Connections between the Bézout's identity and auto-differentiation

Daniel Coble

September 3, 2023

1 Abstract

I draw a connection between the Euclidean Algorithm and Bézout's identity and backpropagation, specifically stating that Bézout's method for calculating integers x, y such that $ax + by = \gcd(a, b)$, is equivalent to reverse-mode differentiation applied to the Euclidean algorithm. This connection inspires the development of a feed-forward algorithm implementation. Three Python implementations of the Euclidean Algorithm/Bézout's identity are given: a standard one, one using the auto-differentiation engine TensorFlow which I claim is equivalent, and a feedforward algorithm.

2 Introduction

According to Wikipedia [1], the emergence of the backpropagation algorithm for efficient calculation of gradients occurred between the 1960's and 70's and was due to people such as Seppo Linnainmaa, Frank Rosenblatt and Henry J. Kelley. The idea to use the chain rule to calculate derivatives is not hard, and in fact the delay in developing machine learning approaches using backpropagation, which came around in 1986, had more to do with constructing neural networks which could efficiently use backpropagation, than the complexity of the backpropagation approach.

Published in 300 B.C., the Euclidean algorithm has been cited as the oldest nontrivial algorithm which is still in use [2], and in the recent decades has grown in importance because of its use in cryptography. Given two positive integers a and b (It can be generalized to other types of numbers.), the Euclidean algorithm calculates the greatest common divisor, $\gcd(a, b)$. The algorithm works due to the lemma below

Lemma 2.1 (Integer division)**.** *For positive integers a and b, $a \geq b$ there is a unique integer division solution to*

$$a = qb + r \qquad 0 \leq r < b$$

Furthermore,

$$\gcd(a, b) = \gcd(b, r)$$

Since $b < a$ and $r < b$, the lemma reduces the size of the two numbers within the gcd. The Euclidean algorithm is nothing more than repeated division to calculate smaller and smaller numbers, terminating when $\gcd(d, 0)$ is reached, and returning d. Bézout's identity (which was discovered for the case of integers by Méziriac [3]). States the following:

Proposition 1 (Bézout's identity)**.** *For all integers a and b, there exists some integers x and y such that $ax + by = \gcd(a, b)$.*

The proof of this applies back-substitution of the intermediate equations from the Euclidean algorithm. Below I illustrate by an example, calculating $\gcd(314, 159)$ then finding x and y with $314x + 159y = \gcd(314, 159)$.

Euclidean algorithm:

$$\begin{aligned}
314 &= 1 \times 159 + 155 & \gcd(314, 159) &= \gcd(159, 155) && \text{(Step 1)}\\
159 &= 1 \times 155 + 4 & &= \gcd(155, 4) && \text{(Step 2)}\\
155 &= 38 \times 4 + 3 & &= \gcd(4, 3) && \text{(Step 3)}\\
4 &= 1 \times 3 + 1 & &= \gcd(3, 1) && \text{(Step 4)}\\
3 &= 3 \times 1 + 0 & &= \gcd(1, 0) \boxed{= 1} && \text{(Step 5)}
\end{aligned}$$

Bézout reverse step:

$$\begin{aligned}
1 &= 4 - 1 \times 3 && \text{(Step 1)}\\
&= 4 - 1 \times (155 - 38 \times 4) = 39 \times 4 - 1 \times 155 && \text{(Step 2)}\\
&= 39 \times (159 - 1 \times 155) - 1 \times 155 = 39 \times 159 - 40 \times 155 && \text{(Step 3)}\\
&= 39 \times 159 - 40(314 - 1 \times 159) \boxed{= -40 \times 314 + 79 \times 159} && \text{(Step 4)}
\end{aligned}$$

The starting point of the Bézout reverse step is a rearrangement of (Step 4) of the Euclidean algorithm, representing 1 as a linear combination of 4 and 3. Then a rearrangement of (Step 3) of the Euclidean algorithm is to represent 1 as a sum of 4s and 155. This continues up the steps of the Euclidean algorithm until 1 is solved in terms of 159 and 314. A Python implementation of the forward and backwards steps of the Bézout algorithm is given below.

```
'''
A standard implementation of the Euclidean/Bezout algorithm, showing forward-mode and
    reverse-mode steps.
Returns a tuple (d, x, y) with d = gcd(a, b), and ax + by = d.
'''
def euclid_bezout(a, b):
    # Euclidean algorithm/forward-mode step
    C = [a, b]
    Q = []
    while(C[-1] != 0):
        q = C[-2]//C[-1]
        r = C[-2] - q*C[-1]
        Q.append(q)
        C.append(r)
    d = C[-2]
    # Bezout/revese-mode step
    x = 1
    y = -Q[-2]
    for i in range(len(Q)-3, -1, -1):
        x_new = y
        y_new = x - y*Q[i]

        x = x_new
        y = y_new
    return (d, x, y)

euclid_bezout(314, 159)
>>>(1, -40, 79)
```

3 Reverse mode auto-differentiation

The Euclidean algorithm itself is not differentiable since it isn't defined for non-integer numbers. However, the *graph*, or computation sequence, of the Euclidean algorithm evaluated at (a, b) is continuous and differentiable. The data from an application of the Euclidean algorithm can be encoded into two sets $\{c_i\}$ and $\{q_i\}$ with $c_1 = a$ and $c_2 = b$, satisfying

$$c_i = q_i c_{i+1} + c_{i+2}$$

The algorithm then terminates at step step N when $c_{N+2} = 0$ and returns c_{N+1}. Now consider step j, obviously we have

$$c_{j+2} = c_i - q_i c_{i+1}$$

Which is the substitution rule for accumulating higher-order elements in terms of lower-order elements. So if at one level d is written as

$$d = wc_{i+1} + zc_{i+2},$$

substitution yields

$$d = wc_{i+1} + z\left(c_i - q_i c_{i+1}\right) = zc_i + \left(w - zq_i\right)c_{i+1} \tag{3.1}$$

Now we can view this process through backpropagation. Imagine a Euclidean algorithm step with transforms tuples $(c_i, c_{i+1}) \to (\widetilde{c}_{i+1}, \widetilde{c}_{i+2})$. The need to use $\sim$ to differentiate the input and output will be shown after the calculation. The rules for iteration are

$$\begin{cases} \widetilde{c}_{i+1} = c_{i+1} \\ \widetilde{c}_{i+2} = c_i - q_i c_{i+1} \end{cases} \tag{3.2}$$

Now examining the product rule,

$$\begin{cases} \frac{\partial d}{\partial c_i} = \frac{\partial d}{\partial \widetilde{c}_{i+1}}\frac{\partial \widetilde{c}_{i+1}}{\partial c_i} + \frac{\partial d}{\partial \widetilde{c}_{i+2}}\frac{\partial \widetilde{c}_{i+2}}{\partial c_i} \\ \frac{\partial d}{\partial c_{i+1}} = \frac{\partial d}{\partial \widetilde{c}_{i+1}}\frac{\partial \widetilde{c}_{i+1}}{\partial c_{i+1}} + \frac{\partial d}{\partial \widetilde{c}_{i+2}}\frac{\partial \widetilde{c}_{i+2}}{\partial c_{i+1}} \end{cases}$$

Starting with the initial data as

$$\begin{cases} \frac{\partial d}{\partial \widetilde{c}_{i+1}} = w \\ \frac{\partial d}{\partial \widetilde{c}_{i+2}} = z \end{cases}$$

We get

$$\begin{cases} \frac{\partial d}{\partial c_i} = w\frac{\partial \widetilde{c}_{i+1}}{\partial c_i} + z\frac{\partial \widetilde{c}_{i+2}}{\partial c_i} \\ \frac{\partial d}{\partial c_{i+1}} = w\frac{\partial \widetilde{c}_{i+1}}{\partial c_{i+1}} + z\frac{\partial \widetilde{c}_{i+2}}{\partial c_{i+1}} \end{cases}$$

Then taking partial derivatives from the iteration rule (3.2),

$$\begin{cases} \frac{\partial d}{\partial c_i} = z \\ \frac{\partial d}{\partial c_{i+1}} = w - zq_i \end{cases} \tag{3.3}$$

And these are the same rules as was shown in (3.1). Notice that $\frac{\partial d}{\partial c_{i+1}} \neq \frac{\partial d}{\partial \widetilde{c}_{i+1}}$, showing the need to differentiate the input and output spaces.

An implementation of the Euclidean algorithm using the automatic differentiation engine TensorFlow is given below. As with a neural network, only the forward pass needs to be explicitly constructed and the reverse section is calculated automatically by the engine.

```
import tensorflow as tf
import numpy as np

'''
A TensorFlow implementation of the Euclidean algorithm. Backpropagation produces the Bezout
    values x and y.
Returns a tuple (d, x, y) with d = gcd(a, b) and ax + by = d
'''
def tensorflow_bezout(a, b):
    a = tf.Variable(a, dtype=tf.float32)
    b = tf.Variable(b, dtype=tf.float32)
```

```
    with tf.GradientTape() as tape:
        tape.watch([a, b])
        c1 = a
        c2 = b
        while(c2 > 0):
            q = c1 // c2
            c1_new = c2
            c2_new = c1 - q*c2

            c1 = c1_new
            c2 = c2_new
        d = c1
    [x, y] = tape.gradient(d, [a, b])
    return (int(d.numpy()), int(x.numpy()), int(y.numpy()))

tensorflow_bezout(314, 159)
>>>((1, -40, 79)
```

4 Forward mode auto-differentiation

Having shown that the values x and y are equal to $\frac{\partial d}{\partial a}$ and $\frac{\partial d}{\partial b}$, respectively, we know any method which calculates those values will produce solutions to Bézout's identity. Forward mode auto-differentiation works on a completely different principle than reverse-mode autodifferentation. For the case of tracking two variables, numbers are represented as tuples of the form

$$\mathbf{c} = \left(c, \frac{\partial c}{\partial a}, \frac{\partial c}{\partial b}\right)$$

Tuples obey the rules of a field, with addition and multiplication defined using the respective rules for combining derivatives.

$$\mathbf{c_1} + \mathbf{c_2} = \left(c_1 + c_2, \frac{\partial c_1}{\partial a} + \frac{\partial c_2}{\partial a}, \frac{\partial c_1}{\partial b} + \frac{\partial c_2}{\partial b}\right)$$

$$\mathbf{c_1} * \mathbf{c_2} = \left(c_1 * c_2, c_1 * \frac{\partial c_2}{\partial a} + c_2 * \frac{\partial c_1}{\partial a}, c_1 * \frac{\partial c_2}{\partial b} + c_2 * \frac{\partial c_1}{\partial b}\right)$$

Since it only contains the Euclidean algorithm, this code looks superficially like the code shown in section 3. However the code in that section does execute the reverse procedure in the back-end of the auto-differentiation engine, while this code carries all calculations in the forward direction. That also means that while the standard reverse-mode algorithm has memory requirements bounded by the size of the inputs, the forward mode algorithm has constant memory requirements. An implementation of forward-mode calculation is given below.

```
'''
A class which represents tuple numbers (c, dc/da, dc/db)
'''
class TupleNum():

    def __init__(self, num):
        self.num = num

    def __mul__(self, other):
        c = self.num[0]*other.num[0]
        dcda = self.num[1]*other.num[0] + self.num[0]*other.num[1]
        dcdb = self.num[2]*other.num[0] + self.num[0]*other.num[2]
        return TupleNum((c, dcda, dcdb))

    def __add__(self, other):
        c = self.num[0] + other.num[0]
        dcda = self.num[1] + other.num[1]
        dcdb = self.num[2] + other.num[2]
        return TupleNum((c, dcda, dcdb))
```

```
20
21      def __sub__(self, other):
22          c = self.num[0] - other.num[0]
23          dcda = self.num[1] - other.num[1]
24          dcdb = self.num[2] - other.num[2]
25          return TupleNum((c, dcda, dcdb))
26
27      def __floordiv__(self, other):
28          c = self.num[0]//other.num[0]
29          return TupleNum((c, 0, 0))
30
31  '''
32  This forward implementation of Euclidean algorithm doesn't contain a reverse Bezout step,
        but produces the values via forward mode autodifferentation.
33  Returns a tuple (d, x, y) with d = gcd(a, b) and ax + by = d
34  '''
35  def forward_bezout(a, b):
36      c1 = TupleNum((a, 1, 0))
37      c2 = TupleNum((b, 0, 1))
38      while(c2.num[0] > 0):
39          q = c1 // c2
40          c1_new = c2
41          c2_new = c1 - q*c2
42
43          c1 = c1_new
44          c2 = c2_new
45      d = c1
46      return d.num
47
48  forward_bezout(314, 159)
49  >>>(1, -40, 79)
```

Careful examination of this method could produce another proof of Bézout's identity, but I don't do that here.

5 Notebook

See working examples of the three implementations at https://github.com/dncoble/Notebooks/blob/main/Euclidean%20Algorithm%20and%20Auto-differentation/Euclidean%20Algorithm%20and%20Auto-differentiation.ipynb.

References

[1] https://en.wikipedia.org/wiki/Backpropagation.

[2] Knuth, D. E., "The art of computer programming, volume 2 seminumerical algorithms," (1997).

[3] https://en.wikipedia.org/wiki/B%C3%A9zout%27s_identity.