Building Neural Networks from Scratch

Part 1

Luca WB

2026-01-08

Table of contents

0.1 Brief summary

The objective from this page to undestand how to implement a neural network from scratch without any external libraries, this page consider that you already have some knwlodge of Artificial Neural Networks. The main reason for this is just to make more compreensible the black box of Neural Networks. So, to this project, the main resource (but not the unique) are the series of videos from Andrej Karpathy. In this first part, I will cover how to implement AutoGrad for do backpropagation. At the end of this post, yo will cable of create MLPs without any external library.

0.2 Basic knowledge of derivative

So, from the start, to make sense at how an NN train and learn something, you first need a very good understanding around the meaning of derivative operations. A derivative is an operation that gives us a formula that describes the slope of a function as it modifies a variable, but for out prupose, we will only work with functions that generate linear derivatives. Thus, for the function $ f(x) = 3x^2 - 4x + 5 $, see the graph bellow.

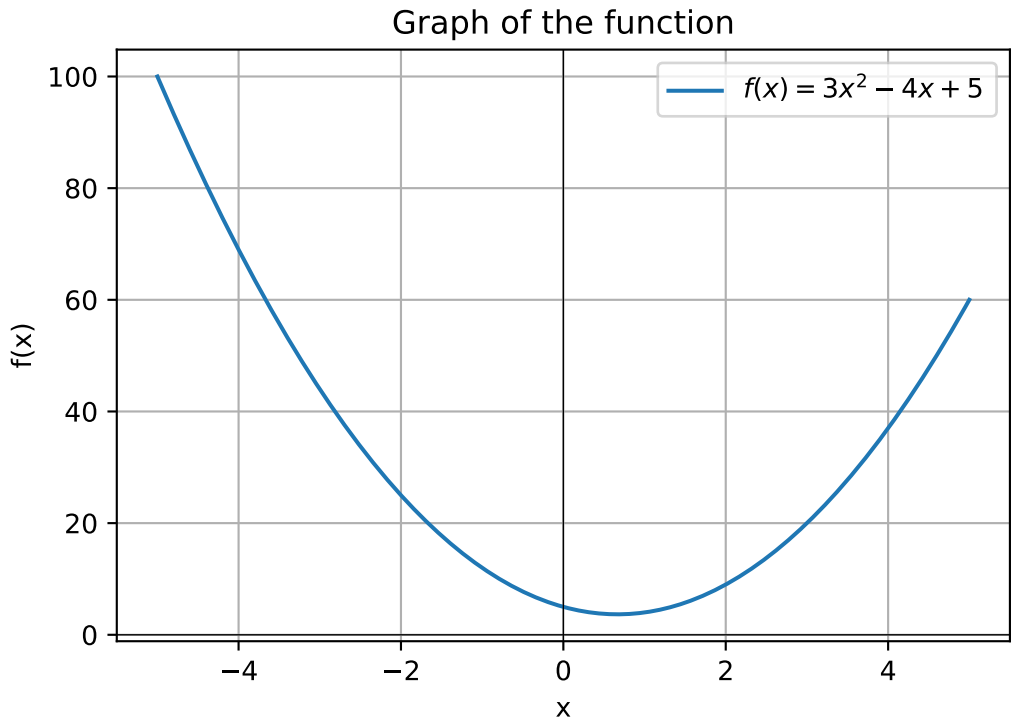
```
import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
    return 3*x**2 - 4*x + 5

# Generate x values
x = np.linspace(-5, 5, 400)
y = f(x)

# Plot
```

```
plt.figure(figsize=(6, 4))
plt.plot(x, y, label=r"$f(x) = 3x^2 - 4x + 5$")
plt.axhline(0, color="black", linewidth=0.5)
plt.axvline(0, color="black", linewidth=0.5)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Graph of the function")
plt.legend()
plt.grid(True)
plt.show()
```



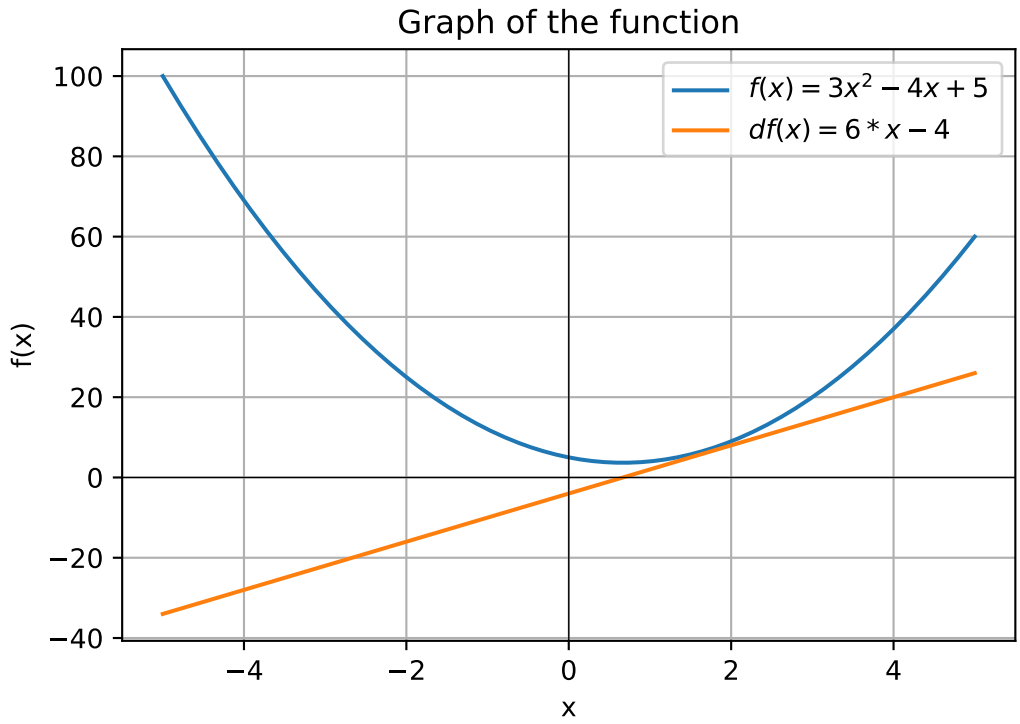
This function is easy to undestand and derivate analytic, the derivate is $\frac{df(x)}{dx} = 6x - 4$. Plotting both function and his derivate, we get the graphic bellow.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
    return 3*x**2 - 4*x + 5

def df(x):
    return 6*x - 4
```

```
10
11  # Generate x values
12  x = np.linspace(-5, 5, 400)
13  y = f(x)
14  y2 = df(x)
15
16  # Plot
17  plt.figure(figsize=(6, 4))
18  plt.plot(x, y, label=r"$f(x) = 3x^2 - 4x + 5$")
19  plt.plot(x, y2, label=r"$df(x) = 6*x - 4$")
20  plt.axhline(0, color="black", linewidth=0.5)
21  plt.axvline(0, color="black", linewidth=0.5)
22  plt.xlabel("x")
23  plt.ylabel("f(x)")
24  plt.title("Graph of the function")
25  plt.legend()
26  plt.grid(True)
27  plt.show()
```



The basic idea is that with we want to minimize the value of a function (main objective in deep learning), we just need to see the value of a derivative in some point A, that give us all the information we need to go to the minimum spot. Just do some numerical exemple, in the graph above, note that the minimum spot is in som evalue around 0 and 2, more close to 0 (precisaly 2/3), so, just pick some rondom number, like -2, the value of $f(x)$ with -2

is 25, and the derivative is -16. The number -16 represent the rate at which the function varies for each increase in the value of x at that point in specific. So, with we increse the value of X a little bit, we can lower the value of X, so probably, the $f(-1.999)$ give us a lower value than $f(-2)$

```
def f(x):
    return 3*x**2 - 4*x + 5

print("f(-2)=",f(-2))
print("f(-1.999)=",f(-1.999))
```

```
f(-2)= 25
f(-1.999)= 24.984003
```

This is the general idea of how we can minimize some function, that is also the main idea of how gradient descendent works.

0.3 How to estimate gradients

In general, to make an framework for working with nn, its just an AutoGrad (a tool that can do diferatiation automatically) and some fancy stuff for make more pratical.

For start, lets make things the most simple for now. Our goal its make an class that can calc for us all the gradietns (its the same as an derivative) from the function $L = -2 \cdot ((2 \cdot 3) + 10)$. But we dont are confortable derivate something with just numbers, so lets consider in this way the function:

```
a = 2
b = -3.0
c = 10
f = -2
e = a*b
d = e + c
L = d * f
L
```

```
-8.0
```

Just to make clear, the knowledge that we want with this, is ow much changes in the final result, increase the values of any of the variables a little To get the gradients, we can use an aproximate that consists in adding a very small number h is all of the values, than subtract the new with the original, and divide by the h. The code bellow shows how to do this with the variable a:

```python
a = 2
b = -3.0
c = 10
f = -2
e = a*b
d = e + c
L = d * f

h = 0.0001
a = 2 + h
b = -3.0
c = 10
f = -2
e = a*b
d = e + c
L2 = d * f

print(f"L(2) = {L}")
print(f"L({a}) = {L2}")
print(f"The slope/gradient: {(L2 - L)/h}")
```

```
L(2) = -8.0
L(2.0001) = -7.999399999999998
The slope/gradient: 6.000000000021544
```

We will not use this method to create our autograd, but we can use this to verify with our gradients are right.

0.4 Lets start making an AutoGrad

The basic idea of the AutoGrad we will make is make some very simple nodes, that represent the number in our calculation and will track all the last two nodes that make him. Basically, in $L = -2 \cdot ((2 \cdot 3) + 10)$, we will consider that a node can only save on number, like in the code represantation

```python
a = 2
b = -3.0
c = 10
f = -2
e = a*b
d = e + c
L = d * f
```

So, for start, lets make the basic of our class:

```
class Value:
    def __init__(self, data, _children=(), _op=""):
        self.data = data
        self.grad = 0 # All nodes will start with no frad, becouse we dont know what is th
        self._prev = set(_children) # Dont worry about this for know, we only use set for
        self._op = _op

    # This is just for us visualize our class
    def __repr__(self):
        return f"Value(data={self.data})"
```

So with this class, we can create some Value's, but we cant use them for anything, so lets make some operations

```
class Value:
    def __init__(self, data, _children=(), _op=""):
        self.data = data
        self.grad = 0
        self._prev = set(_children)
        self._op = _op

    # This is just for us visualize our class
    def __repr__(self):
        return f"Value(data={self.data})"

    # This make that we can add to Value's
    def __add__(self, other):
        out =  Value(self.data + other.data, (self,other), '+')
        return out

    def __mul__(self, other):
        out = Value(self.data * other.data, (self,other), "*")
        return out
```

And with this simples class, we can know calc our formula (not the gradients yet)

```
a = Value(2.)
b = Value(-3.0)
c = Value(10.)
f = Value(-2)
e = a * b
d = e + c
L = d * f
L
```

```
Value(data=-8.0)
```

Bellow, a just make an graph to visualize the operations in an graph, The question is, how we can get the gradients from d and f?

```python
from graphviz import Digraph

def trace(root):
  # builds a set of all nodes and edges in a graph
  nodes, edges = set(), set()
  def build(v):
    if v not in nodes:
      nodes.add(v)
      for child in v._prev:
        edges.add((child, v))
        build(child)
  build(root)
  return nodes, edges

def draw_dot(root):
  dot = Digraph(format='svg', graph_attr={'rankdir': 'LR'}) # LR = left to right

  nodes, edges = trace(root)
  for n in nodes:
    uid = str(id(n))
    # for any value in the graph, create a rectangular ('record') node for it
    dot.node(name = uid, label = "{ data %.4f | grad %.4f }" % (n.data, n.grad), shape='re
    if n._op:
      # if this value is a result of some operation, create an op node for it
      dot.node(name = uid + n._op, label = n._op)
      # and connect this node to it
      dot.edge(uid + n._op, uid)

  for n1, n2 in edges:
    # connect n1 to the op node of n2
    dot.edge(str(id(n1)), str(id(n2)) + n2._op)

  return dot

draw_dot(L)
```