TensorFlow

Курс “Практическое применение по TensorFlow”

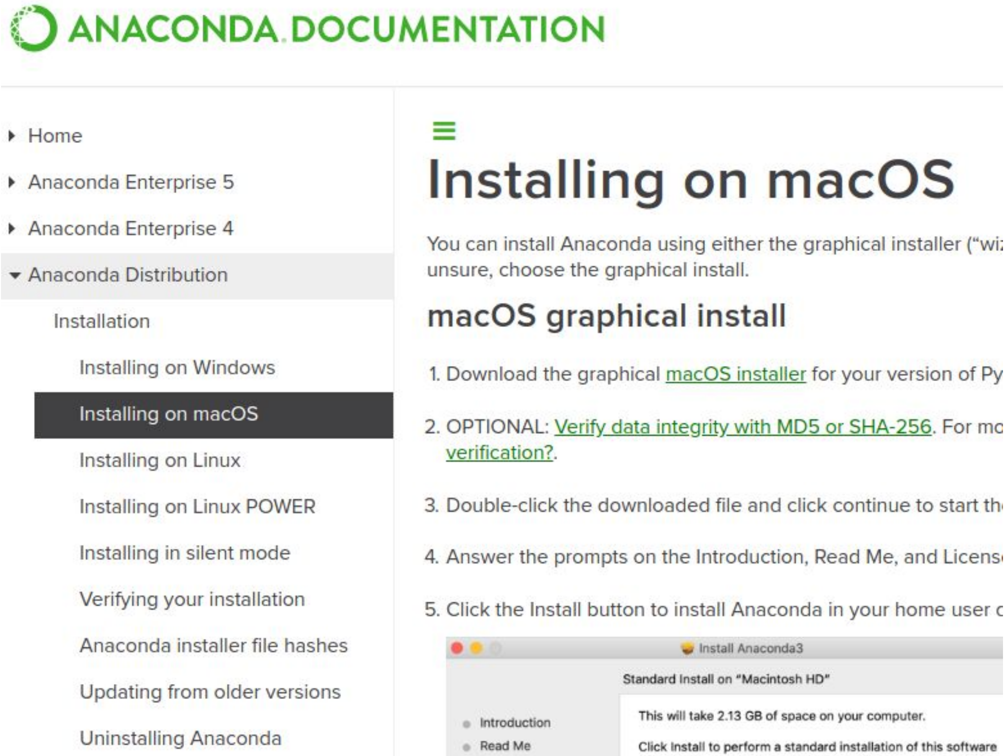
Шигапова Фирюза Зинатуллаевна

1-й семестр, 2019 г.

https://github.com/Firyuza/TensorFlowPractice

TensorFlow 2.0 installation

1. Install Anaconda *https://docs.anaconda.com/anaconda/install/#*



TensorFlow 2.0 installation

2. Create conda environment:

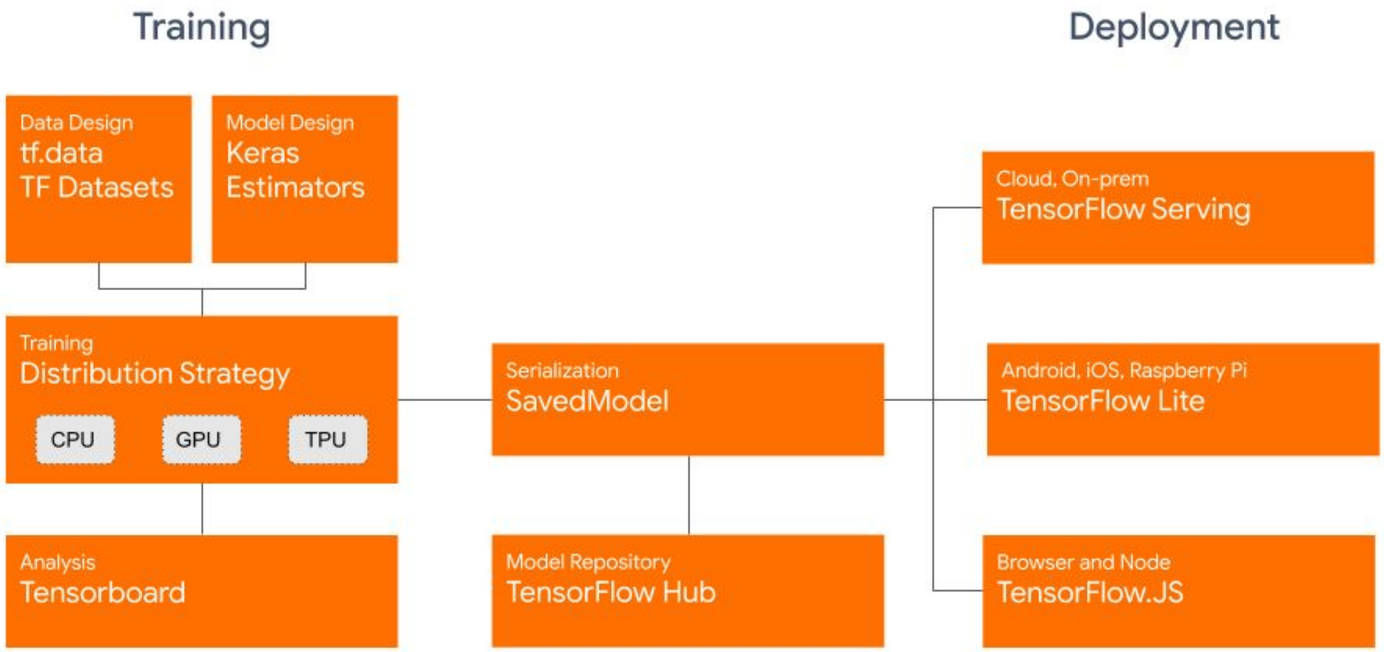
conda create -n tensorflow_2 python=3.x

3. Install TensorFlow into created environment (tensorflow 2.0):

source activate tensorflow_2

pip install tensorflow or **pip install tensorflow-gpu** (Now by default tensorflow 2.0)

https://www.tensorflow.org/install/pip



Source: https://medium.com/tensorflow/tensorflow-2-0-is-now-available-57d706c2a9ab

TensorFlow 1.x vs TensorFlow 2.0

TensroFlow 1.x	TensorFlow 2.0
	Eager execution
Static Graph	tf.function, AutoGraph
Session	Eager execution
Queue runner	tf.data
tf.train.Coordinator	tf.distribute.Strategy
tf.control_dependencies	*Does not exist anymore*
Slim, Keras, Estimator	Keras, Estimator
	tf.GradientTape
	TFX, TF Lite (and micro), TFjs,

Why TensorFlow 1.x is alive?

While eager execution makes development and debugging more interactive, TensorFlow graph execution has advantages for distributed training, performance optimizations, and production deployment. However, writing graph code can feel different than writing regular Python code and more difficult to debug.

Source: https://www.tensorflow.org/guide/eager

TensorFlow 1.x vs. TensorFlow 2.0

TensorFlow 1.x

```
in_a = tf.placeholder(dtype=tf.float32, shape=(2))
in_b = tf.placeholder(dtype=tf.float32, shape=(2))

def forward(x):
  with tf.variable_scope("matmul", reuse=tf.AUTO_REUSE):
    W = tf.get_variable("W", initializer=tf.ones(shape=(2,2)),
                        regularizer=tf.contrib.layers.l2_regularizer(0.04))
    b = tf.get_variable("b", initializer=tf.zeros(shape=(2)))
    return W * x + b

out_a = forward(in_a)
out_b = forward(in_b)

reg_loss = tf.losses.get_regularization_loss(scope="matmul")

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  outs = sess.run([out_a, out_b, reg_loss],
                feed_dict={in_a: [1, 0], in_b: [0, 1]})
```

TensorFlow 2.x

```
W = tf.Variable(tf.ones(shape=(2,2)), name="W")
b = tf.Variable(tf.zeros(shape=(2)), name="b")

def forward(x):
  return W * x + b

out_a = forward([1,0])
print(out_a)
```

tf.function and AutoGraph

- **tf.function** allows to transform Python syntax into high-performance TensorFlow Static Graph

- Or “defines a **computations** as a **graph** of TensorFlow operations”

- **tf.function** “accelerates” part of code but doesn’t allow to debug at runtime

- Using tf.function **AutoGraph** automatically applied (tf.autograph)

Source: https://github.com/tensorflow/community/blob/master/rfcs/20180918-functions-not-sessions-20.md

tf.function and AutoGraph

```
v = tf.Variable(1.0)
@tf.function
def f():
  v.assign(2.0)
  return v.read_value()

print(f()) # Always prints 2.0.
```

@tf.function uses it

```
def foo(x):
    if x > 0:
        y = x * x
    else:
        y = -x
    return y

converted_foo = tf.autograph.to_graph(foo)

x = tf.constant(1)
y = converted_foo(x)  # converted_foo is a TensorFlow Op-like.
assert tf.is_tensor(y)
```

Source: https://www.tensorflow.org/api_docs/python/tf/autograph/to_graph

tf.function and AutoGraph

```python
@tf.function
def f(x):
  if x > 0:
    x = x + 1
  return x

tf.config.experimental_run_functions_eagerly(True)
print(f(tf.constant(1))) # tf.Tensor(2, shape=(), dtype=int32)
```

To debug @tf.function

Source: https://www.tensorflow.org/guide/effective_tf2

Build Model

1. Functional API

```
self.conv3x3_a = tf.keras.layers.Conv2D(filters, (kernel_size, kernel_size), padding='same')
self.conv3x3_a_gn = GroupNorm(filters)
self.conv3x3_b = tf.keras.layers.Conv2D(filters, (kernel_size, kernel_size), padding='same')
self.conv3x3_b_gn = GroupNorm(filters)
self.max_pool = tf.keras.layers.MaxPool2D(pool_size=(max_pool_size, max_pool_size))
```

2. Sequential API

```
self.sequential_model = tf.keras.Sequential()
self.sequential_model.add(tf.keras.layers.Conv2D(filters, (kernel_size, kernel_size), padding='same'))
self.sequential_model.add(GroupNorm(filters))
self.sequential_model.add(tf.keras.layers.Conv2D(filters, (kernel_size, kernel_size), padding='same'))
self.sequential_model.add(GroupNorm(filters))
self.sequential_model.add(tf.keras.layers.MaxPool2D(pool_size=(max_pool_size, max_pool_size)))
```

Run model via Keras API

It has the **same** API as it is used in **TensorFlow 1.x**

Next slide 👉

Keras. Compile model

Define necessary components for model **training**:

- optimizer
- loss function
- metrics

```
# Configure a model for categorical classification.
model.compile(optimizer=tf.keras.optimizers.RMSprop(0.01),
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=[tf.keras.metrics.CategoricalAccuracy()])
```

Source: https://www.tensorflow.org/tutorials/keras/classification

Keras. Fit model

Define dataset for training and validation phases:

- training data (**tf.data** can be used)
- validation data (**tf.data** can be used)
- epochs
- batch size

```
model.fit(data, labels, epochs=10, batch_size=32,
          validation_data=(val_data, val_labels))
```

Source: https://www.tensorflow.org/tutorials/keras/classification

Keras. Evaluate model

Returns loss and metrics values on valid/test set

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

Source: https://www.tensorflow.org/tutorials/keras/classification

Keras. Predict model

Returns predictions of trained model at inference

```
predictions = model.predict(test_images)
```

Source: https://www.tensorflow.org/tutorials/keras/classification

Custom “components”

- Layers
- Models
- Loss

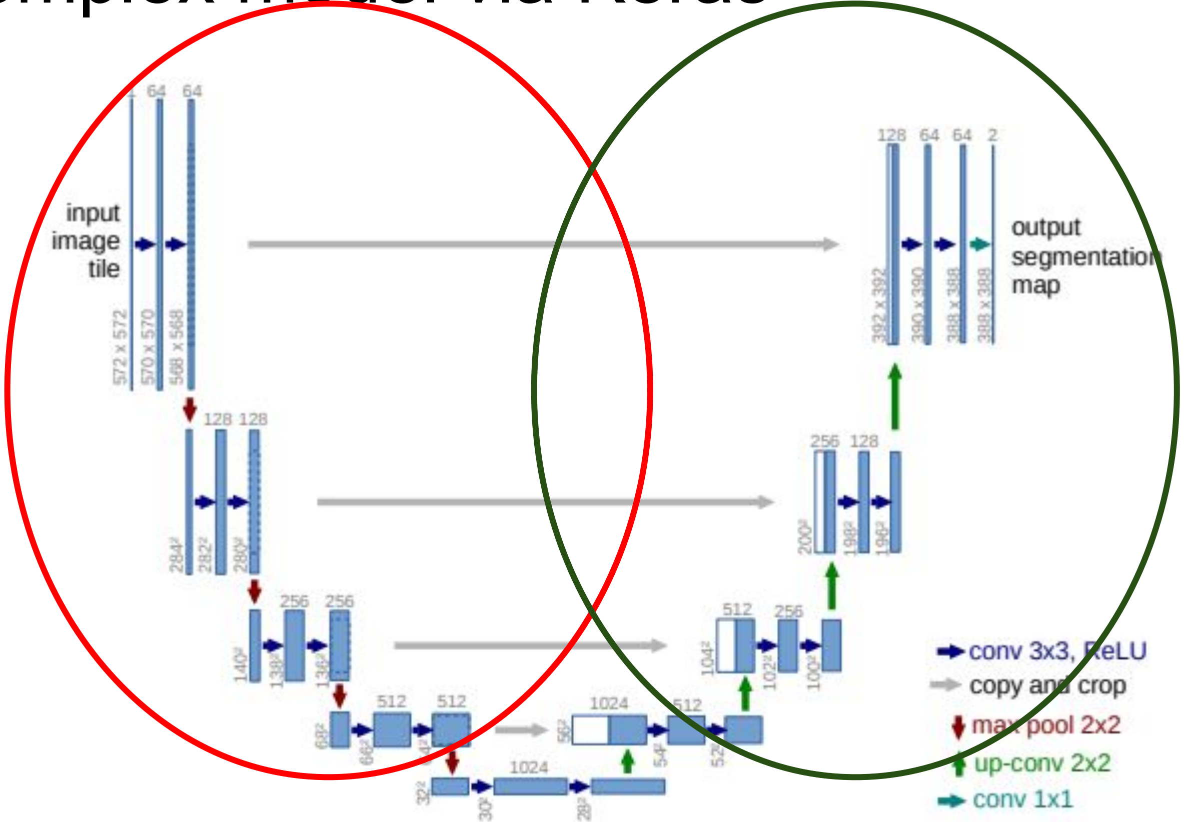
Have **similar** creating logic

- Callbacks

- Accuracy

Build complex model via Keras

Down
convolutions

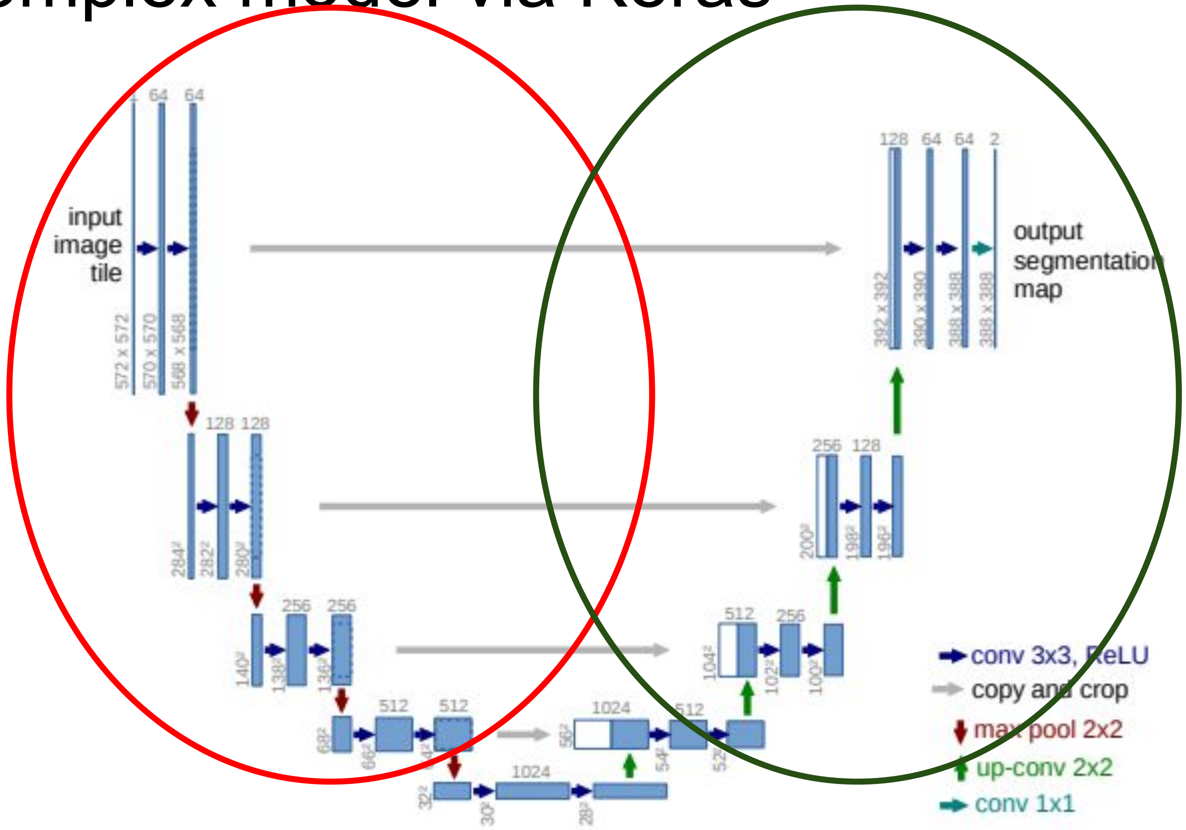


Up
convolutions

Source: https://arxiv.org/pdf/1505.04597.pdf

Build complex model via Keras

Create
custom
Layer



Create
custom
Layer

Source: https://arxiv.org/pdf/1505.04597.pdf

tf.keras.layers.Layer

1. **Inherit** *custom* layer from **tf.keras.layers.Layer**

```python
class UNetDownBlock(tf.keras.layers.Layer):
    def __init__(self, filters, kernel_size, max_pool_size):
        super(UNetDownBlock, self).__init__()
```

2. Create **trainable** variables in **constructor** or in **build** method

```python
class UNetDownBlock(tf.keras.layers.Layer):
    def __init__(self, filters, kernel_size, max_pool_size):
        super(UNetDownBlock, self).__init__()
        self.conv3x3_a = tf.keras.layers.Conv2D(filters, (kernel_size, kernel_size), padding='same')
        self.conv3x3_a_gn = GroupNorm(filters)
        self.conv3x3_b = tf.keras.layers.Conv2D(filters, (kernel_size, kernel_size), padding='same')
        self.conv3x3_b_gn = GroupNorm(filters)
        self.max_pool = tf.keras.layers.MaxPool2D(pool_size=(max_pool_size, max_pool_size))
```

3. In **call** method write **forward** operations

```python
def call(self, input_tensor, is_training=True):
    output = self.conv3x3_1a(input_tensor)
    output = self.conv3x3_1a_gn(output)
    output_1 = self.conv3x3_1b(output)
    output_1 = self.conv3x3_1b_gn(output_1)
    output = self.max_pool_1(output_1)

    return output_1, output
```

tf.keras.layers.Layer

1. Use **Layers** as outputs for **model** creating

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

2. Use **standard** keras API:

- compile
- fit
- evaluate
- etc.

Different architectures can be wrapped as single keras Layer

- ResNet
- Variational AutoEncoder
- etc.

tf.keras.models.Model

1. Wrap custom Layers into **Model** (tf.keras.models.Model)
2. Build **tf.keras.Model** as tf.keras.layers.Layer:
 - in **constructor** or in **build** method create object of custom **Layer**
 - in **call** method write **forward** operations of the **Model**
3. In **train** pipeline:
 - create Model object
 - build object

```
self.unet = UNet()
self.unet.build((None, cfg.train.image_size,  cfg.train.image_size, 3))
```

4. At train step just use created Model object for calling **forward operations**

```
for image, label in dataset:
    with tf.GradientTape() as tape:
        segmentation_map = self.unet(image, True)
```

tf.keras.models.Model

5. Or use **standard** keras API:

- compile
- fit
- etc.

Custom Loss

1. **Inherit** *loss* from **tf.keras.losses.Loss**

```
class MaxMargin(tf.keras.losses.Loss):
    def __init__(self, delta, from_logits=False,
                 reduction=tf.keras.losses.Reduction.AUTO,
                 name='max_margin'):
        super(MaxMargin, self).__init__(reduction=reduction,
                                        name=name)
```

2. Create needed **variables** in **constructor** or in **build** method
3. In **call** method write **forward** operations

```
def call(self, v1, v2):
    norm = tf.sqrt(tf.reduce_sum(tf.square(tf.subtract(v1, v2, axis=1), axis=1), axis=1))

    return tf.maximum(0., self.delta - norm)
```

Custom Loss

4. In train pipeline create Loss object

```
self.max_margin_loss = MaxMargin(cfg.train.delta)
```

5. At train step just use created Loss object

```
total_loss += self.max_margin_loss(emb1, emb2)
```

Custom Accuracy

1. **Inherit** from **tf.keras.metrics.Metric**
2. In constructor add variable for tracking custom accuracy value

```
class BinaryTruePositives(tf.keras.metrics.Metric):
  def __init__(self, name='binary_true_positives', **kwargs):
    super(BinaryTruePositives, self).__init__(name=name, **kwargs)
    self.true_positives = self.add_weight(name='tp', initializer='zeros')
```

3. Update accuracy value using **update_state** method

```
def update_state(self, y_true, y_pred, sample_weight=None):
  y_true = tf.cast(y_true, tf.bool)
  y_pred = tf.cast(y_pred, tf.bool)

  values = tf.logical_and(tf.equal(y_true, True), tf.equal(y_pred, True))
  values = tf.cast(values, self.dtype)

  self.true_positives.assign_add(tf.reduce_sum(values))
```

Custom Accuracy

4. Override result method for getting current accuracy value

```
def result(self):
    return self.true_positives
```

5. Work in pipeline as with pre-made keras accuracies

Custom Callback

1. **Inherit** custom callback from **tf.keras.callbacks.Callback**

2. **Override** needed methods:

- on_train_batch_begin
- on_train_batch_end
- on_epoch_begin
- etc

Custom Callback

```
class MyCustomCallback(tf.keras.callbacks.Callback):

  def on_train_batch_begin(self, batch, logs=None):
    print('Training: batch {} begins at {}'.format(batch, datetime.datetime.now().time()))

  def on_train_batch_end(self, batch, logs=None):
    print('Training: batch {} ends at {}'.format(batch, datetime.datetime.now().time()))

  def on_test_batch_begin(self, batch, logs=None):
    print('Evaluating: batch {} begins at {}'.format(batch, datetime.datetime.now().time()))

  def on_test_batch_end(self, batch, logs=None):
    print('Evaluating: batch {} ends at {}'.format(batch, datetime.datetime.now().time()))
```

Source: https://www.tensorflow.org/guide/keras/custom_callback

tf.GradientTape

- Track operations for automatic differentiation
- All **trainable** variable are **automatically** watched
- For manually watching use method **watch()**

tf.GradientTape. How To Use?

- Create context using **tf.GradientTape()**

```
with tf.GradientTape() as tape:
    segmentation_map = self.unet(image, True)
```

- Operations recorded by *GradientTape* if they executed **within** this **context** manager or by calling **watch()**
- Compute **gradient** and **apply** them to **optimizer**

```
grads = tape.gradient(loss_value, self.unet.trainable_variables)
self.optimizer.apply_gradients(zip(grads, self.unet.trainable_variables))
```

tf.GradientTape. How To Use?

```
with tf.GradientTape() as tape:
    segmentation_map = self.unet(image, True)

    loss_value = self.unet_loss(label, segmentation_map)

grads = tape.gradient(loss_value, self.unet.trainable_variables)
self.optimizer.apply_gradients(zip(grads, self.unet.trainable_variables))
```

Immediately compute gradients

Try toy example with **x = tf.Variable(5.)**

tf.GradientTape. “Watching”

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x)
    y = x * x
dy_dx = g.gradient(y, x)  # 6.0
```

Non-trainable variable => have to manually “watch”

tf.GradientTape. Lifecycle

- By default **tf.GradientTape()** allows to call gradient method once and then tape is removed
- For multiple calls:

```
x = tf.constant(3.0)
with tf.GradientTape(persistent=True) as g:
    g.watch(x)
    y = x * x
    z = y * y
dz_dx = g.gradient(z, x) # 108.0
dy_dx = g.gradient(y, x)  # 6.0
```

tf.GradientTape. Control gradients

```
x = tf.Variable(3.0)
with tf.GradientTape(watch_accessed_variables=False) as g:
    y = x * x
dy_dx = g.gradient(y, x)  # None
```

No gradients will be computed since watching is **disabled**

```
x = tf.Variable(3.0)
with tf.GradientTape(watch_accessed_variables=False) as g:
    g.watch(x)
    y = x * x
dy_dx = g.gradient(y, x)  # 6
```

Manually watch needed operations