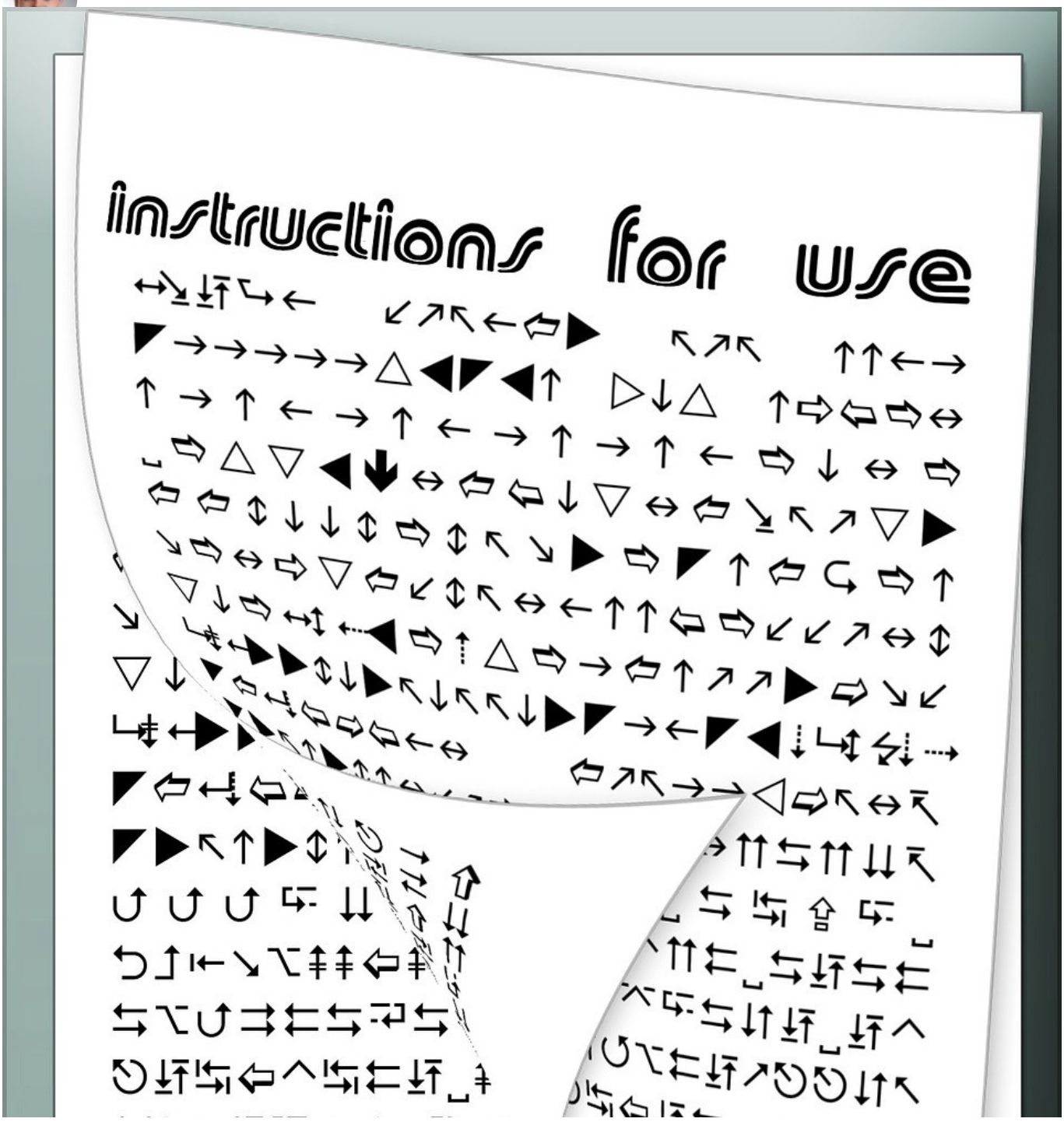
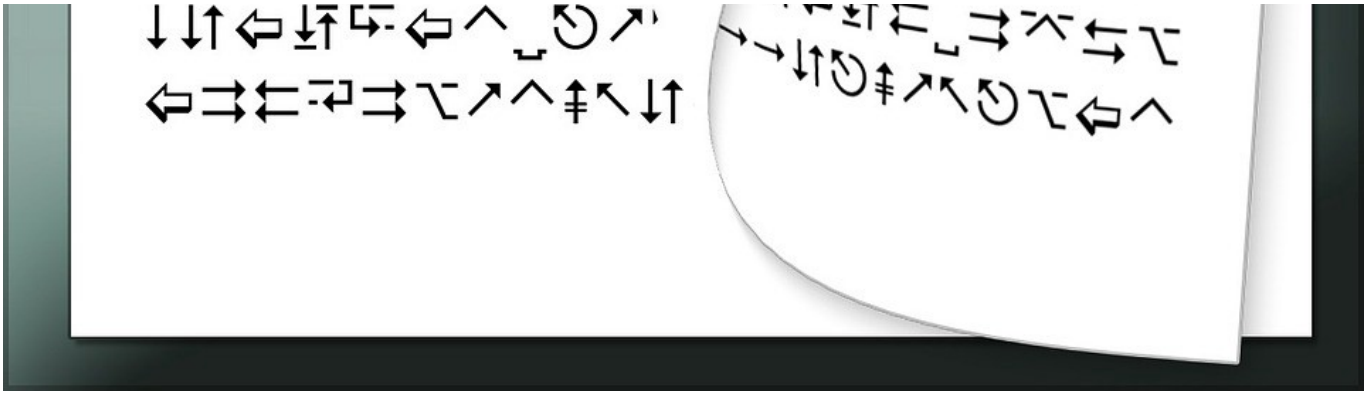
Practical Coding in TensorFlow 2.0

Eager execution, @tf.function, TensorArray, and advanced control flow

Dmitry Grebenyuk Following
Oct 12 · 4 min read ★





An image by Gerd Altmann

Summary

Fierce competition with PyTorch brought us a new version of TensorFlow (TF). The package has undergone many changes, but the key one is the retirement of `session.run()`. Instead of a familiar pattern of building and executing a static graph, TF 2 uses eager mode by default. Such code can be written in a pythonic fashion and converted into a computational graph. To execute the code as a static graph, a developer has to decorate the desired function with `@tf.function`.

In this post, I will explain these concepts using examples. I assume that the reader knows Python and the basics of machine learning (ML). If you are new to the field, welcome! This course will be an excellent start. Here, you can find a colab version of this post.

Installation

To install TF 2.x, please visit this page.

To check your current version:

```
import tensorflow as tf
print(tf.__version__)
```

Eager Execution

The code executes eagerly in TF 2. It means that you feed data to a computational graph without calling `session.run()` or using placeholders. The computational graph is a structure that defines a sequence of operations. That structure allows us to

calculate derivatives automatically by moving backwards along the graph. Watch this video for more detail. In TF 1, a developer had to create a graph, then execute it. Now the graph is constructed dynamically, and the execution is similar to a function call.

To see how it works, let's make a simple model. For example, we have a dataset with 3 training examples, where each example is a 2-dimensional vector.

```
import numpy as np

np.random.seed(0)
data = np.random.randn(3, 2)
```

First, we have to initialise variables. There are no variable scopes in TF 2. So the best way to keep all variables in a count is to use Keras layers.

```
inputer = tf.keras.layers.InputLayer(input_shape=(2))

denser1 = tf.keras.layers.Dense(4, activation='relu')

denser2 = tf.keras.layers.Dense(1, activation='sigmoid')
```

Then we can define a simple model. We can run the model by merely calling it as a function. Here, the data goes into a dense layer with 4 hidden units, then into a final layer with one unit.

```
def model_1(data):

x = inputer(data)
  x = denser1(x)
  print('After the first layer:', x)
  out = denser2(x)
  print('After the second layer:', out)

return out

print('Model\'s output:', model(data))
```

```
...
After the first layer: tf.Tensor(
[[0.9548421  0.         0.         1.4861959 ]
 [1.3276602  0.18780036 0.50857764 0.        ]
 [0.45720425 0.         0.         2.5268495 ]], shape=(3, 4),
dtype=float32)
After the second layer: tf.Tensor(
[[0.27915245]
 [0.31461754]
 [0.39550844]], shape=(3, 1), dtype=float32)
Model's output: tf.Tensor(
[[0.27915245]
 [0.31461754]
 [0.39550844]], shape=(3, 1), dtype=float32)
```

To get a numpy array from a tensor:

```
print('Model\'s output:', model_1(data).numpy())
```

```
...
Model's output: [[0.27915245]  [0.31461754]  [0.39550844]]
```

However, the eager execution can be slow. The graph is computed dynamically. That is, let's see how it is built in our model. The input data goes in the first layer, which is the first node. When you add the second node, the first node's output goes in the second node, and the output of the second node is calculated, and so on. It allows us to print intermediate states of the model (as we did in the example above), but make computations slower.

Static graph

Fortunately, we still can construct a static graph by decorating the model with `@tf.function`. A static graph, in contrast to dynamic, first connects all the nodes making one big computational operation, then executes it. Thus, we cannot see intermediate states of the model or add any nodes on the fly.

```
@tf.function
def model_2(data):
  x = inputer(data)
  x = denser1(x)
  print('After the first layer:', x)
```

```
    out = denser2(x)
    print('After the second layer:', out)

    return out

print('Model\'s output:', model_2(data))
```

```
...
After the first layer: Tensor("dense_12/Relu:0", shape=(3, 4),
dtype=float32)
After the second layer: Tensor("dense_13/Sigmoid:0", shape=(3, 1),
dtype=float32)
Model's output: tf.Tensor(
[[0.27915245]
 [0.31461754]
 [0.39550844]], shape=(3, 1), dtype=float32)
```

The second advantage is that a static graph is built only once, while dynamic rebuilt after each model call. It can slow down computations when you repeatedly reuse the same graph. For example, when you recalculate the loss from batches during the training.

```
for i, d in enumerate(data):
  print('batch:', i)
  model_1(d[np.newaxis, :])  # eager model

for i, d in enumerate(data):
  print('batch:', i)
  model_2(d[np.newaxis, :])  # static model
```

```
...
batch: 0
After the first layer: tf.Tensor(
[[0.9548421 0.        0.        1.486196 ]], shape=(1, 4),
dtype=float32)
After the second layer: tf.Tensor(
[[0.27915245]], shape=(1, 1), dtype=float32)

batch: 1
After the first layer: tf.Tensor(
[[1.3276603  0.18780035 0.50857764 0.        ]], shape=(1, 4),
dtype=float32)
After the second layer: tf.Tensor(
[[0.3146175]], shape=(1, 1), dtype=float32)

batch: 2
After the first layer: tf.Tensor(
```

```
[[0.45720425 0.         0.         2.5268495 ]], shape=(1, 4),
dtype=float32)
After the second layer: tf.Tensor(
[[0.39550844]], shape=(1, 1), dtype=float32)

batch: 0
After the first layer: Tensor("dense_12/Relu:0", shape=(1, 4),
dtype=float32)
After the second layer: Tensor("dense_13/Sigmoid:0", shape=(1, 1),
dtype=float32)

batch: 1

batch: 2
```

The internal prints are called only during the graph building and, in the second case, the graph is built only once and then reused. For big datasets, that difference in time can be dramatic.

Advanced control flow

AutoGraph simplifies the use of if/else statements and for/while loops. In contrast to TF 1, now they can be written using python syntaxis. For example:

```
a = np.array([1, 2, 3], np.int32)

@tf.function
def foo(a):
  b = tf.TensorArray(tf.string, 4)
  b = b.write(0, "test")

  for i in tf.range(3):
    if a[i] == 2:
      b = b.write(i, "fuzz")
    elif a[i] == 3:
      b = b.write(i, "buzz")

return b.stack()

...
tf.Tensor([b'test' b'fuzz' b'buzz' b''], shape=(4,), dtype=string)
```

The use of arrays is now similar to one in Java. First, you declare an array with the desired data type and length:

```
tf.TensorArray(data_type, length)
```

The common data types are `tf.int32, tf.float32, tf.string` . To transform array `b` back to a tensor, use `b.stack()` .

Machine Learning Programming TensorFlow