Here are my implementations:

CDualNumber operator+(const CDualNumber& d)

{

CDualNumber ret;

ret.m\_real = m\_real + d.m\_real;

ret.m\_dual = m\_dual + d.m\_dual;

return ret;

}

CDualNumber operator-(const CDualNumber& d)

{

CDualNumber ret;

ret.m\_real = m\_real - d.m\_real;

ret.m\_dual = m\_dual - d.m\_dual;

return ret;

}

CDualNumber operator\*(const CDualNumber& d)

{

CDualNumber ret;

ret.m\_real = m\_real \* d.m\_real;

ret.m\_dual = m\_real \* d.m\_dual + m\_dual \* d.m\_real;

return ret;

}

## Does changing gradient step size work the same way with dual numbers as it does for finite differences?

Yes! Modifying gradient step size behaves exactly like it did in finite differences. We don’t have an epsilon anymore, so have one less parameter to tune by hand, which is always nice.

## How would you modify the program if the function F took multiple parameters?

If function F was of the form , you could call function F twice, once for x and once for y.

CDualNumber dualX(x, 1.0f);

float dzdx = F(dualX, y).m\_dual;

CDualNumber dualY(y, 1.0f);

float dzdx = F(x, dualY).m\_dual;

If you step through the code for both of those, you’ll realize that the math for m\_real goes through the same steps for each call, and that it’s own m\_dual that changes each time. To reduce that redundancy, you can make the CDualNumber class have two m\_dual members; one for parameter X and another for parameter Y. When doing operator overloading, you treat the dual numbers equivalently, but distinctly. For instance, here is the multiplication operator:

CDualNumber operator\*(const CDualNumber& d)

{

CDualNumber ret;

ret.m\_real = m\_real \* d.m\_real;

ret.m\_dualX = m\_real \* d.m\_dualX + m\_dualX \* d.m\_real;

ret.m\_dualY = m\_real \* d.m\_dualY + m\_dualY \* d.m\_real;

return ret;

}

This could be further generalized to make m\_dual into an array, where the dual number operations operated in a loop. This is how the MNIST training code works, except it uses a sparse array.

## CHALLENGE: Can you implement other operations?

This is a hard challenge, much respect if you attempted any!

**Division**

The easiest to solve is probably division. If you have this:

The first step is to try and get rid of the epsilon on the bottom.

Thinking about that in a more focused way, what could we multiply by, to make the epsilon disappear? If we get an answer to that, we can multiply the top and bottom by that value, and only have epsilon values on the top, which would let us get our answer.

The answer is that we can multiply by .

When doing FOIL, that makes the “outer” and “inner” cancel out, and leaves us with only the “first” and “last” terms. The “last” term has an epsilon squared which is zero, which only leaves us with “first”, which is a purely real value. You might recognize this as the “complex conjugate” and this being the same method we use to divide complex numbers!

If we multiply the top by the same value, we get:

The full division formula is then:

In summary,

**Square Root**

Let’s also look at square root, .

Taking the derivative of that is pretty easy:

If is a dual number that we are trying to take the square root of, we can treat as a function that we already know the value and derivative of: and .

We are trying to calculate , and we want both the value and the derivative.

The value of the result is just going to be the square root of the value, so we can calculate that for the real component of the result.

The derivative (dual value) of the result can be calculated using the chain rule.

Replacing with the real part of x, and with the dual part of x, and replacing with the formula we calculated earlier, we get:

So, taking the square root of a dual number is…

You can make other operations using the same process of applying the chain rule.

If you just want some code you can copy, you can find more operations here:

<https://blog.demofox.org/2017/03/13/neural-network-gradients-backpropagation-dual-numbers-finite-differences/>