Tensorflow 2

Core API

Tensorflow API Hierarchy

- Tensorflow exposes APIs at multiple abstraction levels

Easier model design and training

tf.estimator, tf.keras, tf.data	High-level APIs for distributed training
tf.losses, tf.metrics, tf.optimizers, etc.	Components useful when building custom NN models
Core TensorFlow (Python)	Python API gives you full control
Core TensorFlow (C++)	C++ API is quite low level
CPU, GPU, TPU, Others	TF Runs on different hardware

More customization

Tensors

- The main components of the Core API are classes to represent tensors (of course)
  - Numeric constants of any shape
  - Trainable weights of your model
  - Etc.

Tensors

- First, import tensorflow:

```
import tensorflow as tf
```

- Create a constant scalar tensor:

```
x = tf.constant(3)
print(x.shape) # ()
```

- Create a constant rank-1 tensor:

List-like value (list, tuple, numpy array, etc.)

```
x = tf.constant([1, 2, 3, 4])
print(x.shape) # (4,)
```

Tensors

- Create a constant rank-2 tensor:

List of lists, or tuple of tuples, etc.

```
x = tf.constant([[1, 2, 3], [4, 5, 6]])
print(x.shape) # (2, 3)
```

- Create a constant rank-3 tensor:

```
x = tf.constant([[[1, 2, 3], [4, 5, 6]],
                 [[0, 2, 4], [1, 3, 5]]])
print(x.shape) # (2, 2, 3)
```

- And so on….

Tensors

- In general:

```
tf.constant(value, dtype=None, shape=None, name='Const')
```

Optional data type (e.g. `tf.int32`, `tf.float32`)
Default: inferred from value

For debugging, tensorboard etc.

If set, value reshaped to match.
Scalars are expanded, e.g.: `tf.constant(4, shape=(3,3))`
Default: inferred from value

Tensors

- You can **stack** tensors on top of each other:

```
x1 = tf.constant([1, 2, 3]) # shape: (3,)
x2 = tf.constant([3, 4, 5]) # shape: (3,)
x3 = tf.stack([x1, x2]) # shape: (2, 3)
```

- And take **slices**, just like with numpy arrays

```
x4 = x3[:, 0] # all rows, first column, shape: (2,)
# subtle difference
x4 = x3[:, 0:1] # same values, shape: (2,1) -> rank-2
```

Tensors

- You can reshape tensors:

```
x = tf.constant([[1, 2, 3], [4, 5, 6]])
print(x.shape) # (2, 3)
y = tf.reshape(x, [3, 2])
print(y)
# [[1, 2],
#  [3, 4],
#  [5, 6]]
```

- Remember: reshape reads tensors **by row**

Tensors

- `tf.constant` produces constant tensors
- `tf.Variable` produces tensors that can be modified (e.g. model weights!)

```
# x <- 2
x = tf.Variable(2, dtype=tf.float32, name='my_variable')
# x <- 10.3
x.assign(10.3)
# x <- x + 4
x.assign_add(4)
# x <- x - 1.2
x.assign_sub(1.2)
```

Tensors

- In general:

tf.Variable(initial_value=None, trainable=None, name=None, dtype=None, shape=None)

For debugging, tensorboard, etc.

Any tensor-like object (int/float, list, numpy array, tf.constant, etc.)

Default, inferred from initial_value
Normally, dtype and shape are **fixed after construction**

Tells `GradientTape()` (see after) whether to consider this variable or not.

Tensors

- Just like any `Tensor`, variables can be used as inputs to tf operations. Additionally, all the operators overloaded for the `Tensor` class are carried over to variables:

```
w = tf.Variable([[1.], [2.]]) # shape: (2, 1)
x = tf.constant([[3., 4.]]) # shape: (1, 2)
z = tf.matmul(w, x) # shape: (2, 2)

w = tf.Variable([[1., 2.]]) # shape: (1, 2)
x = tf.constant([[3., 4.]]) # shape: (1, 2)
z = w + x # shape: (1, 2)
```

Tensor Ops

- TensorFlow offers a rich library of operations (tf.add, tf.matmul, tf.linalg.inv etc.) that consume and produce tf.Tensors. These operations automatically convert native Python types:

- Point-wise operations (many more):

```
a=tf.constant([5,3,8])
b=tf.constant([3,-1,2])

c=tf.add(a, b)
# with overloading
c=a+b

d=tf.multiply(a, b)
# with overloading
d=a*b

e=tf.math.exp(a)
```

Tensor Ops

- Ops can also work on native Python lists and numpy arrays:

```
# native python list
a_py=[1,2]
b_py=[3,4]
tf.add(a_py, b_py)

# numpy arrays
a_np=np.array([1,2])
b_np=np.array([3,4])
tf.add(a_np, b_np)
```

- TF Tensor to NumPy array conversion (mostly done automatically by numpy ops):

```
x_np = x_tf.numpy()
```

Tensor Ops

- Many TensorFlow operations are accelerated using the GPU for computation. Without any annotations, TensorFlow automatically decides whether to use the GPU or CPU for an operation—copying the tensor between CPU and GPU memory, if necessary.

```
x = tf.constant([1, 2, 3])
print(x.device)
# something like:/job:localhost/replica:0/task:0/device:CPU:0
```

Tensor Ops

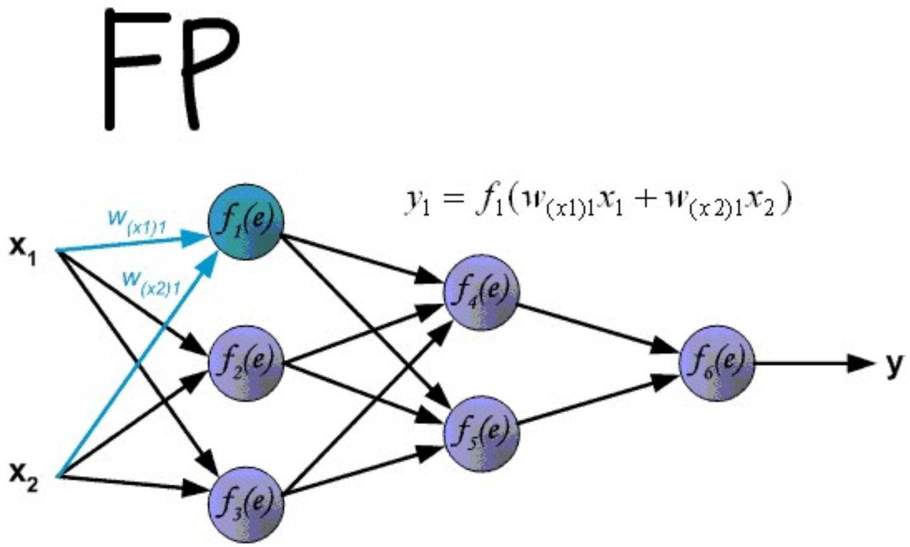
- You can force the execution on one particular device using the `tf.device` context manager:

```
# CPU:0 for the main system’s CPU.
# GPU:0 for the 1st GPU, GPU:1 for the 2nd GPU, etc.
with tf.device("GPU:0"):
    x = tf.random.uniform([1000, 1000])
    y = tf.matmul(x, x)
```

GradientTape

- Tensorflow has the ability to calculate the partial derivative of a function with respect to any variable automatically.

- To do so:
  - The function must be expressed using **only TensorFlow ops** (not arbitrary Python code)
  - The computation of the function must be recorded using TF's `GradientTape()` so:
    - TF can remember what operations happened and in what order during the **forward pass**.
    - During the **backward pass**, these operations are traversed in reverse order to compute gradients



source: medium.com

GradientTape

- `GradientTape()` is a so-called context manager within which these gradients are computed in TensorFlow.

Any function made of tf ops.
The computation must be recorded when the function is executed (not defined)

```
w0 = tf.Variable(0.0)
w1 = tf.Variable(0.0)
with tf.GradientTape() as tape:
    y = my_func(X, Y, w0, w1)
dw0, dw1 = tape.gradient(y, [w0, w1])
```

Compute the gradient of any function recorded in tape with respect to any parameter.

TF Core API

- **Notebook**: Linear_Regression_from_Scratch.ipynb

Linear Regression from Scratch

- Create a toy training dataset:

```
X_train=tf.constant(range(10), dtype=tf.float32)
# overloaded operators
Y_train=3*X_train + 5 + tf.random.normal(X_train.shape, 0.0,
0.1)

print(”Train X:{}".format(X_train))
print(”Train Y:{}".format(Y_train))
```

Random tensor drawn from a Gaussian with mean = 0.0, std = 0.1

Linear Regression from Scratch

- Create a toy test dataset:

```
X_test=tf.constant(range(10, 20), dtype=tf.float32)
Y_test=3 * X_test + 5 + tf.random.normal(X_test.shape, 0.0, 0.1)

print(”Test X:{}".format(X_test))
print(”Test Y:{}".format(Y_test))
```

Linear Regression from Scratch

- Define our model as: $y = w1 \ * x + w0$

```
def my_model(X, w0, w1):
    return w1*X + w0
```

- Define a Mean Squared Error (MSE) loss function, since this is a regression problem (*):

```
def loss_mse(X, Y, w0, w1):
    Y_hat=my_model(X, w0, w1)
    return tf.reduce_mean((Y_hat-Y)**2)
```

  - (*) Note that the MSE loss is already defined in `tf.losses`, here we're re-inventing the wheel

Linear Regression from Scratch

- Define a function to compute the gradients of the model weights with respect to the loss, using `GradientTape()`:

```
def compute_gradients(X, Y, w0, w1):
    with tf.GradientTape() as tape:
          loss=loss_mse(X, Y, w0, w1)
    return tape.gradient(loss, [w0, w1])
```

Linear Regression from Scratch

- Build the training loop (initialize constants and weights….):

```
STEPS=1000
LEARNING_RATE=.02

w0=tf.Variable(0.0)
w1=tf.Variable(0.0)
```

Remember, this is linear regression. Never initialize your weights to zero in NNs.

Linear Regression from Scratch

- Build the training loop (…and train):

```
for i in range(STEPS):
    dw0, dw1 = compute_gradients(X_train, Y_train, w0, w1)
    w0.assign_sub(dw0 * LEARNING_RATE)
    w1.assign_sub(dw1 * LEARNING_RATE)

    if i % 100 == 0:
            loss = loss_mse(X_train, Y_train, w0, w1)
            print("Step {}, Loss: {}, w0: {}, w1: {}\n".format(
                  i, loss, w0.numpy(), w1.numpy()))
```

- Note: no mini-batches, no validation set, etc. It's a toy example....

Linear Regression from Scratch

- Evaluate results on test set:

```
loss=loss_mse(X_test, Y_test, w0, w1)
loss.numpy()
```