turn simple functions into layers like so:

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

- More complex layers can be subclassed
  - Store the trainable parameters, weights and biases using the *add_weight()* method
  - Store the activation function
  - Create the forward pass method
  - Provide a way to calculate the shape
- For multiple inputs or multiple outputs the passed parameters need to be lists or tuples instead

Custom Layers

Tip: Adding a training argument to the call method lets you have different functionality in training vs. inference

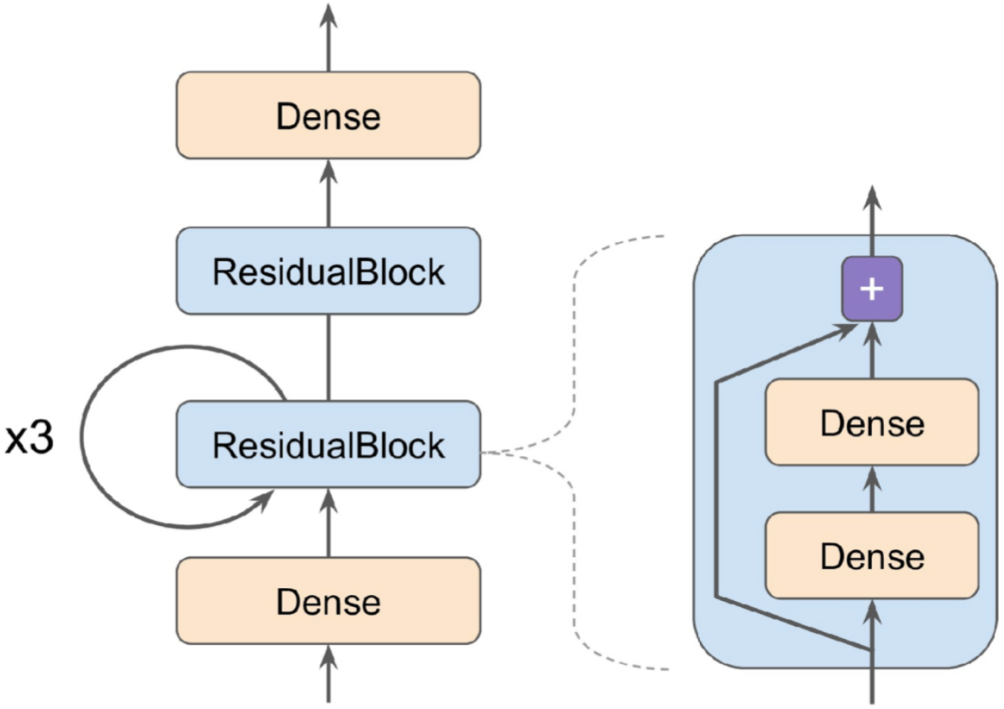
-build creates layer Variables using Add_weight
-Kernel is the weight matrix

-super().build tells parent class that layer is built

```
class MyDense(keras.layers.Layer):

    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape) # must be at the end

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])
```

Custom Models

- Already made custom models in the last class
- Really useful for complex architectures
- These techniques cover 99% of possible use cases!



ResidualBlock is the layer than adds an input to the output

Look at the Plus operation after 2nd Dense layer

Custom Models

Tip: One useful use case is to add internal model losses to use during training, like a reconstruction error or regularizer

```
class ReconstructingRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                          kernel_initializer="lecun_normal")
                       for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 3

Computing the Gradient

Computing Differentiation

- In order to optimize neural networks (find an appropriate set of weights) need to differentiate
- Three popular techniques to compute differentiation
  - Numerical differentiation
  - Symbolic differentiation
  - Automatic differentiation (Autodiff)
    - Forward Autodiff
    - Reverse Autodiff

Numerical Differentiation

- In order to optimize neural networks (find an appropriate set of weights) need to differentiate
- Actual derivative of a function

$$f'(x_0) = \lim_{h \to 0} (f(x_0 + h) - f(x_0))$$

- Can be approximated by forward difference (among others techniques)

$$f'(x) \approx \frac{f(x_0+h) - f(x_0)}{h}$$

Numerical Example

Consider **f** as a function with w1 and w2

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

- Approximation method gives below numbers

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

Numerical differentiation shortcomings?

- It is approximation, so it is not accurate (look next slide for symbolic and numerical diff error)
- It is resource intensive, especially when the model is very complex. Approximation for a typically large neural network with tens of 1000 weights is very compute-intensive and challenging.

Symbolic Differentiation

- Symbolic differentiation finds the derivative of a given formula with respect to a specified variable, producing a new formula as its output.

$$\frac{\partial f}{\partial w1} = 6 * \text{w1} + 2 * \text{w2}$$

$$\frac{\partial f}{\partial w2} = 2 * \text{w1}$$

- At w1=5, w2=3 this gives 36 and 10 respectively

- In general, symbolic mathematics programs manipulate formulas to produce new formulas, rather than performing numeric calculations based on formulas.

Symbolic Differentiation

- Consider another function & use symbolic diff

$$f(x) = \exp(\exp(\exp(x)))$$

- Derivative is

$$f'(x) = \exp(x) * \exp(\exp(x)) * \exp(\exp(\exp(x)))$$

- Evaluating naively would involve calling $\exp()$ **six times** to evaluate $f'(x)$
- Better solution: evaluate first term $\exp(x)$ then use that to evaluate $\exp(\exp(x))$ and so on
- Can use TF graph to take care of this

Symbolic Differentiation Shortcoming

- Previous example was simple
- Gets worse when encountering something like

```
def my_func(a, b):
    z=0
    for i in range(100):
      z = a * np.cos(z+i) + z * np.sin(b-i)
    return z
```

- Calculating such a function even using symbolic differentiation is intense!
- Clearly auto-differentiation would help here!

Automatic differentiation (Autodiff)

- Autodiff finds an algorithmic way to differentiate
- One of the biggest benefits of using computation graph is ability to compute symbolic gradients
- If we define a set of operations, where w1 and w2 are weights respectively and z is output we can compute the gradients as follows using GradientTape:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

```
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

- **GradientTape** keeps tracks of calculations on weights
- Use tape to calculate gradient

Warning: Takes a lot of memory and clears memory when done

Forward-Mode Autodiff

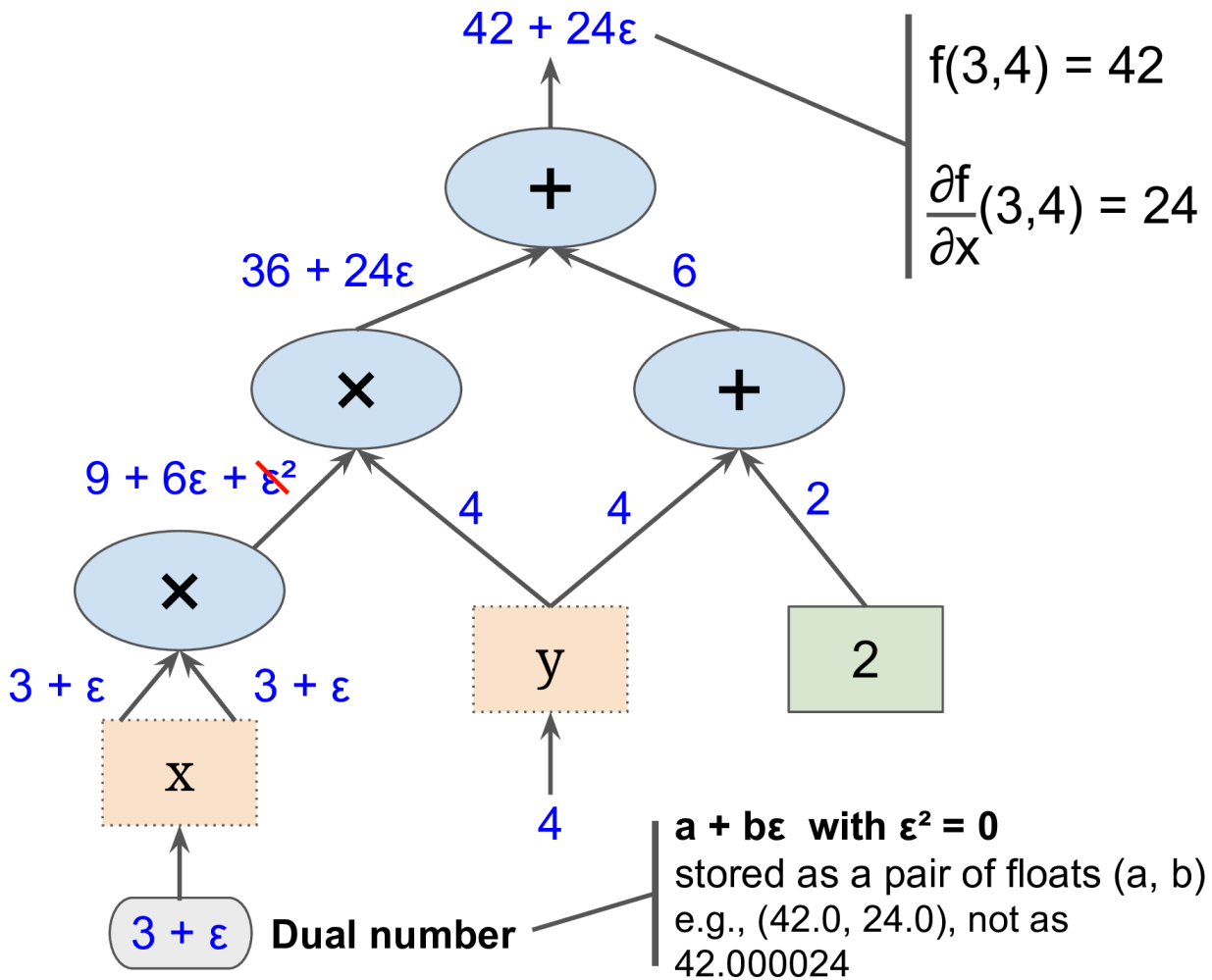
- It relies on *dual numbers*, which are numbers of the form $a + b\epsilon$ where a and b are real numbers and ϵ is an infinitesimal number such that $\epsilon^2 = 0$ (but $\epsilon \neq 0$)
- You can think of the dual number $42 + 24\epsilon$ as something akin to 42.0000⋯000024 with an infinite number of 0s
- Using below algebra rules, It can be shown that

$$h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$$

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$
$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$
$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

- So computing $h(a + \epsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot

Forward-Mode Autodiff (From input to output)



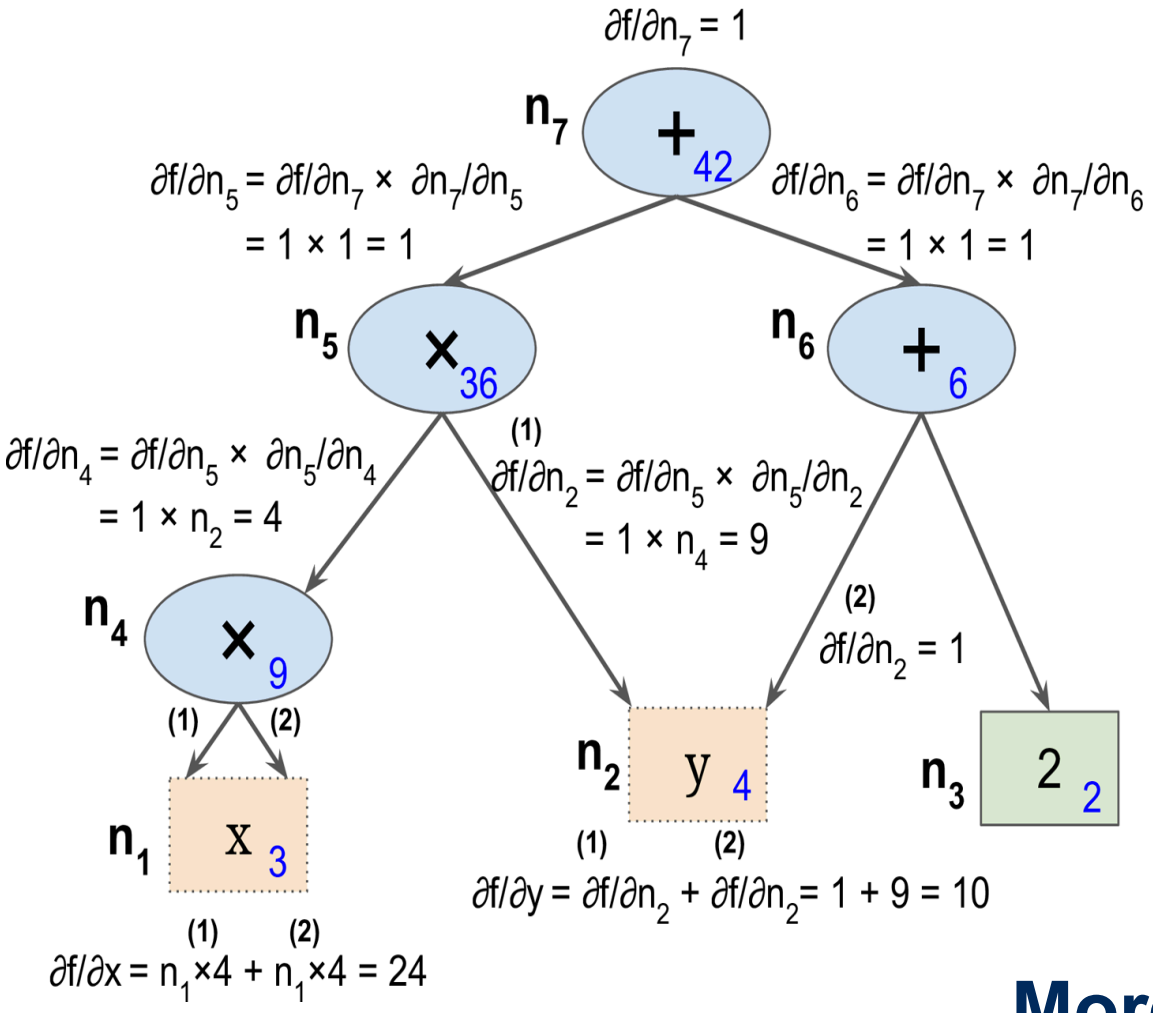
Forward-Mode Autodiff Shortcoming

- For each gradient, the graph has to be parsed
- When number of inputs is significantly larger than output (in deep learning neural net), computation is becoming a bottleneck for forward-mode autodiff

Reverse-Mode Autodiff (From output to input)

- Implemented by TensorFlow
- Runs in two step:
  - Forward from Input to output to calculate value of each nodes
  - Reverse mode: Calculate the derivatives of each node

Reverse-Mode Autodiff (From output to input)



- Value of N_7 is the output
- Values calculated in step 1 is on bottom right of each node in blue
- Step 2 goes from n_7 to inputs and calculates partial derivatives
- Relies on chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

More scalable!

Optimizers

- In reality, we won't typically call tape.gradient() or write our own gradient descent code
- Tensorflow comes bundled with optimizers out of the box

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

- In custom training loops you do need to however as we will see

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 4

Training Loop

Default Training Loop

- Keras.fit() makes training easy but doesn't give flexibility if you have special requirements
- Also fit function does not give you full view under the hood

```
 model.compile(loss="sparse_categorical_crossentropy",
               optimizer="sgd",
               metrics=["accuracy"])                              ))

>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [======] - 3s 49us/sample - loss: 0.7218     - accuracy: 0.7660
                                     - val_loss: 0.4973 - val_accuracy: 0.8366
```

Training Steps: How does fit() works?

1. Loop over the epochs
2. Loop over the number of batches
3. Create a random mini-batch of data
4. While tracking gradients:
   a. Make a prediction for the batch
   b. Compute the prediction loss
5. Compute gradients for all trainable variables
6. Use the optimizer to make a gradient descent step on the variables
7. Update and display mean loss and other metrics

Custom Training Loop

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {}/{}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

Custom Training Loop

- With a custom training loop you do not need to compile the model first
- With tf.data.Dataset you can create mini-batches with the batch method instead
- You must define all the hyper parameters for training upfront:

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

Extra Training Items

- Add other model losses (e.g. L2 regularization) after computing the losses from the output
- Modify the gradients (e.g. gradient clipping) before applying them with the optimizer
- Add weight constraints (e.g. max norm regularization) just after applying them with the optimizer
- For layers that behave differently in training vs testing (e.g. batchnorm or dropout) you must pass the training=True hyperparameter

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 5

TensorBoard

Tensorboard

- Useful to track progress of training/validation losses visually during long training runs
- To use it you must create binary log files, each is referred to as a summary
- Tensorboard server monitors for these log files in a log directory and reads in any updates as they happen
- Usually you want to keep each run in a separate directory so that they're not overwritten

```
my_logs/
├── run_2019_06_07-15_15_22
│   ├── train
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
│   │   └── plugins/profile/2019-06-07_15-15-32
│   │       └── local.trace
│   └── validation
│       └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
└── run_2019_06_07-15_15_49
    └── [...]
```

Tensorboard Callback

- Keras lets you use tensorboard easily with the use of callbacks; just provide a directory name
- Callbacks are typically run at the end of every epoch

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # e.g., './my_logs/run_2019_06_07-15_15_22'

tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[tensorboard_cb])
```

Checkpointing & Early Stopping Callback

- Another great use of callbacks by the way is checkpointing:

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
```

- Another use is early stop:

-Patience: number of epoch after which no improvement in model leads to stop the training

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

Tensorboard Server

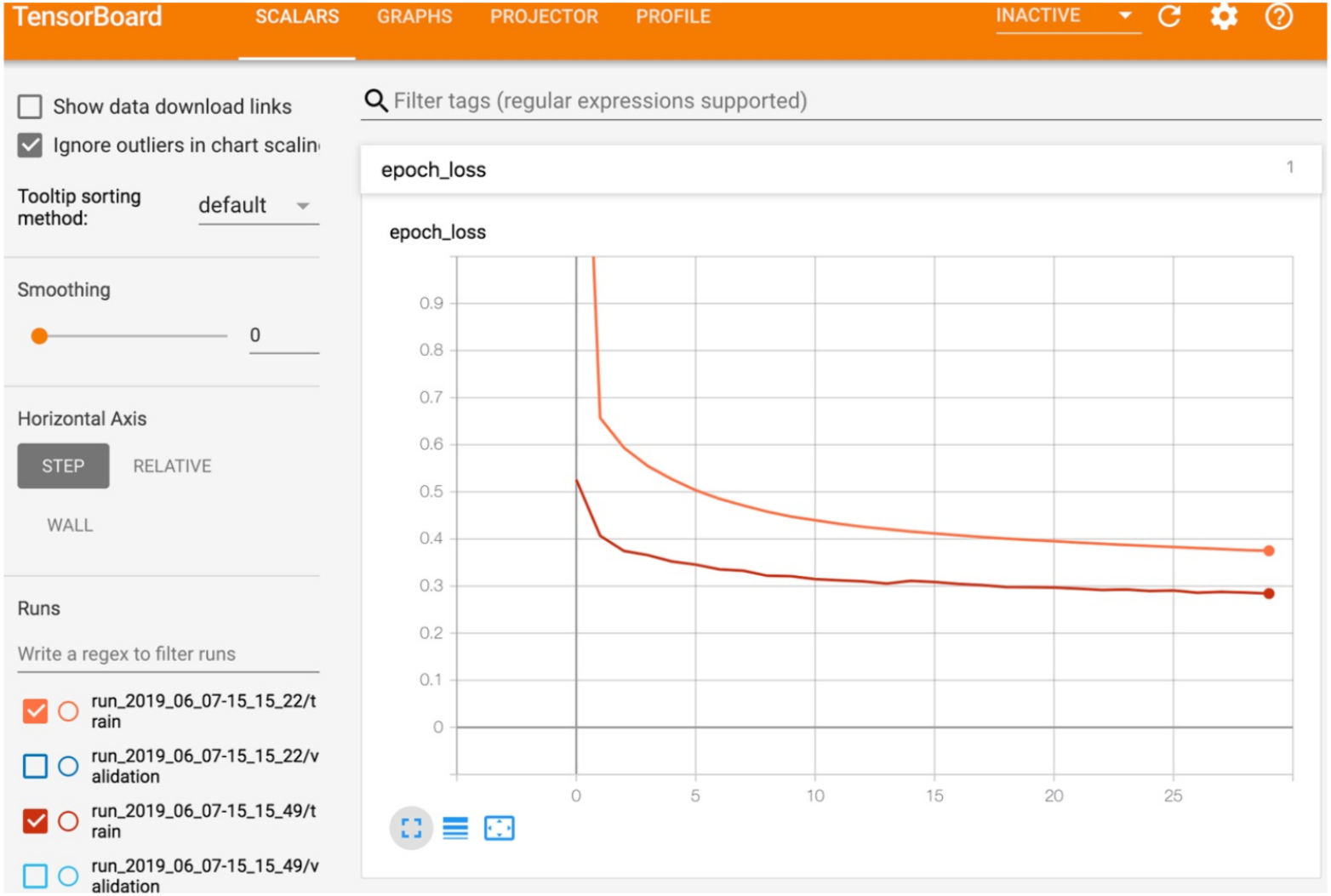
- The first way to access is from the command line in a similar way to launching Jupyter
- Then open *http://localhost:6006* in your browser

```
$ tensorboard --logdir=./my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)
```

- commands shown
- Useful when working in Jupyter often

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs --port=6006
```

Tensorboard



Module 10 – Section 6

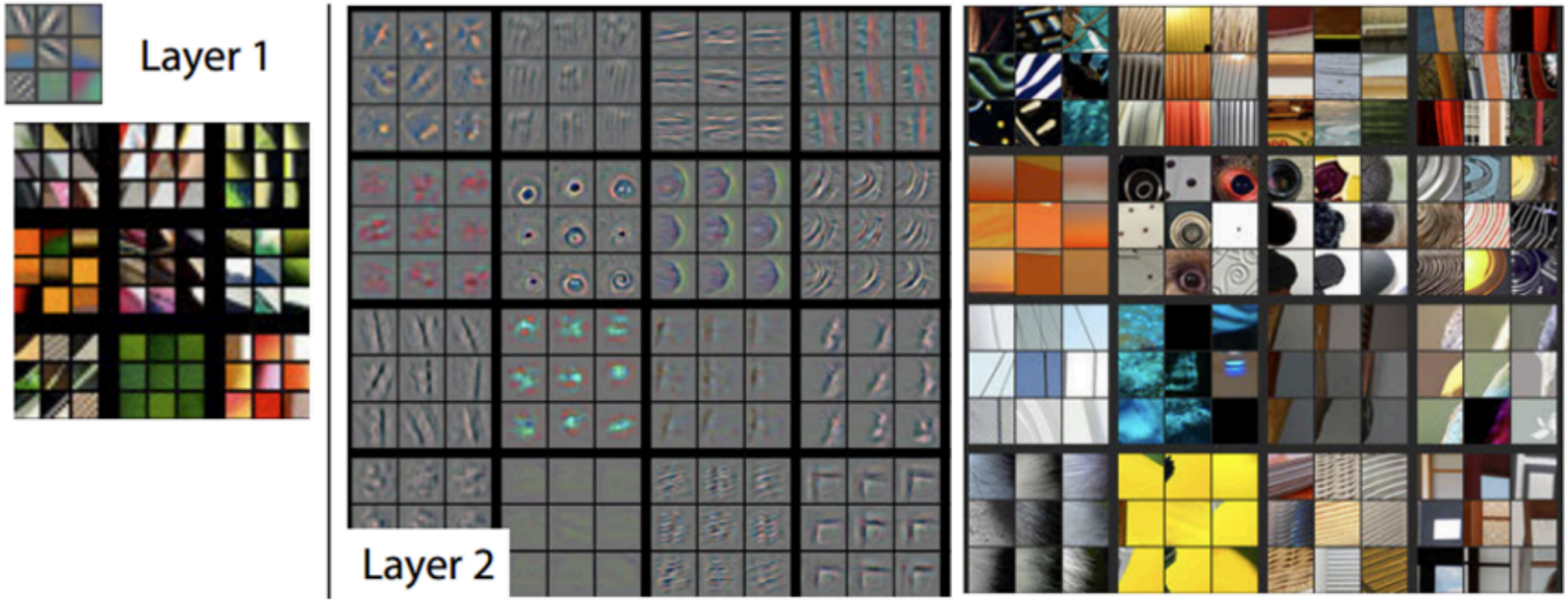
Wrap up

Homework

- Keep working on your Term Project!

Next Class

- Distributing TensorFlow, CNNs & RNNs
  - Solving cutting edge problems with deep learning
- Read textbook chapters 11 & 12 in preparation



Any questions?

Join the conversation with us online:

facebook.com/uoftscs

@uoftscs

linkedin.com/company/university-of-toronto-school-of-continuing-studies

@uoftscs

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Thank You

Thank you for choosing the University of Toronto School of Continuing Studies