How to avoid NaN gradients when using `tf.where` to select finite results.

date: 2017-feb-7 [2019-jul-23]
owner: jvdillon

Abstract

So you have a NaN in your gradient and don't know why? Assuming you use `tf.where`, `tf.minimum`, `tf.maximum`, this note might help!

Tl;dr: *Instead of this:* `tf.where(x_ok, f(x), safe_f(x))`
Do this: `tf.where(x_ok, f(tf.where(x_ok, x, safe_x)), safe_f(x))`

Both give the same result. Only the latter gives the correct[1] gradient.

(Keep reading if the "tl;dr" doesn't make sense!)

Detailed Example

Let's develop our intuition of the problem by considering a specific example. Suppose you wish to differentiate the following function:

$$f(x) = \begin{cases} \frac{1}{x} & x \geq 1 \\ 0 & x < 1 \end{cases}.$$

A naive implementation results in NaNs in the gradient, i.e.,

```
import tensorflow.compat.v2 as tf
import tensorflow_probability as tfp
tf.enable_v2_behavior()

f = lambda x: tf.where(x < 1., 0., 1. / x)
x = tf.constant(0.)
tfp.math.value_and_gradient(f, x)[1]
# ==> nan  ...bah.
```

[1] Arguably. Don't ask.

The basic pattern for avoiding NaN gradients when using `tf.where` is to call `tf.where` *twice*.[2] The innermost `tf.where` ensures that the result `f(x)` is always finite. The outermost `tf.where` ensures the correct result is chosen. For the running example, the trick plays out like this:

```
def safe_f(x):
  safe_x = tf.where(tf.equal(x, 0.), 1., x)   # inner tf.where
  return tf.where(x < 1., 0., 1. / safe_x)    # outer tf.where; just like f(x)
```

But did it work?

```
x = tf.constant(0.)
tfp.math.value_and_gradient(safe_f, x)[1]
# ==> 0.0   ...yay! double-where trick worked.
```

General Recipe

1. Use an inner `tf.where` to ensure the function has no asymptote.
   I.e., alter the input to the inf generating function such that no inf can be created.
2. Use a second `tf.where` to always select the valid code-path.
   I.e., implement the mathematical condition as you would "normally", i.e., the "naive" implementation.

In Python code, the recipe is:

Instead of this: `tf.where(x_ok, f(x), alt_f(x))`
Do this: `tf.where(x_ok, f(tf.where(x_ok, x, safe_x)), alt_f(x))`

Can we do better?

With luck, sometimes things work out even more cleanly. For example,

```
def cross_entropy(x, y, axis=-1):
  safe_y = tf.where(tf.equal(y, 0.), tf.ones_like(y), y)
  return -tf.reduce_sum(x * tf.math.log(safe_y), axis)

def entropy(x, axis=-1):
  return cross_entropy(x, x, axis)
```

Here we only needed one `tf.where` because the `x *` acts like an outer `tf.where`.

But did it work?

[2] The "double-tf.where" trick always works, assuming you have access to the data *before* it becomes +/- inf.

```
x = tf.constant([0.1, 0.2, 0., 0.7])
e = entropy(x)
# ==> 0.80181855
tfp.math.value_and_gradient(entropy, x)[1]
# ==> [1.30258512,  0.60943794, 0., -0.64332503]  ...yay! no nan.
```

For additional discussion, see [this StackOverflow post](#).