

```

1     import numpy as np
2     import operator as op
3     import itertools as it
4     from functools import partial
5     from core import primitive, getval, untake
6
7     P = primitive
8
9     # ----- Operator gradients -----
10    I = lambda x : x # Identity operator
11    op.neg = P(op.neg, lambda ans, x : [op.neg])
12    op.add = P(op.add, lambda ans, x, y : unbroadcast(ans, x, y, [I, I]))
13    op.mul = P(op.mul, lambda ans, x, y : unbroadcast(ans, x, y, [lambda g : y * g, lambda g : x * g]))
14
15    op.sub = P(op.sub, lambda ans, x, y : unbroadcast(ans, x, y, [I, op.neg]))
16    op.div = P(op.div, lambda ans, x, y : unbroadcast(ans, x, y, [lambda g : g / y, lambda g : - g * x /
17    y**2]))
18    op.pow = P(op.pow, lambda ans, x, y : unbroadcast(ans, x, y, [lambda g : g * y * x ** (y - 1),
19
20
21
22    lambda g : g * np.log(x) * x ** y]))
23
24    isarray = lambda x : isinstance(getval(x), np.ndarray)
25    isfloat = lambda x : isinstance(getval(x), float)
26
27    def unbroadcast(ans, x, y, funs):
28        return [unbroadcast_fun(ans, x, funs[0]),
29                unbroadcast_fun(ans, y, funs[1])]

```

```

24
25 ✓ def unbroadcast_fun(ans, x, fun):
26     if isfloat(x) and isarray(ans):
27         return lambda g : np.sum(fun(g))
28     elif isarray(x):
29         shape = x.shape
30 ✓     def new_fun(g):
31         result = fun(g)
32         while result.ndim > len(shape):
33             result = np.sum(result, axis=0)
34         for axis, size in enumerate(shape):
35             if size is 1:
36                 result = np.sum(result, axis, keepdims=True)
37         return result
38     return new_fun
39 else:
40     return fun
41
42 # ----- Numpy gradients -----
43
44 np.abs      = P(np.abs,      lambda ans, x : [lambda g : np.sign(x) * g])
45 np.exp      = P(np.exp,      lambda ans, x : [lambda g : ans * g])
46 np.log      = P(np.log,      lambda ans, x : [lambda g : g / x])
47 np.sin      = P(np.sin,      lambda ans, x : [lambda g : g * np.cos(x)])
48 np.cos      = P(np.cos,      lambda ans, x : [lambda g : - g * np.sin(x)])
49 np.tan      = P(np.tan,      lambda ans, x : [lambda g : g / np.cos(x) **2])
50 np.sinh     = P(np.sinh,     lambda ans, x : [lambda g : g * np.cosh(x)])
51 np.cosh     = P(np.cosh,     lambda ans, x : [lambda g : g * np.sinh(x)])
52 np.tanh     = P(np.tanh,     lambda ans, x : [lambda g : g / np.cosh(x) **2])

```

```

53 np.square = P(np.square, lambda ans, x : [lambda g : g * 2 * x])
54 np.sign    = P(np.sign,   lambda ans, x : [lambda g : 0.0])
55 np.full    = P(np.full,   lambda ans, shape, fill_value : [None, lambda g : np.sum(g)])
56 np.reshape = P(np.reshape, lambda ans, x, shape, order=None : [lambda g : np.reshape(g, x.shape,
order=order)])
57 np.ravel   = P(np.ravel,   lambda ans, x,          order=None : [lambda g : np.reshape(g, x.shape,
order=order)])
58 np.expand_dims = P(np.expand_dims, lambda ans, x, axis          : [lambda g : np.squeeze(g, axis)])

59 np.squeeze = P(np.squeeze, lambda ans, x, axis          : [lambda g : np.repeat(g,
x.shape[axis], axis)])
60 np.repeat  = P(np.repeat, lambda ans, x, shape, axis    : [lambda g : np.sum(g, axis,
keepdims=True)])
61 np.transpose = P(np.transpose, lambda ans, x            : [lambda g : np.transpose(g)])

62 np.split   = P(np.split, lambda ans, A, idxs, axis=0    : [lambda g : np.concatenate(g,
axis=axis)])

63
64 ✓ def make_grad_np_sum(ans, x, axis=None, keepdims=False):
65     if not isinstance(x):
66         return [I]
67     shape = x.shape
68     if axis is None:
69         return [lambda g : np.full(shape, g)]
70     else:
71         if keepdims:
72             return [lambda g : np.repeat(g, shape[axis], axis)]
73         else:
74             return [lambda g : np.repeat(np.expand_dims(g, axis),
75                                         shape[axis], axis)]
76 np.sum = P(np.sum, make_grad_np_sum)
77

```

```

77
78 ✓ def make_grad_np_mean(ans, x, axis=None, keepdims=False):
79     if not isinstance(x, np.ndarray):
80         return [I]
81     shape = x.shape
82     if axis is None:
83         return [lambda g : np.full(shape, g) / np.prod(shape)]
84     else:
85         if keepdims:
86             return [lambda g : np.repeat(g, shape[axis], axis) / shape[axis]]
87         else:
88             return [lambda g : np.repeat(np.expand_dims(g, axis),
89                                     shape[axis], axis) / shape[axis]]
90 np.mean = P(np.mean, make_grad_np_mean)
91
92 ✓ def make_grad_np_max(ans, x):
93     def gradfun(g):
94         idxs = np.argmax(getval(x))
95         return untake(g, np.unravel_index(idxs, x.shape))
96     return [gradfun]
97 np.max = P(np.max, make_grad_np_max)
98
99 ✓ def make_grad_np_dot(ans, A, B):
100 ✓ def grad_np_dot_A(g):
101     if B.ndim is 2:
102         return np.dot(g, B.T)
103     elif A.ndim is 2:
104         return np.outer(g, B)
105     else:
106         return g * B
107 ✓ def grad_np_dot_B(g):
108     if A.ndim is 2:
109         return np.dot(A.T, g)
110     elif B.ndim is 2:
111         return np.outer(A, g)
112     else:
113         return g * A
114     return [grad_np_dot_A, grad_np_dot_B]
115 np.dot = P(np.dot, make_grad_np_dot)
116
117 ✓ def make_grad_np_concatenate(ans, arr_list, axis=0):
118     def grad_np_concatenate(g):
119         idxs = np.cumsum([a.shape[axis] for a in getval(arr_list)[: -1]])
120         return np.split(g, idxs, axis=axis)
121     return [grad_np_concatenate]

```

```
122     np.concatenate = P(np.concatenate, make_grad_np_concatenate)
123
124     # ----- Special list constructor -----
125
126     ✓ class ArgnumGrad(object):
127         def __init__(self, fun_with_argnum):
128             self.fun = fun_with_argnum
129         def __getitem__(self, argnum):
130             return partial(self.fun, argnum)
131
132     def kylist(*args):
133         return list(args)
134     kylist = primitive(kylist, lambda ans, *args : ArgnumGrad(lambda argnum, g : g[argnum]))
135
136     # Wrap the concatenation function to automatically wrap the list into a kylist.
137     unwrapped_np_concatenate = np.concatenate
138     def concatwrapper(*args, **kwargs):
139         args = (kylist(*(args[0])),) + args[1:]
140         return unwrapped_np_concatenate(*args, **kwargs)
141     np.concatenate = concatwrapper
```