tf.scan

```
scan(
    fn,
    elems,
    initializer=None,
    parallel_iterations=10,
    back_prop=True,
    swap_memory=False,
    infer_shape=True,
    name=None
)
```

Defined in `tensorflow/python/ops/functional_ops.py`.

See the guide: Higher Order Functions > Higher Order Operators

scan on the list of tensors unpacked from `elems` on dimension 0.

The simplest version of `scan` repeatedly applies the callable `fn` to a sequence of elements from first to last. The elements are made of the tensors unpacked from `elems` on dimension 0. The callable fn takes two tensors as arguments. The first argument is the accumulated value computed from the preceding invocation of fn. If `initializer` is None, `elems` must contain at least one element, and its first element is used as the initializer.

Suppose that `elems` is unpacked into `values`, a list of tensors. The shape of the result tensor is `[len(values)] + fn(initializer, values[0]).shape`.

This method also allows multi-arity `elems` and accumulator. If `elems` is a (possibly nested) list or tuple of tensors, then each of these tensors must have a matching first (unpack) dimension. The second argument of `fn` must match the structure of `elems`.

If no `initializer` is provided, the output structure and dtypes of `fn` are assumed to be the same as its input; and in this case, the first argument of `fn` must match the structure of `elems`.

If an `initializer` is provided, then the output of `fn` must have the same structure as `initializer`; and the first argument of `fn` must match this structure.

For example, if `elems` is `(t1, [t2, t3])` and `initializer` is `[i1, i2]` then an appropriate signature for `fn` in `python2` is: `fn = lambda (acc_p1, acc_p2), (t1 [t2, t3]):` and `fn` must return a list, `[acc_n1, acc_n2]`. An alternative correct signature for `fn`, and the one that works in `python3`, is: `fn = lambda a, t:`, where `a` and `t` correspond to the input tuples.

Args:

- `fn`: The callable to be performed. It accepts two arguments. The first will have the same structure as `initializer` if one is provided, otherwise it will have the same structure as `elems`. The second will have the same (possibly nested) structure as `elems`. Its output must have the same structure as `initializer` if one is provided, otherwise it must have the same structure as `elems`.
- `elems`: A tensor or (possibly nested) sequence of tensors, each of which will be unpacked along their first dimension. The nested sequence of the resulting slices will be the first argument to `fn`.
- `initializer`: (optional) A tensor or (possibly nested) sequence of tensors, initial value for the accumulator, and the

expected output type of `fn`.

- `parallel_iterations`: (optional) The number of iterations allowed to run in parallel.
- `back_prop`: (optional) True enables support for back propagation.
- `swap_memory`: (optional) True enables GPU-CPU memory swapping.
- `infer_shape`: (optional) False disables tests for consistent output shapes.
- `name`: (optional) Name prefix for the returned tensors.

Returns:

A tensor or (possibly nested) sequence of tensors. Each tensor packs the results of applying `fn` to tensors unpacked from `elems` along the first dimension, and the previous accumulator value(s), from first to last.

Raises:

- `TypeError`: if `fn` is not callable or the structure of the output of `fn` and `initializer` do not match.
- `ValueError`: if the lengths of the output of `fn` and `initializer` do not match.

Examples:

```
elems = np.array([1, 2, 3, 4, 5, 6])
sum = scan(lambda a, x: a + x, elems)
# sum == [1, 3, 6, 10, 15, 21]

elems = np.array([1, 2, 3, 4, 5, 6])
initializer = np.array(0)
sum_one = scan(
    lambda a, x: x[0] - x[1] + a, (elems + 1, elems), initializer)
# sum_one == [1, 2, 3, 4, 5, 6]

elems = np.array([1, 0, 0, 0, 0, 0])
initializer = (np.array(0), np.array(1))
fibonaccis = scan(lambda a, _: (a[1], a[0] + a[1]), elems, initializer)
# fibonaccis == ([1, 1, 2, 3, 5, 8], [1, 2, 3, 5, 8, 13])
```

Last updated November 2, 2017.