Introduction to Tensorflow 2.0

Tensor: rank, shape

- The central unit of data in Tensorflow is called tensor.
- A tensor's rank is its number of dimensions
- A tensor's shape is a tuple of integers specifying the array's length along each dimension.

```
3. # rank 0 tensor; a scalar with shape []
[1., 2., 3.] # a rank 1 tensor; a vector with shape [3]
[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]
[[[1., 2., 3.]], [[2., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]
```

Initialization of tensors

- tf.zeros()
- tf.ones()
- tf.fill((n,m,…), value=..)
- tf.random.normal((n,m,…), mean=.., stddev=…)
- tf.random.uniform((n,m,…),minval=…., maxval=…)
- tf.constant()
  - Can only be used for tensors whose values cannot be modified.
- tf.eye(n)
- tf.diag([…,…,…,…])
  - tf.range(…,…,…) can be used to build an array.

Shape & types of tensors

- tdata_b = tf.reshape(tdata_a, (n, m, …))
- tdata.get_shape()

- tdata_b = tf.squeeze(tdata_a)
- tdata_b = tf.expand_dims(tdata_a, n)

- Types of tensors
  - tf.float32, tf.float64, tf.int32, tf.int8

```
tvar = tf.random.uniform((2,3,1))
print("Before squeeze: ", tvar)
tf.squeeze(tvar)
print("After squeeze: ", tvar)
```

```
Before squeeze:  tf.Tensor(
[[[0.44425166]
  [0.5232893 ]
  [0.14688873]]

 [[0.35450578]
  [0.8383919 ]
  [0.6759068 ]]], shape=(2, 3, 1), dtype=float32)
After squeeze:  tf.Tensor(
[[[0.44425166]
  [0.5232893 ]
  [0.14688873]]

 [[0.35450578]
  [0.8383919 ]
  [0.6759068 ]]], shape=(2, 3, 1), dtype=float32)
```

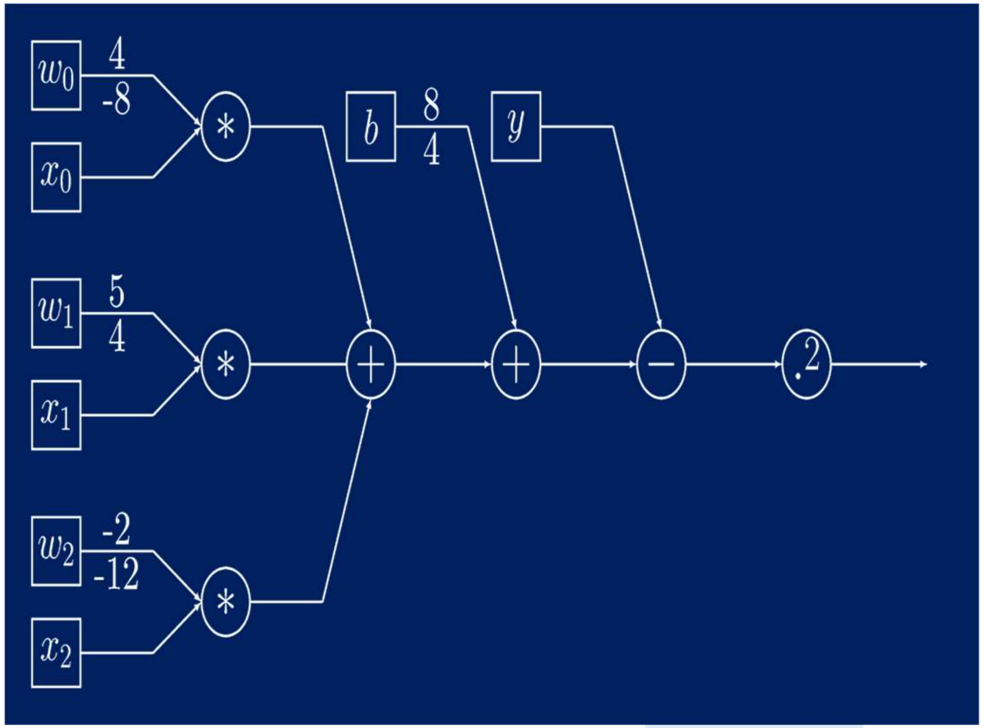
Tensorflow variable

- The ***Variable()*** constructor requires an initial value for the variable, which can be a Tensor of any type and shape.
  - tval = tf.Variable (tf.ones((2,2)))
- The value can be changed using one of the assign methods.
  - ***tval.assign(....)*** can be used to change the variable values.
  - ***tval.assign_add(....)*** and ***tval.assign_sub(....)*** can be used to add or subtract the some values to the tensor variables.

```
tf.Variable(
    initial_value=None, trainable=None, validate_shape=True, caching_device=None,
    name=None, variable_def=None, dtype=None, import_scope=None, constraint=None,
    synchronization=tf.VariableSynchronization.AUTO,
    aggregation=tf.compat.v1.VariableAggregation.NONE, shape=None
)
```

Tensorflow operators

- tf.add(a, b)
- tf.substract(a, b)
- tf.multiply(a, b)
- tf.div(a, b)
- tf.pow(a, b)
- tf.exp(a)
- tf.sqrt(a)
- tf.log(a)



Features

- TensorFlow 2.0 offers multiple levels of abstraction.
  - Symbolic (Declarative) style：build a model by manipulating a graph of layers
  - Imperative style：build a model by extending a class

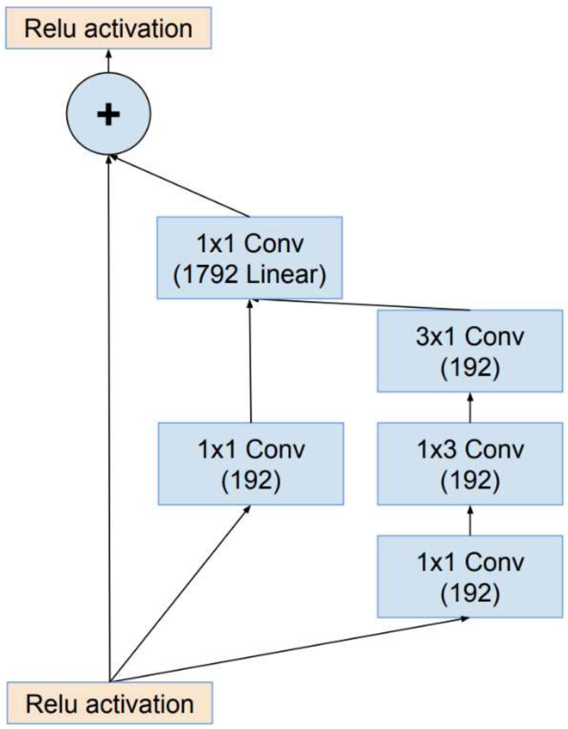
- Eager execution is turned on by default.
  - Declarative has been turned into imperative.

Symbolic (Declarative) APIs

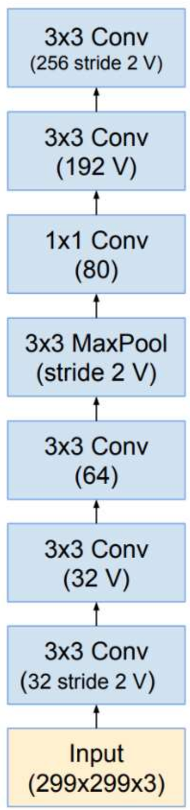
- A neural network is regarded as a "graph of layers"
  - DAG or simply stack
- Follow the keras method

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Your symbolically defined model



stack



Benefits and Limitations

- Model is a graph-like data structure, which can be inspected by
  - *keras.utils.plot_model* or *model.summary()*
- Symbolic models provide a consistent API such that it's simple to reuse and share

```
from tensorflow.keras.applications.vgg19 import VGG19
base = VGG19(weights=' imagenet' )
model = Model(inputs=base.input,
outputs=base_model.get_layer( 'block4_pool' ).output)
image = load( 'elephant.png' )
block4_pool_features = model.predict(image)
```

- Easy to copy and clone
  - *Model.get_config()*, *model.to_json()*, *model.save()*, *clone_model(model)*
- Can only be used to build models that are directed acyclic graphs of layers.

Keras

reference:
https://keras.io/
&
https://www.tensorflow.org/guide/keras/overview

Keras

- Easy to use API to build neural networks.
- The Keras API integrates seamlessly with your TensorFlow workflows
- Keras has stronger adoption in both the industry and the research community than any other deep learning framework except TensorFlow itself.
- The Keras API is the official frontend of TensorFlow, via the tf.keras module.
- Keras has built-in support for multi-GPU data parallelism

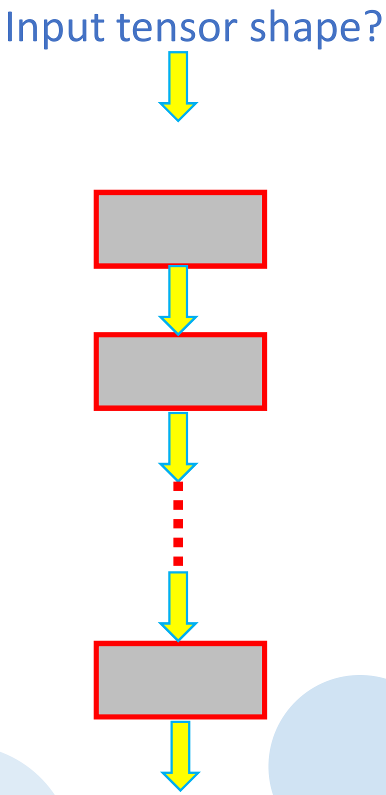
Multilayer Perceptron (MLP) for multi-class softmax classification:

```
Import tensorflow.keras as keras        # use tensorflow 2.0
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',optimizer=sgd,metrics=['accuracy'])
model.fit(x_train, y_train,  epochs=20, batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

Keras sequential mode

- The Sequential model is a linear stack of layers.

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

model = Sequential([
        Dense(32, input_shape=(784,)),
        Activation('relu'),
        Dense(10),
        Activation('softmax')
])
```



- Can add layers via the *.add()* method.

```
model.add(Dense(32))
model.add(Activation('relu'))
```

Keras sequential mode

- The shape of the input layer of a sequential model can be specified by explicitly using ***model.add(layers.Input(shape=(x,x), batch_size=xx))***
  - The default of batch_size is **NONE**, which means any positive integer can be expected.
- It can also be declared inside the first hidden layer of the model by specifying
  - The ***input_shape*** argument which has to be a tuple such as ***input_shape=(784,)*** , ***input_shape=(10,5)***
    - In python, a=(5) is not a tuple.
  - For rank-1 input tensors (excluding the batch size), it can also be specified by ***input_dim=n***.
    - The complete shape will be reported as ***(NONE, n)***
  - If the batch size is fixed, the shape of input layer can also be specified by using ***batch_input_shape=(30,50,50,3)*** or an additional argument ***batch_size=30***.
- The shape of input tensors of the following layers do not have to be specified since the framework can do *automatic shape inference* about its *shape*.

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
```

```
model = Sequential()
model.add(Dense(32, input_dim=784))
```

Keras input object: *tf.keras.Input*

- Input() is used to instantiate a Keras tensor.

```
tf.keras.Input(
    shape=None, batch_size=None, name=None, dtype=None, sparse=False, tensor=None,
    ragged=False, **kwargs
)
```

- A Keras model can be built just by knowing the inputs and outputs of the model.
  - For instance, if a, b and c are Keras tensors, we can write:

    model = Model(input=[a, b], output=c)

- Input produces a symbolic tensor (i.e. a placeholder). This symbolic tensor can be used with other TensorFlow ops, as such:

```
x = Input(shape=(32,))
y = tf.square(x)
```

Keras layer: *tf.keras.layers.Dense*

- Dense implements the operation: *output = activation(dot(input, kernel) + bias)*
  - kernel variable is a weights matrix created by the layer.
  - Densely (Fully) -connected NN layer.
- Example:
  - Dense(32, input_shape=(16,)) # first layer
  - Dense(32)
- The activation parameter can be set as *activation='tanh'*

```
tf.keras.layers.Dense(
    units, 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
)
```

An example of Dense layer

- Output shape?
- # of Parameter?

```python
from tensorflow.keras.layers import Dense
model = tf.keras.models.Sequential()

model.add(Dense(3, batch_size=6, input_shape=(2,4)))
```

```python
model.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (6, 2, 3)                 15
=================================================================
Total params: 15
Trainable params: 15
Non-trainable params: 0
_________________________________________________________________
```

Keras layer: *tf.keras.layers.Conv2D*

- This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If use_bias is True, a bias vector is created and added to the outputs. Finally, if activation is not None, it is applied to the outputs as well.
- data_format can decide where the channel is.
  - data_format='channels_first‘ (default: ‘channels_last’)
- kenel_size can be a scalar or a 2D tuple.
- Padding can be 'valid' or 'same'
- The activation parameter can be set as *activation='tanh'*

```
tf.keras.layers.Conv2D(
    filters, kernel_size, strides=(1, 1), padding='valid', 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
)
```

An example of Conv2D layer

- What is padding?
- Output shape?
- Strides?
- Total params?

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

model = tf.keras.models.Sequential()

model.add(Conv2D(
    batch_input_shape=(None, 6, 10, 12),
    filters=4,
    kernel_size=(5,3),
    name = 'Hello',
    strides=1,
#    padding='same',      # Padding method
#    data_format='channels_first',
))
```

```
model.summary()
```

```
Model: "sequential_25"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Hello (Conv2D)               (None, 2, 8, 4)           724
=================================================================
Total params: 724
Trainable params: 724
Non-trainable params: 0
```

Keras layer: *tf.keras.layers.Flatten*

- Flattens the input. Does not affect the batch size.

```
tf.keras.layers.Flatten(
    data_format=None, **kwargs
)
```

- Example:

```
model = tf.keras.models.Sequential()
model.add(Conv2D(64, 3,
                 padding='same',
                 input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

Keras pooling layer: *MaxPool2D, AveragePooling2D*

- The setting of each parameter in this layer is pretty similar to Conv2D.
- Default data_format is 'channels_last'

```
tf.keras.layers.MaxPool2D(
    pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs
)
```

```
tf.keras.layers.AveragePooling2D(
    pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs
)
```

Keras layer: *Reshape*

- Reshapes an output to a certain shape.

```
tf.keras.layers.Reshape(
    target_shape, **kwargs
)
```

- Example:

```
# as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# now: model.output_shape == (None, 3, 4)
# note: `None` is the batch dimension

# as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
# now: model.output_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, None, 2, 2)
```

CNN model by Keras

```
model = Sequential()

# Conv layer 1 output shape (32, 28, 28)
model.add(Convolution2D(
    batch_input_shape=(None, 1, 28, 28),
    filters=32,
    kernel_size=5,
    strides=1,
    padding='same',     # Padding method
    data_format='channels_first',
))
model.add(Activation('relu'))

# Pooling layer 1 (max pooling) output
shape (32, 14, 14)
model.add(MaxPooling2D(
    pool_size=2,
    strides=2,
    padding='same',    # Padding method
    data_format='channels_first',
))
```

```
# Conv layer 2 output shape (64, 14, 14)
model.add(Convolution2D(64, 5, strides=1,
padding='same', data_format='channels_first'))
model.add(Activation('relu'))

# Pooling layer 2 (max pooling) output shape (64, 7, 7)
model.add(MaxPooling2D(2, 2, 'same',
data_format='channels_first'))

# Fully connected layer 1 input shape (64 * 7 * 7) =
(3136), output shape (1024)
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation('relu'))

# Fully connected layer 2 to shape (10) for 10 classes
model.add(Dense(10))
model.add(Activation('softmax'))
```

Keras compilation

```
compile(optimizer, loss=None, metrics=None,
loss_weights=None, sample_weight_mode=None,
weighted_metrics=None, target_tensors=None)
```

- Before training a model, the learning process has to be configured via the compile method.
  1) An optimizer
  2) A loss function
  3) A list of metrics
- For example:

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- Note that if you're satisfied with the default settings, in many cases the optimizer, loss, and metrics can be specified via *string identifiers* as a shortcut.

```
# For a mean squared error regression problem
model.compile(optimizer=keras.optimizer.RMSprop(learning_rate=1e-3),loss='mse')
```

```
# For custom metrics
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

Optimizer

- Many built-in optimizers, losses, and metrics are available
- Optimizers:
  - SGD() (with or without momentum)
  - RMSprop()
  - Adam()
- Losses:
  - MeanSquaredError()
  - KLDivergence()
  - CosineSimilarity()
- Metrics:
  - AUC()
  - Precision()
  - Recall()

Keras training

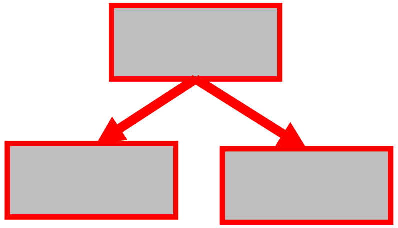
- Keras models are trained on Numpy arrays of input data and labels.
- Call the *fit()* method to train a model. typically use the fit function.

```
# Generate test data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels,
num_classes=10)

# Train the model, iterating on the data in batches of
32 samples
model.fit(data, one_hot_labels, epochs=10,
batch_size=32)
```

```
fit(x=None, y=None, batch_size=None, epochs=1,
verbose=1, callbacks=None, validation_split=0.0,
validation_data=None, shuffle=True,
class_weight=None, sample_weight=None,
initial_epoch=0, steps_per_epoch=None,
validation_steps=None, validation_freq=1,
max_queue_size=10, workers=1,
use_multiprocessing=False))
```

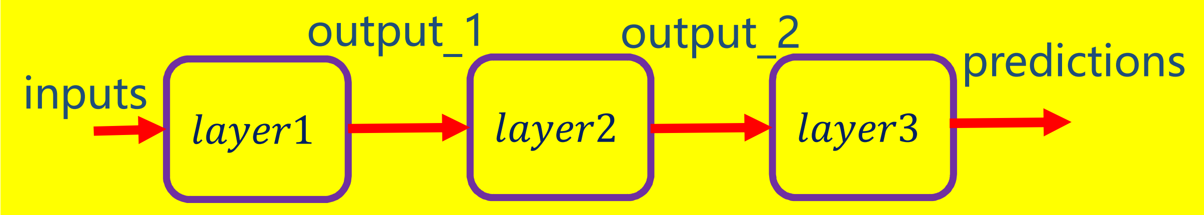
```
train_on_batch(x, y, sample_weight=None,
class_weight=None, reset_metrics=True)
```

Model class built with the functional API



◇ In the functional API, given some input tensor(s) and output tensor(s), you can instantiate a Model via:

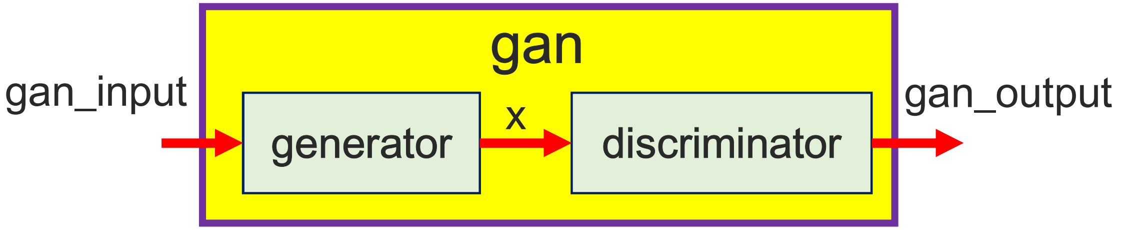
```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))
```




```
# a layer instance is callable on a tensor, and returns a tensor
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)

# This creates a model that includes the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

```
def create_gan(discriminator, generator):
    discriminator.trainable=False
    gan_input = Input(shape=(100,))
    x = generator(gan_input)
    gan_output= discriminator(x)
    gan= Model(inputs=gan_input, outputs=gan_output)
    gan.compile(loss='binary_crossentropy', optimizer='adam')
    return gan
```

A keras model can accept an input tensor argument.



- Three keras models (**generator, discriminator, gan**) have been created.
  - The models of **generator** and **discriminator** have been concatenated to form the model of **gan**.

Dataset provided in Keras

- from keras.datasets import mnist

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# data pre-processing
X_train = X_train.reshape(X_train.shape[0], -1) / 255.   # normalize
X_test = X_test.reshape(X_test.shape[0], -1) / 255.      # normalize
y_train = np_utils.to_categorical(y_train, num_classes=10)
y_test = np_utils.to_categorical(y_test, num_classes=10)
```

- **from** keras.datasets **import** cifar10
- **from** keras.datasets **import** fashion_mnist

The Keras functional API in TensorFlow2.0

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(784,), name='img')
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs, name='mnist_model')
# keras.utils.plot_model(model, 'my_first_model.png')
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=keras.optimizers.RMSprop(),
              metrics=['accuracy'])
history = model.fit(x_train, y_train,
                    batch_size=64,
                    epochs=5,
                    validation_split=0.2)
test_scores = model.evaluate(x_test, y_test, verbose=2)
```

Other useful keras function

- *model.summary()* summarizes the info of the model.
- *model.layers[0].get_weights()* can be used to fetch the training weights.
- *model.layers[0].activation* can show the activation function used.
- *model.trainable=**True*** can be used to decide whether to update the model weights.
- *model.train_on_batch* can train on only one batch of data.

Writing Custom Layers, Models, and Training

Imperative (or Model Subclassing) APIs

- Building models is like Object-Oriented Python development
  - Extend a Model class defined by the framework
  - Instantiate layers
  - Write the forward pass of your model imperatively (the backward pass is generated automatically) in the **call()** method.

```
class CNN_Encoder(tf.keras.Model):
  def __init__(self, embedding_dim):
    super(CNN_Encoder, self).__init__()
    self.fc = tf.keras.layers.Dense(embedding_dim)

  def call(self, x):
    x = self.fc(x)
    x = tf.nn.relu(x)
    return x
```

```python
import tensorflow as tf

class MyModel(tf.keras.Model):

  def __init__(self):
    super(MyModel, self).__init__()
    self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
    self.dense2 = tf.keras.layers.Dense(5, activation=tf.nn.softmax)

  def call(self, inputs):
    x = self.dense1(inputs)
    return self.dense2(x)

inputs = keras.Input(shape=(2,))

model = MyModel()
model(inputs)
```

```
<tf.Tensor 'my_model_33/Identity:0' sh
```

```python
model.summary()
```

```
Model: "my_model_33"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_98 (Dense)             (None, 4)                 12
_________________________________________________________________
dense_99 (Dense)             (None, 5)                 25
=================================================================
Total params: 37
Trainable params: 37
Non-trainable params: 0
```

The *call()* method

- __call__() method in the class definition can make a object callable.
- call() method in custom modules or layers are invoked during the execution of __call__() method.

Benefits and Limitations

- Your forward pass is written imperatively, making it easy to swap out parts implemented by the library (say, a layer, activation, or loss function) with your own implementation.
- Your model is no longer a transparent data structure, it is an opaque piece of bytecode. When using this style, you're trading usability and reusability to gain flexibility
- Hard to reuse
- Hard to inspect:
  - model.save(), model.get_config(), and clone_model do not work for subclassed models. Likewise, model.summary() only gives you a list of layers (and doesn't provide information on how they're connected, since that's not accessible)

Custom layers and models with keras

- The Layer class will define inner computation blocks, while the Model class will define the outer model -- the object you will train.
- The Model class has the same API as Layer, with the following differences:
  - It exposes built-in training, evaluation, and prediction loops (model.fit(), model.evaluate(), model.predict()).
  - It exposes the list of its inner layers, via the model.layers property.
  - It exposes saving and serialization APIs.

Writing custom layers

- A layer encapsulates both a state (the layer's "weights") and a transformation from inputs to outputs (a "call", the layer's forward pass).
- In many cases, you may not know in advance the size of your inputs, and you would like to lazily create weights when that value becomes known, some time after instantiating the layer.
  - *build()* function will be called once in __call__(), when the shape and type of input is known.
  - Should call *add_weight()* to create weights .

Writing custom layers

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

  def build(self, input_shape):
    self.kernel = self.add_weight("kernel",
                                  shape=[int(input_shape[-1]),
                                         self.num_outputs])


  def call(self, input):
    return tf.matmul(input, self.kernel)

layer = MyDenseLayer(4)
print(layer(tf.ones([1,2])))
print(layer.trainable_variables)
```

```
tf.Tensor([[-0.7848737  -1.3362324  -1.4629352   0.21280456]], shape=(1, 4), dtype=float32)
[<tf.Variable 'my_dense_layer_15/kernel:0' shape=(2, 4) dtype=float32, numpy=
array([[-0.3946271 , -0.5729692 , -0.8009021 ,  0.60638285],
       [-0.39024663, -0.7632632 , -0.6620331 , -0.3935783 ]],
      dtype=float32)>]
```

- *build()* function will be called once in __call__(), when the shape and type of input is known.
- Should call to add_weight().

call(): Called in __call__ after making sure build() has been called once.

Training loops

- Models defined in either the Sequential, Functional, or Subclassing style can be trained in two ways:
  - Built-in training routine and loss function
    - model.compile, model.fit
    - Custom training loop, loss function

```
def train(dataset, epochs):
  for input_image, target in dataset:
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      gen_output = generator(input_image, training=True)

      disc_real_output = discriminator([input_image, target], training=True)
      disc_gen_output = discriminator([input_image, gen_output], training=True)

      gen_loss = generator_loss(disc_gen_output, gen_output, target)
      disc_loss = discriminator_loss(disc_real_output, disc_gen_output)

    generator_gradients = gen_tape.gradient(gen_loss,
                                            generator.trainable_variables)

    discriminator_gradients = disc_tape.gradient(disc_loss,
                                                 discriminator.trainable_variables)
```

Steps for building & training models

- Build a model object
  - Declare a model object
  - Declare model architectures
  - Write the forward pass

```
class Model(object):
 def __init__(self):

  self.W = tf.Variable(5.0)
  self.b = tf.Variable(0.0)

 def __call__(self, x):
  return self.W * x + self.b
```

- Define loss function

```
def loss(predicted_y, target_y):
 return tf.reduce_mean(tf.square(predicted_y - target_y))
```

- Write training loop
  - Call model function
  - Compare against with the target result
  - Compute the gradient of loss with respect to the trainable weight variable.
  - Apply the gradients to update weight variables.
    - *optimizer.apply_gradient()*
    - *assign_sub()*

```
def train(model, inputs, outputs, learning_rate):
 with tf.GradientTape() as t:
  current_loss = loss(model(inputs), outputs)
 dW, db = t.gradient(current_loss, [model.W, model.b])
 model.W.assign_sub(learning_rate * dW)
 model.b.assign_sub(learning_rate * db)
```

b=b-learning_rate*db

Tf.keras.optimizers.Optimizer()

- Possible *Optimizer*(): SGD, Adam
- Relevant APIs:
  - minimize()
  - apply_gradients()
  - ……

```
# Create an optimizer.
opt = tf.keras.optimizers.SGD(learning_rate=0.1)

# Compute the gradients for a list of variables.
with tf.GradientTape() as tape:
  loss = <call_loss_function>
vars = <list_of_variables>
grads = tape.gradient(loss, vars)

# Process the gradients, for example cap them, etc.
# capped_grads = [MyCapper(g) for g in grads]
processed_grads = [process_gradient(g) for g in grads]

# Ask the optimizer to apply the processed gradients.
opt.apply_gradients(zip(processed_grads, var_list))
```

Gradient calculation

```
@tf.function
def add(a, b):
  return a + b

v = tf.Variable(1.0)
with tf.GradientTape() as tape:
  result = add(v, 1.0)
tape.gradient(result, v)
```

◇Calculation of gradient

```
def add(a, b):
    return 2*a*a + 3*b

v1 = tf.Variable(1.2)
v2 = tf.Variable(2.0)

with tf.GradientTape() as tape:
    result = add(v1, v2)

g1=tape.gradient(result, [v1, v2])

print(g1)
```

```
[<tf.Tensor: id=215, shape=(), dtype=float32, numpy=4.8>, <tf.Tensor: id=212, s
hape=(), dtype=float32, numpy=3.0>]
```

Use of GraidentTape()

https://www.tensorflow.org/api_docs/python/tf/GradientTape

- By default ***GradientTape*** will automatically watch any *trainable variables* that are accessed inside the context.
  - Don't need to call watch(var)

```
variable_a=tf.Variable(4.0)
variable_b=tf.Variable(5.0)
with tf.GradientTape(watch_accessed_variables=False) as tape:
  tape.watch(variable_a)
  y = variable_a ** 2  # Gradients will be available for `variable_a`.
  z = variable_b ** 3  # No gradients will be available since `variable_b` is
                       # not being watched.
```

  - High–order derivatives

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
  g.watch(x)           # if this statement is removed,d2y_dx2 will equal NONE.
  with tf.GradientTape() as gg:
    gg.watch(x)
    y = x * x
  dy_dx = gg.gradient(y, x)     # Will compute to 6.0
d2y_dx2 = g.gradient(dy_dx, x)  # Will compute to 2.0
```

Persistent (持久) tape

- When multiple gradient calls to the same variable, the persistent of GradientTape has to be set to **True**.

```
x = tf.Variable(3.0)
with tf.GradientTape() as g1, tf.GradientTape() as g2:
    y = x * x
    y2 = y * y
dy = g1.gradient(y, x)
d2y = g2.gradient(y2, x)
```

```
x = tf.Variable(3.0)
with tf.GradientTape(persistent=True) as g:
    y = x * x
    y2 = y * y
dy = g.gradient(y, x)
d2y = g.gradient(y2, x)
del g
```

- Trainable variables (created by [tf.Variable]() where `trainable=True` is default) are automatically watched.
    - x = tf.Variable(3.0, trainable=False) won't be watched.
- Tensors can be manually watched by invoking the `watch` method on this context manager.

```
x1 = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x1)
    x2 = x1 * x1
    y = 2*x2 + 1
g.gradient(y, [x1, x2])
```

```
[<tf.Tensor: id=216363, shape=(), dtype=float32, numpy=12.0>,
 <tf.Tensor: id=216360, shape=(), dtype=float32, numpy=2.0>]
```

- By default, the resources held by a GradientTape are released as soon as GradientTape.gradient() method is called.
- To compute multiple gradients over the same computation, create a persistent gradient tape. This allows multiple calls to the gradient() method as resources are released when the tape object is garbage collected.

Linear Regression

https://www.tensorflow.org/tutorials/customization/custom_training

◈Model definition

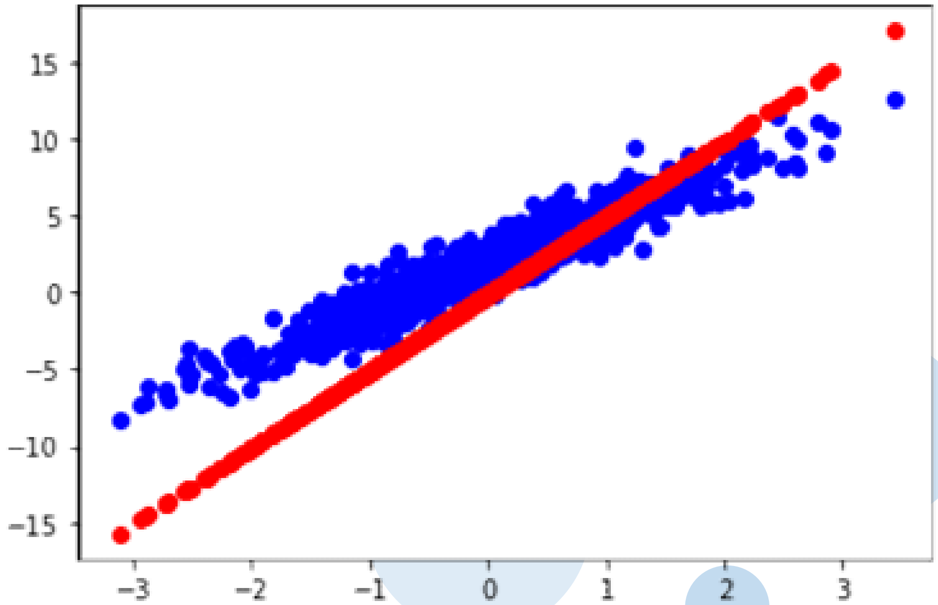
```
class Model(object):
  def __init__(self):
    # Initialize the weights to `5.0` and the bias to `0.0`
    # In practice, these should be initialized to random values (for
example, with `tf.random.normal`)
    self.W = tf.Variable(5.0)
    self.b = tf.Variable(0.0)

  def __call__(self, x):
    return self.W * x + self.b

model = Model()

assert model(3.0).numpy() == 15.0
```

◈Loss and train method definition

```
def loss(target_y, predicted_y):
  return tf.reduce_mean(tf.square(target_y - predicted_y))

def train(model, inputs, outputs, learning_rate):
  with tf.GradientTape() as t:
    current_loss = loss(outputs, model(inputs))
  dW, db = t.gradient(current_loss, [model.W, model.b])
  model.W.assign_sub(learning_rate * dW)
  model.b.assign_sub(learning_rate * db)
```

```python
TRUE_W = 3.0
TRUE_b = 2.0
NUM_EXAMPLES = 1000

inputs  = tf.random.normal(shape=[NUM_EXAMPLES])
noise   = tf.random.normal(shape=[NUM_EXAMPLES])
outputs = inputs * TRUE_W + TRUE_b + noise

model = Model()

# Collect the history of W-values and b-values to plot later
Ws, bs = [], []
epochs = range(10)
for epoch in epochs:
  Ws.append(model.W.numpy())
  bs.append(model.b.numpy())
  current_loss = loss(outputs, model(inputs))

  train(model, inputs, outputs, learning_rate=0.1)
  print('Epoch %2d: W=%1.2f b=%1.2f, loss=%2.5f' %
        (epoch, Ws[-1], bs[-1], current_loss))

# Let's plot it all
plt.plot(epochs, Ws, 'r',epochs, bs, 'b')
plt.plot([TRUE_W] * len(epochs), 'r--',
         [TRUE_b] * len(epochs), 'b--')
plt.legend(['W', 'b', 'True W', 'True b'])
plt.show()
```

```
Current loss: 9.255826
Epoch 0: W=5.00 b=0.00, loss=9.25583
Epoch 1: W=4.59 b=0.42, loss=6.14175
Epoch 2: W=4.26 b=0.75, loss=4.20099
Epoch 3: W=4.00 b=1.01, loss=2.99145
Epoch 4: W=3.80 b=1.22, loss=2.23762
Epoch 5: W=3.64 b=1.38, loss=1.76780
Epoch 6: W=3.51 b=1.51, loss=1.47498
Epoch 7: W=3.41 b=1.61, loss=1.29248
Epoch 8: W=3.33 b=1.69, loss=1.17873
Epoch 9: W=3.26 b=1.75, loss=1.10783
```



Pytorch

https://towardsdatascience.com/minimal-pytorch-subset-for-deep-learning-for-data-scientists-8ccbd1ccba6b

Creation of tensors

- torch.Tensor([[1,2,3],[4,5,6]])
- torch.from_numpy(a)
  - a=np.array ([[1,2,3],[4,5,6]])
  - t=t.numpy() # convert tensor to numpy
- torch.ones(3,5)
- torch.zeros(3,5)
- torch.randn(3,5)
- torch.randint(lvalue,hvalue,size)

Tensor operations

- A=torch.Tensor(…), B=torch.Tensor(…)
- t=A.mm(B) #multiplication
- t=t.t() #transpose
- t=t**2 #square
- t.size() # shape of tensors

Build Models Using *nn.Sequential()*

- Style 1: Add a layer sequentially.
  - net1 = nn.Sequential()
  - net1.add_module('conv', nn.Conv2d(3, 3, 3))
  - net1.add_module('batchnorm', nn.BatchNorm2d(3))
  - net1.add_module('activation_layer', nn.ReLU())
- Style 2: No layer names.
  - *net2 = nn.Sequential(nn.Conv2d(3, 3, 3),nn.BatchNorm2d(3), nn.ReLU())*
- Style 3: Specify layer names.
  - *from collections import OrderedDict*
  - *net3= nn.Sequential(OrderedDict([*

```
('conv', nn.Conv2d(3, 3, 3)),
('batchnorm', nn.BatchNorm2d(3)),
('activation_layer', nn.ReLU())
]))
```

Build Models Using *nn.ModuleList()*

- Can use nn.ModuleList() to build a sub-module.
- Still need to define forward()

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10) for i in
range(10)])
    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, l in enumerate(self.linears):
            x = self.linears[i // 2](x) + l(x)
        return x
```

Class construction from nn.Module

- Inherit ***nn.Module***
- Must include *__init__(self, parameter……)*
  - Can declare the layers from the library.
  - EX: nn.Linear(5,10)
    - Define a linear transformation layer
- Should define *forward(self,x)* to build the computation graph.
- A custom model can be created by *model=myNeuralNet()*

```
class myNeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        # Define all Layers Here
        self.lin1 = nn.Linear(784, 30)
        self.lin2 = nn.Linear(30, 10)
    def forward(self, x):
        # Connect the layer Outputs here to define the forward pass
        x = self.lin1(x)
        x = self.lin2(x)
    return x
```

Common used default layers

- nn.Linear
- nn.Conv2d
- nn.MaxPool2d
- nn.ReLU
- nn.BatchNorm2d
- nn.Dropout
- nn.Embedding
- nn.GRU/nn.LSTM
- nn.Softmax
- nn.LogSoftmax
- nn.MultiheadAttention
- nn.TransformerEncoder
- nn.TransformerDecoder

nn.Conv2d

CONV2D

CLASS `torch.nn.Conv2d(in_channels: int, out_channels: int, kernel_size: Union[T, Tuple[T, T]], stride: Union[T, Tuple[T, T]] = 1, padding: Union[T, Tuple[T, T]] = 0, dilation: Union[T, Tuple[T, T]] = 1, groups: int = 1, bias: bool = True, padding_mode: str = 'zeros')` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

Creation of custom layers

- Use *nn.Parameter()* to create the tensor variables used in the module.

```
class myCustomLinearLayer(nn.Module):
    def __init__(self,in_size,out_size):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(in_size, out_size))
        self.bias = nn.Parameter(torch.zeros(out_size))
    def forward(self, x):
        return x.mm(self.weights) + self.bias
```

Training a Neural Network

- Fetch training data
- Do a forward pass using *model(t_data)*
- Calculate the loss
  - Use some default loss criterion
    - Nn.CrossEntropyLoss , nn.NLLLoss , nn.KLDivLoss nn.MSELoss.
  - Custom defined loss function.
- Use *loss.backward()* call to calculate the gradients.
- Call *optimizer.step()* to update the model weights.
- Can call *model.eval()* to check the performance.

```
num_epochs = 5
    for epoch in range(num_epochs):
    # Set model to train mode
    model.train()
    for x_batch,y_batch in train_dataloader:
        # Clear gradients
        optimizer.zero_grad()
        # Forward pass - Predicted outputs
        pred = model(x_batch)
        # Find Loss and backpropagation of gradients
        loss = loss_criterion(pred, y_batch)
        loss.backward()
        # Update the parameters
        optimizer.step()
    model.eval()
    for x_batch,y_batch in valid_dataloader:
        pred = model(x_batch)
        val_loss = loss_criterion(pred, y_batch)
```

Gradient computation in PyTorch

- torch.autograd is PyTorch’s automatic differentiation engine that helps us to compute gradients.
- A tensor x has to be created with *requires_grad=True*. This signals to autograd that every operation on it should be tracked. When we call *.backward()* on z, autograd calculates these gradients and stores them in the tensor’s *.grad* attribute. Hence, we can see the gradients in x.grad.
  - x=torch.tensor([[1.1,1],[1,1]], requires_grad=True)
  - y=torch.sum(x)
  - z=y*y
  - z.backward()
  - x.grad

Custom Loss Function

- Define a function to compute loss
  - Use the custom loss as before.

```
def customMseLoss(output,target):
    loss = torch.mean((output - target)**2)
    return loss
```

```
output = model(x)
loss = customMseLoss(output, target)
loss.backward()
```

- Define a loss object

```
class CustomNLLLoss(nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self, x, y):
        # x should be output from LogSoftmax Layer
        log_prob = -1.0 * x
        loss = log_prob.gather(1, y.unsqueeze(1))
        loss = loss.mean()
        return loss
criterion = CustomNLLLoss()
loss = criterion(preds,y)
```

Optimizers

- An optimizer will be used to apply the gradients computed from the *loss.backward()* to change the weights in the network.
- Common default optimizers:
  - torch.optim.Adadelta
  - torch.optim.Adagrad
  - torch.optim.RMSprop
  - torch.optim.Adam.
- Instantiation of optimizer:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, betas=(0.9, 0.999))
```

- Use *optimizer.zero_grad()* and *optimizer.step()* while training the model.

Using GPU/Multiple GPUs

- Call the function *torch.cuda.is_available()* to check if GPU is available to use.
- To use a GPU, put the model to GPU using *model.to('cuda')*
- To use multiple GPUs, use *nn.DataParallel(model)*

```
num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    for x_batch,y_batch in train_dataloader:
        if train_on_gpu:
            x_batch,y_batch = x_batch.cuda(), y_batch.cuda()
        optimizer.zero_grad()
        pred = model(x_batch)
        loss = loss_criterion(pred, y_batch)
        loss.backward()
        optimizer.step()
    model.eval()
```