Introduction to Tensorflow

Narada Warakagoda

What is Tensorflow?

- An open source software library from Google.
- Supports deep learning and some machine learning algorithms.
- Can be run on multiple machines and GPUs.
- Several APIs Python, Java and C++.
- Tensorflow Python API is tightly integrated with Keras, a high level set of APIs for deep learning.
- Latest version (as of July 2022) is 2.9
- For more information refer https://www.tensorflow.org/api_docs/python/tf

Tensorflow execution modes

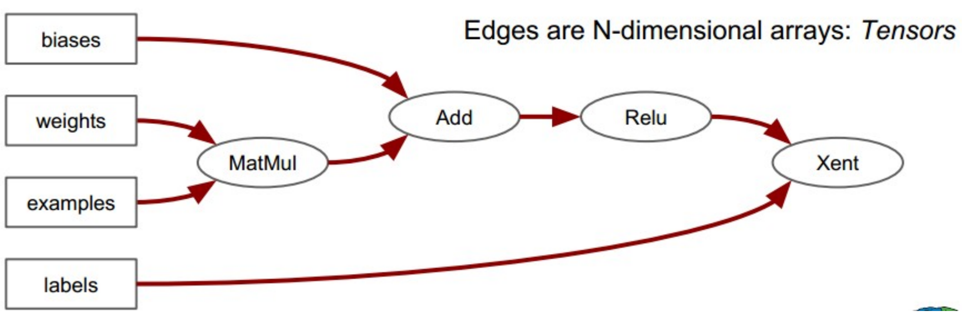
- Graph execution mode
  - Consists of two phases:
    - Creation of a computation graph
    - Run the data through the computation graph
  - Natural way of supporting Automatic Differentiation
  - Early Tensorflow versions supported this mode only
- Eager execution mode
  - No execution phases similar to other languages such as plain Python
  - Useful in debugging
  - Generally slower than Graph execution mode
  - Default mode in Tensorflow 2.x

Graph mode example

- Consider a very simple model $\hat{\mathbf{y}} = f(\mathbf{xW} + \mathbf{b})$
- Loss function $C = g(\hat{\mathbf{y}}, \mathbf{y})$

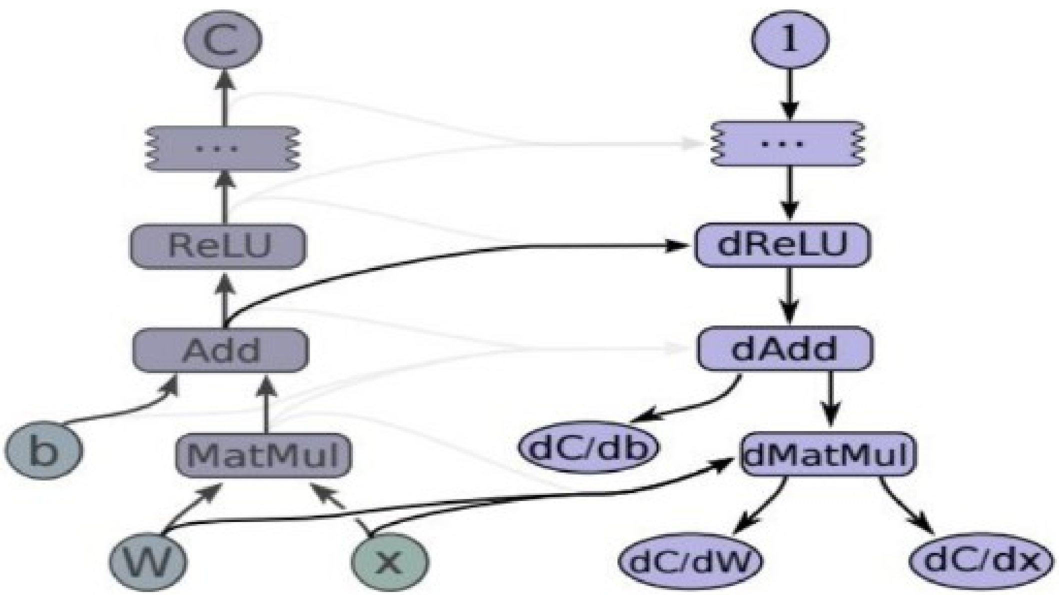
where

- $\mathbf{x} = [x_1, x_2, \cdots, x_L$ is the input training example
- $\mathbf{y} = [y_1, y_2, \cdots, y_K]$ is the label of the example
- $\mathbf{W}$ is the weight matrix of the network
- $\mathbf{b}$ is the bias vector of the network
- $f(\cdot)$ is the activation function and assume that it is a ReLu
- $g(\cdot)$ is the loss function, and assume that it represents cross entropy Xent



Optimization with the computation graph

- Gradient computation graph is automatically generated (i.e. Automatic Differentiation)



User defined
forward computaion graph

Tensorflow defined
gradient computaion graph

Advantages of Graph mode

- Flexibility (i.e. not dependent on Python)
- Can be exploited on different devices
- Used in exporting and importing models
- Can easily be optimized

Enforcing Graph mode

- Use `tf.function` directly or as a Python decorator

```python
# Define a Python function.
def a_regular_function(x, y, b):
  x = tf.matmul(x, y)
  x = x + b
  return x

# ‘a_function_that_uses_a_graph‘ is a TensorFlow ‘Function‘.
a_function_that_uses_a_graph = tf.function(a_regular_function)
```

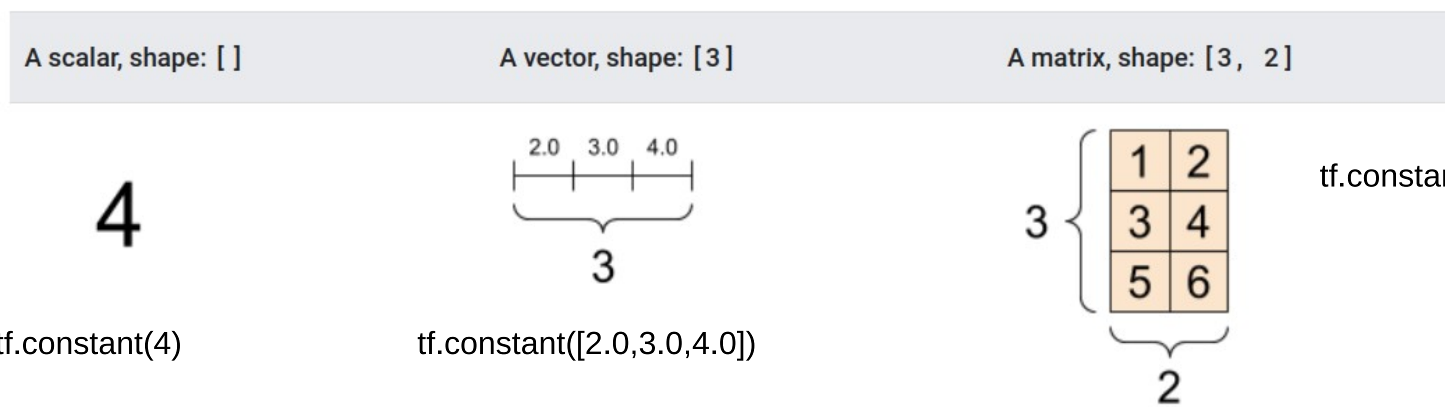
```python
# Define a Python function. with decoration
@tf.function
def a_regular_function(x, y, b):
  x = tf.matmul(x, y)
  x = x + b
  return x

# ‘a_regular_function‘ is a TensorFlow ‘Function‘.
```

Eager Execution mode

- This is the default mode of Tensorflow
- Code is executed line by line
- Since there is no graph, Tensorflow needs to “record” all relevant operations to support Automatic Differentiation
  - `tf.GradientTape` is the data structure used for this purpose.

Tensorflow Tensors

- Main data structure of Tensorflow is a Tensor
- Tensors can be created using different functions
  - `tf.constant`
  - `tf.variable`
  - `tf.zeros`
  - `tf.ones and more ..`

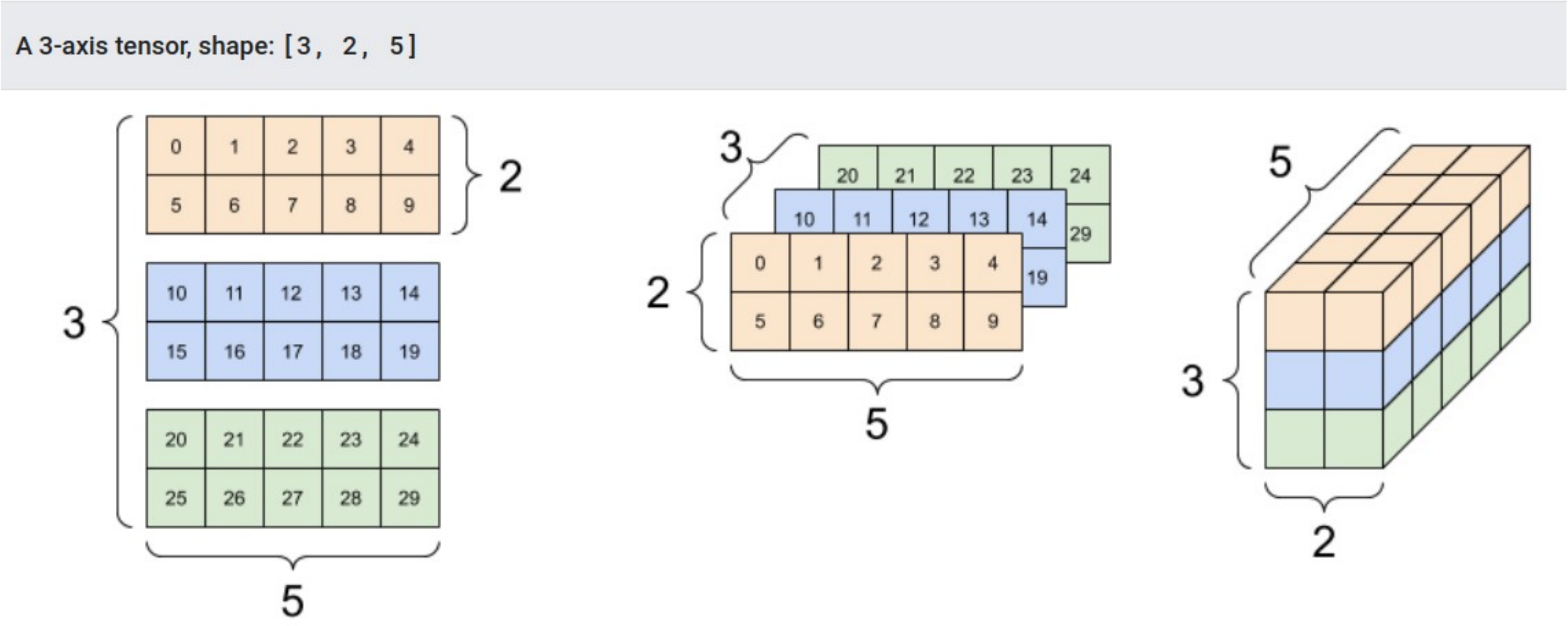


tf.constant(4)

tf.constant([2.0,3.0,4.0])

tf.constant([[1,2],
[3,4],
[5,6]])

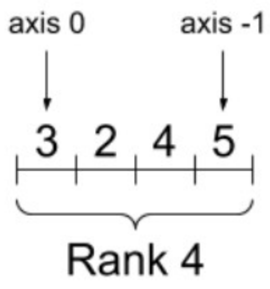
Tensorflow rank-3 Tensor

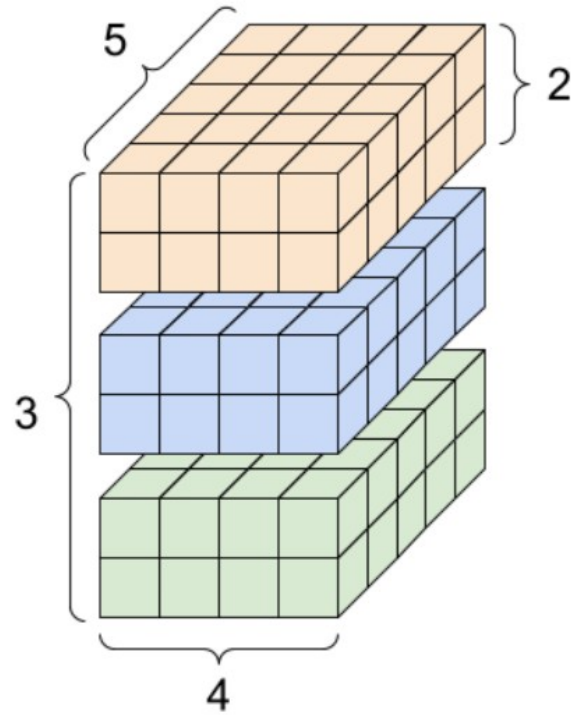



```
rank_3_tensor = tf.constant([
                                        [[0, 1, 2, 3, 4],
                                         [5, 6, 7, 8, 9]],
                                        [[10, 11, 12, 13, 14],
                                         [15, 16, 17, 18, 19]],
                                        [[20, 21, 22, 23, 24],
                                        [25, 26, 27, 28, 29]],
                                    ])
```

Tensorflow rank-4 Tensor

A rank-4 tensor, shape: `[3, 2, 4, 5]`






```
tf.zeros([3, 2, 4, 5])
```

Main Steps in training a model with Tensorflow

- Define the model architecture
- Define the loss function
- Choose and optimizer and learning rate
- Create the data feeding code
- Write the training/validation loop
  - Training step
    - Run the next training mini-batch through the model
    - Calculate the loss
    - Calculate the gradients and update model parameters
    - Update loss (and metric) summaries
  - Validation step
    - Run the next validation mini-batch through the model
    - Calculate the loss
    - Update loss (and metric) summaries

Model Architecture (Layers)

- Typically models are composed using `tf.keras.layers` which can be
  - Predefined layers (eg: `tf.keras.layers.Conv2D` , `tf.keras.layers.Dense`)
  - Custom layers (create your own layers by extending `tf.keras.layers.Layer`)

Pre-defined Layers Example (2D Convolution)

```
conv = tf.keras.layers.Conv2D(
  filters, # e.g. 64
  kernel_size, # e.g. (3, 3)
  strides=(1, 1),
  padding='valid', # other option is 'same'
  data_format=None,
  dilation_rate=(1, 1),
  activation=None,
  use_bias=True,
  kernel_initializer='glorot_uniform',
  bias_initializer='zeros',
  kernel_regularizer=None,
  bias_regularizer=None,
  activity_regularizer=None,
  kernel_constraint=None,
  bias_constraint=None,
  **kwargs)
```

Predefined layer example (Dropout)

```
dropout = tf.keras.layers.Dropout(0.5)
tf.random.set_seed(123)
x = [[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]]
dropout(x, training=True)  # [[0., 2., 0.], [2., 2., 2.]]
dropout(x, training=True)  # [[2., 0., 2.], [2., 2., 0.]]
dropout(x, training=False) # [[1., 1., 1.], [1., 1., 1.]]
```

Pre-defined layer example (Batch Normalization)

```
batch_norm = tf.keras.layers.BatchNormalization(axis=-1,
↪ center=False, scale=False)
x = np.array([[-1.0, 4.0, 1.0], [1.0, -4.0, 3.0]])
# [batch_size, num_features] == [2, 3]
# feature 1: mean = 0, std ~= 1
# feature 2: mean = 0, std ~= 4
# feature 3: mean = 2, std ~= 1
batch_norm(x, training=True) # ~= [[-1.0, 1.0, -1.0], [1.0,
↪ -1.0, 1.0]]
batch_norm(x, training=False) # ~= ?
```

Keras custom layer (example)

```python
class FullyConnected_v2(tf.keras.layers.Layer):
  def __init__(self, num_outputs):
    super(FullyConnected_v2, self).__init__()
    self.num_outputs = num_outputs

  def build(self, input_shape):
    """Assume input_shape[0] is batch size and input_shape[1] is size of input samples."""
    self.W = tf.Variable(np.random.uniform(-0.1, 0.1, size=[input_shape[1], self.num_outputs]))
    self.b = tf.Variable(np.zeros(self.num_outputs))

  def call(self, x):
    return tf.matmul(x, self.W) + self.b

# array of shape [batch_size, num_inputs] ==> [batch_size, 5]
fc = FullyConnected_v2(num_outputs=5)
# array of shape [2, 3] ==> [2, 5]
fc(np.array([[1.0, 0.4, 0.2], [-0.4, 0.3, 0.2]]))
```

Notes on custom layers

- The build method is called the first time __call__ is called.
- We implement call rather than __call__. The __call__ metod is implemented in the parent class, and will call ’call’
- We may also use self.add_variable method to add variables to layer
- Remember to call super method in __init__ to initialize Layer class properly.
- Optional: Implement get_config and compute_output_shape (for serialization and model summary purposes respectively).

Model Architecture (Models)

- Once you have all the layers, compose the intended model using one of
  - Keras sequential API
  - Keras Functional API
  - Extending `tf.keras.Model`

Composing model (sequential API)

- Just add your layers to the `tf.keras.models.sequential` call

```
from tensorflow.keras.layers import Conv2D, Flatten, Dense

model = tf.keras.models.Sequential([
  Conv2D(32, kernel_size=3, activation='relu'),
  Conv2D(64, kernel_size=3, strides=2, activation='relu'),
  Flatten(),
  Dense(128, activation='relu'),
  Dense(10, activation='softmax')
])
```

Composing model (Functional API)

- Need to define a dummy input (specify the size without batch size)
- Then call the layers one after other

```
inputs = tf.keras.Input(shape=(32, 32, 3))
x = Conv2D(32, kernel_size=3, activation='relu')(inputs)
x = Conv2D(64, kernel_size=3, strides=2, activation='relu')(x)
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
outputs = Dense(10, activation='softmax')(x)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Composing model (Extending tf.keras.Model)

- Need to implement __init__() and call() methods.

```
class MyModel(tf.keras.Model):
  def __init__(self):
    super(MyModel, self).__init__()
    self.conv1 = Conv2D(32, kernel_size=3, activation='relu')
    self.conv2 = Conv2D(64, kernel_size=3, strides=2, activation='relu')
    self.flatten = Flatten()
    self.d1 = Dense(128, activation='relu')
    self.d2 = Dense(10, activation='softmax')

  def call(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.flatten(x)
    x = self.d1(x)
    return self.d2(x)
model = MyModel()
```

Loss function

- tf.keras.losses has many pre-defined loss functions to choose from
  - MeanSquaredError (MSE)
  - SparseCategorocalCrossEntropy (CE when labels are sparse)
  - CategoricalCrossEntropy (CE when labels are one-hot encoded)

```
mse = tf.keras.losses.MeanSquaredError()
loss = mse([0., 0., 1., 1.], [1., 1., 1., 0.])
print('Loss: ', loss.numpy())  # Loss: 0.75

cce = tf.keras.losses.SparseCategoricalCrossEntropy(from_logits=False) # set true if not softmax applied
loss = cce([0, 1, 2], [[.9, .05, .05], [.5, .89, .6], [.05, .01, .94]])
print('Loss: ', loss.numpy())  # Loss: 0.3239
```

Optimizer

- tf.keras.optimizers has several optimizers to choose from. Pick one of them and create an optimizer object.
- Optimizer implements an update rule only (Gradient computation is outside of its scope)

```
import tensorflow as tf

# Stochastic gradient descent, with or without momentum
op1=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.0,
                            nesterov=False, name='SGD', **kwargs)

op2=tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9,
                                momentum=0.0, epsilon=1e-07,
                                centered=False,name='RMSprop',**kwargs)

op3=tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,
                             beta_2=0.999, epsilon=1e-07,
                             amsgrad=False,name='Adam', **kwargs)
```

Data Feeding

- Typical data feeding pipeline is based on `tf.data.Dataset`
- Basic steps:
  - Create a `tf.data.Dataset` object with the data you have. It can "consume" several different sources such as
    - numpy arrays,
    - Python generators,
    - tfrecord data,
    - CSV data and
    - sets of files
  - Perform desired operations on the `Dataset` object (eg: `shuffle`, `map`, `repeat` and `batch`)
  - Iterate through the `Dataset` object to get data for training (or validation).

Data feeding example

- Dataset object consumes a Python generator
- Functions used in code are in the next slide

```python

#############################################################
# Create a Dataset object consuming Python generator
dataset = tf.data.Dataset.from_generator(
        generator_for_filenames(image_names, class_labels),
        output_types=(tf.string, tf.string),
        output_shapes=(None, None)
    )

# Shuffle data
dataset = dataset.shuffle(buffer_size=100)

# Map Dataset objects (file names to images)
dataset = dataset.map(read_image)

#Map Dataset objects (images to preprocessed images)
dataset = dataset.map(preprocess)

#Introduce batching
dataset = dataset.batch(batch_size)




#######################################################
# Now you can use the Dataset
for image, label in dataset:
    print(image, label)
    #Feed your data through the model etc.
    # pred=model(image)
```

Data feeding example (Auxiliary functions)

```python


def generator_for_filenames(*filenames):
    #Wrapping a list of filenames as a generator function
    def generator():
        for f in zip(*filenames):
            yield f
    return generator


def preprocess(image,class_lab):
    #A preprocess function the is run after images are read.
    #Here you can do augmentation and other pre-processing
    #on the images.

    # Set images size to a constant
    image = tf.image.resize(image, [HEIGHT, WIDTH])

    # Other pre-processing stuff here
    return image,class_lab


def read_image(img_f, class_lab):
    #Read images from file using tensorflow
    img_reader = tf.io.read_file(img_f)
    img = tf.image.decode_png(img_reader, channels=3)

    return img, class_lab
```

Training Loop

- There are two main alternatives
  - Use the predefined methods `tf.keras.model.compile` and `tf.keras.model.fit`
  - Write training steps using non-Keras (Pure Tensorflow) API
    - Run the next training mini-batch through the model
    - Calculate the loss
    - Calculate the gradients and update model parameters
    - Update loss (and metric) summaries
- Previous steps are common to any of these alternatives (i.e. definition of model, loss, optimizer and data feeder etc.)

Common steps

```
import tensorflow as tf
from tensorflow.keras import losses, metrics, optimizers


#define your model in get_model() function
model = get_model()

#define your loss function
loss_fn = losses.Crossentropy(from_logits=False)

#define your optimizer
optimizer_fn = optimizers.Adam(lr=1e-4)

#Create a Dataset object in get_train_data() function
train_dataset = get_train_data()

#create a Dataset object in get_val_data() function
val_dataset  = get_val_data()

# Create function to calculate an optional  custom metric
metric_fn = get_metric_fn()
```

Keras compile-fit

```
model.compile(
    optimizer=optimizer_fn,  # Optimizer
    loss=loss_fn,# Loss function to minimize
    metrics=[metric_fn],# List of metrics to monitor
)

history = model.fit(
    train_dataset,
    batch_size=64,
    epochs=2,
    validation_data=val_dataset,
)
```

Pure Tensorflow training loop

```
# Start train/validation loop, assume train_epochs=100
for epoch in range(train_epochs):

        for image, y in train_dataset:
                y_pred = train_step(image, y)
                step += 1


        # reset metrics
        train_loss.reset_states()
        train_accuracy.reset_states()

        # Do validation after each epoch
        for image, y in val_dataset:
                y_pred = val_step(image, y)

        # reset metrics
        val_loss.reset_states()
        val_accuracy.reset_states()
```

Train Step

Validation Step

Pure Tensorflow training step

```python
# Used to running averages for summaries
train_accuracy = metrics.Mean()
train_loss = metrics.Mean()
val_accuracy = metrics.Mean()
val_loss = metrics.Mean()


# Train step for a single batch
def train_step(image, y):

        # get predictions loss and update model parameters
        with tf.GradientTape() as tape:
                y_pred = model(image, training=True)
                loss = loss_fn(y, y_pred)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer_fn.apply_gradients(zip(grads, model.trainable_variables))

        # Update running averages for summaries
        m = metric_fn(y, y_pred)
        train_loss.update_state(loss)
        train_accuracy.update_state(m)

        return y_pred
```

Pure Tensorflow validation step

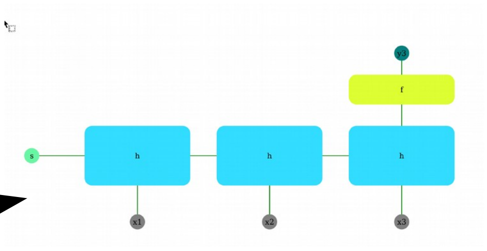
```
#validation step for a single batch
def val_step(image, y):

        y_pred = model(image)
        loss = loss_fn(y, y_pred)

        # Update running averages for summaries
        m = metric_fn(y, y_pred)
        val_loss.update_state(loss)
        val_accuracy.update_state(m)

        return y_pred
```

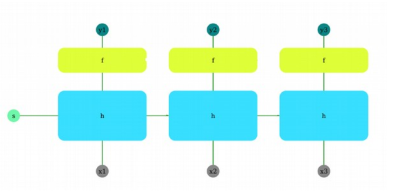
Tensorflow RNN implementation

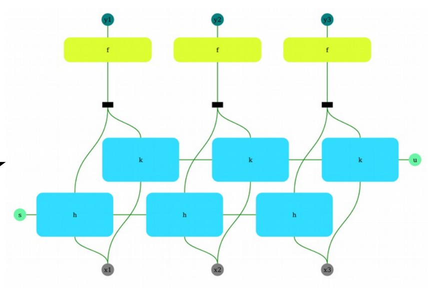
- Keras built-in RNN networks (standard RNN configurations)
  - `tf.keras.layers.SimpleRNN`
  - `tf.keras.layers.LSTM`
  - `tf.keras.layers.GRU`
- RNN Cell wrapped in tf.keras.layer.RNN
  - `tf.keras.layers.SimpleRNNCell`
  - `tf.keras.layers.LSTMCell`
  - `tf.keras.layers.GRUCell`
  - Custom cell extending `tf.keras.layers.AbstractRNNCell`

Keras built-in RNNs (I)

```python
import tensorflow as tf
import numpy as np

inputs = np.random.random([32, 10, 8]).astype(np.float32)

# RNN with only last output
simple_rnn1 = tf.keras.layers.SimpleRNN(4)

# The output has shape ‘[32, 4]‘.
output = simple_rnn1(inputs)

# RNN with output at all time steps
simple_rnn2 = tf.keras.layers.SimpleRNN(4, return_sequences=True,
                                         return_state=True)

# whole_sequence_output has shape ‘[32, 10, 4]‘.
# final_state has shape ‘[32, 4]‘.
whole_sequence_output, final_state = simple_rnn2(inputs)

# Bidirectional RNN based on simple_rnn2
simple_bidirectional = tf.keras.layers.Bidirectional(simple_rnn2)
```



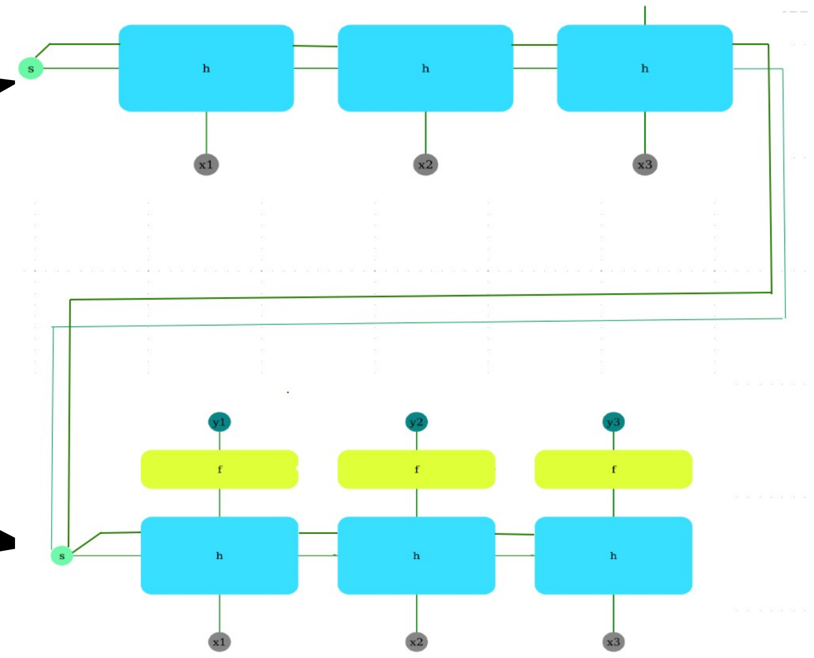




Keras built-in RNNs (II)

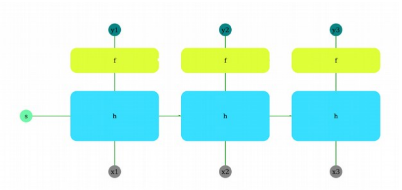
- Encoder-decoder using LSTMs

```
import tensorflow as tf
import numpy as np

# Define encoder and decoder values for running the model
enc_input_val = np.random.random([32, 10, 8]).astype(np.float32)
dec_input_val = np.random.random([32, 6, 7]).astype(np.float32)

#Define the encoder
encoder_input = tf.keras.layers.Input(shape=(10,8))
encoder=tf.keras.layers.LSTM(64, return_state=True, name="encoder")

# Return states in addition to output
output, state_s, state_c = encoder(encoder_input)
encoder_state = [state_s, state_c]

# Evaluate encoder
output_val, state_s_val, state_c_val = encoder(encoder_input_val)
print(output_val.shape, state_s_val.shape, state_c_val.shape)

#define the decoder
decoder_input = tf.keras.layers.Input(shape=(6,7))
decoder = tf.keras.layers.LSTM(64, return_sequences=True,
                               name="decoder")

# Pass the 2 states to a new LSTM layer, as initial state
dec_output = decoder(decoder_input, initial_state=encoder_state)
output = tf.keras.layers.Dense(10)(dec_output)

#define the full model (encoder-decoder)
model = tf.keras.Model([encoder_input, decoder_input], output)
#model.summary()

# Predict using the model
output_val= model.predict([encoder_input_val, decoder_input_val])
print(output_val.shape)
```

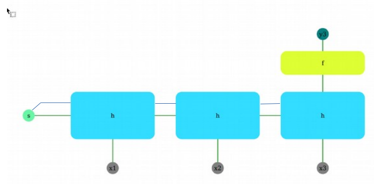


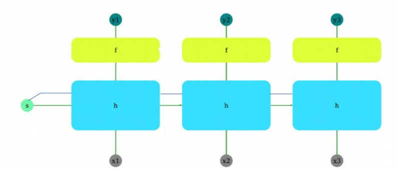
Cells wrapped in keras.RNN

```python
inputs = np.random.random([32, 10, 8]).astype(np.float32)

# RNN wrapping a simple RNN cell
rnn = tf.keras.layers.RNN(
    tf.keras.layers.SimpleRNNCell(4),
    return_sequences=True,
    return_state=True)

# whole_sequence_output has shape ‘[32, 10, 4]‘.
# final_state has shape ‘[32, 4]‘.
whole_sequence_output, final_state = rnn(inputs)
```



```python
inputs = tf.random.normal([32, 10, 8])
# RNN wrapping an LSTM cell, single output
rnn_single = tf.keras.layers.RNN(tf.keras.layers.LSTMCell(4))
output = rnn_single(inputs)
print(output.shape)

# RNN wrapping an LSTM cell, sequential  output
rnn_seq = tf.keras.layers.RNN(
   tf.keras.layers.LSTMCell(4),
   return_sequences=True,
   return_state=True)
whole_seq_output, final_state_s, final_state_c = rnn_seq(inputs)
```





Custom Cells wrapped in keras.RNN

- Class extends `tf.keras.layers.AbstractCell`
- Call method has the signature `(output, next_state) = call(input, state)`

```python

class MyVanillaRNNCell(tf.keras.layers.AbstractRNNCell):
  def __init__(self, units):
     super(VanillaRNNCell, self).__init__()
     self.units = units
     self.dense = layers.Dense(units)

@property
  def state_size(self):
      return self.units


 def call(self, x, state):
   c = tf.concat([x, state], axis=-1)
   h = self.dense(c)
   output = activations.tanh(h)
   return output, output


my_own_rnn=tf.keras.layers.RNN(MyVanillaRNNCell (4))
```