Einsum Gradient

April 20, 2025

Last time I talked about how to find the jacobian of einsum. You can use that in backpropogation but you'll run out of RAM very quickly. Today I want to explain how you can backpropogate through einsum operations without needing the full jacobian. Here is the setup: we have neural network $N : \mathbb{R}^{np} \to \mathbb{R}$, and somewhere in that network is an einsum operation that we want to backpropogate across. For simplicity well make it matrix multiplication

$$X_{ij}W_{jk} = Z_{jk} \tag{1}$$

We have the upstream gradient G, which is the same shape as Z, and we want to find the gradients w.r.t. X, W, and then generalize the process to any einsum operation. If we calculate the gradient w.r.t. to a single element of W we see that

$$\left(\nabla N_W\right)_{jk} = \sum_i G_{ik}X_{ij} \tag{2}$$

This is the sum of the product of the k'th column of G and the j'th column of X. Notice that (2) can be written as an einsum

$$X_{ij}G_{ik} = \left(\nabla N_W\right)_{jk} \tag{3}$$

This is similar to (1). We left the subscripts alone, and put the upstream gradient G in place of W. Lets do the same thing to find ∇N_X.

$$\left(\nabla N_X\right)_{ij} = \sum_k G_{ik}W_{jk} \tag{4}$$

$$G_{ik}W_{jk} = \left(\nabla N_X\right)_{ij} \tag{5}$$

For ∇N_X we take the sum of the product of the i'th row of G with the j'th row of W. Comparing (1), (2), (5) we can see the pattern. We leave the subscripts alone and put G in place of whatever we are differentiating with respect to. This rule almost works, but we can easily think of a situation where it fails. Consider

$$X_{ab}W_{cd} = Z_{ab} \tag{6}$$

our rule would give

$$X_{ab}G_{ab} \xrightarrow{??} \left(\nabla N_W\right)_{cd} \tag{7}$$

which doesnt work. There is no way to yield a cd matrix from the operands on the LHS. For another example that wouldnt work Consider

$$X_{bij}W_{jk} = Z_{ik} \tag{8}$$

Here our rule says ∇N_X can be found with $G_{ik}W_{jk} \xrightarrow{??} (\nabla N_X)_{bij}$ but again this operation doesnt make sense since the b dimension is missing from G, W. We notice that this is a *shape* issue and not a *value* issue. Looking at (8) we can see the gradient will be the same across all batches, meaning the gradient we want is some ij matrix repeated across the b dimension. To get it we just need to reshape our the output from our rule . Let A be a placeholder matrix to indicate the output of the einsum. The correct gradient for (8) is

$$1_b \otimes (G_{ik}W_{jk} \to A_{ij}) = (\nabla N_X)_{bij} \tag{9}$$

We can revisit (6), (7) and get the gradient with the right shape via

$$1_{cd} \otimes (X_{ab}G_{ab} \to a) = (\nabla N_W)_{cd} \tag{10}$$

At this point we are done. Backpropogating over an einsum operation can be done in two steps:

1. Take the original operation and replace the target variable with the upstream gradient
2. Reshape/broadcast the result

Putting it into Code

Were going to write a little autograd engine. We need a class for holding data, and then einsum and sigmoid functions.

```
from collections import deque
import numpy as np

class Thing:
    def __init__(self, data, _children=[]) -> None:
        self.data = data if data.shape else data.reshape(1,1)
        self._children = _children
        self._backward = lambda: None
        self.grad = np.zeros_like(self.data)

    def backward(self):
        self.grad, visited, queue = np.ones((1, 1)), set(), deque([self])
        while queue: v = queue.popleft(); [visited.add(v), v._backward(), queue.extend(n for n
```

This holds data/grad. I reshape scalars to $(1, 1)$ because its easier to handle. Calling `backward()` does BFS and calls `._backward()` on all children. Pretty janky looking one-line BFS

Sigmoid

```
def _sigmoid(x):
    out = Thing(1 / (1 + np.exp(-x.data)), _children=[x])
    def _backward():
        x.grad += out.data * (1 - out.data) * out.grad
```

```
    out._backward = _backward
    return out
```

Straightforward. We just need to multiply `out.grad` with sigmoid derivative

Einsum

Little more involved. Basically going to do the original einsum but replace the target variable with `out.grad`. However applying this to (8) directly would yield

```
x_grad = einsum('ik,jk->bij', out.grad, w)
```

This is going to throw an error since `b` is not defined. In a case like this we need to einsum into `->ij` and then repeat along the `b` dimension. I cant explain it well in words but if you read the code it should be easy to understand

```
from einops import repeat

def _einsum(ptrn, x, w):
    out = Thing(np.einsum(ptrn, x.data, w.data), _children=[x, w])
    def _backward():
        x_ptrn, w_ptrn, z_ptrn = *ptrn.split('->')[0].split(','), ptrn.split('->')[-1]
        z_ptrn = z_ptrn if z_ptrn else 'zy'

        w_grad_ptrn = ''.join([c for c in w_ptrn if c in set(x_ptrn + z_ptrn)])
        x_grad_ptrn = ''.join([c for c in x_ptrn if c in set(w_ptrn + z_ptrn)])

        x_grad = np.einsum(f'{z_ptrn},{w_ptrn}->{x_grad_ptrn}', out.grad, w.data)
        w_grad = np.einsum(f'{z_ptrn},{x_ptrn}->{w_grad_ptrn}', out.grad, x.data)

        w_shape = dict(zip(w_ptrn, w.data.shape))
        x_shape = dict(zip(x_ptrn, x.data.shape))

        w_broadcast_string = f"{' '.join(w_grad_ptrn)} -> {' '.join(w_shape.keys())}"
        w_grad = repeat(w_grad, w_broadcast_string, **w_shape)

        x_broadcast_string = f"{' '.join(x_grad_ptrn)} -> {' '.join(x_shape.keys())}"
        x_grad = repeat(x_grad, x_broadcast_string, **x_shape)

        x.grad += x_grad
        w.grad += w_grad

    out._backward = _backward
    return out
```

1. We start by making the `_grad_ptrn`'s for `x, w`. These come from the set of subscripts of the other two operands

2. Next we calc gradient

3. Finally we reshape the gradient by repeating along missing dimensions. In (8), `reshape` would get passed `'i j -> b i j', b =x.shape[0]`

Testing

Lets spin up a random NN and see how it goes. I dont want to code loss functions today so We'll just sum at the end.

```
import torch
from torch import einsum
torch.manual_seed(10)
np.random.seed(10)


x = torch.randn(2, 3, requires_grad = True)
thing = Thing(x.detach().numpy())

shapes = [(3, 4), (4, 5), (1, 2, 5), (5, 3)]
ptrns = ['ij,jk->ik', 'ij,jk->ik', 'ij,cij->ij', 'ab,bd->']

torch_weights = [torch.randn(s, requires_grad = True) for s in shapes]
thing_weights = [Thing(w.detach().numpy()) for w in torch_weights]

for (ptrn, w, thing_w) in zip(ptrns, torch_weights, thing_weights):
    x = torch.sigmoid(einsum(ptrn, x, w))
    thing = _sigmoid(_einsum(ptrn, thing, thing_w))

x.backward()
thing.backward()

for w, thing_w in zip(torch_weights, thing_weights):
    print(np.allclose(w.grad.detach().numpy(), thing_w.grad))
# prints all true
```

Alright thats about it. We can now backpropogate through einsum operations without needing the whole jacobian. This is a lot quicker, and can handle real network architectures without crashing my PC.