Back-propagation Lecture Notes

Diego Bellani

30 September 2024

1 Inroduction

The task performed by a neural network is determined by the values of its weights. To find the weight needed to perform a certain task we use a series of examples (the training data,) together with a loss function that determines how far we are from performing said task on the given examples. In order to get our network closer to perform the task we want on the examples we use a *gradient* descent algorithm.

We need a way to calculate the gradient of the loss of the neural network on the example data w.r.t. the weights of the network.

In the context of neural networks it is often the case that we modify the model itself to make the task learnable with the available data. This is takes a lot of trial and error, therefore we need an automated way to calculate said gradient.

A possible way to do so is to is *numerical differentiation*. That is, in its most basic form, to approximate the gradient using the definition of the derivative, i.e.

$$\lim_{h\to 0}\frac{f(x+h)-f(x)}{h}.$$

Let us try to see if this is a viable option with an example, let

$$f(x_1,x_2)=\log(x_1)+x_1x_2-\sin(x_2) \tag{1}$$

and say that we want to calculate $\nabla f(2,5)$. We now want to confront the value that we would obtain from the numerical approximation against the formula that we would find by hand. Since $\nabla f(2,5)$ has two elements, let us start from the second. We know from basic calculus that $\frac{\partial f(x_1,x_2)}{\partial x_2}=x_1-\cos(x2)$. Now to use the limit definition in our numerical approximation, since computers cannot take limits, we are going to use a small number instead for h. Let us write a quick implementation below.

```
import numpy
def f(x1,x2): return numpy.log(x1) + x1*x2 - numpy.sin(x2)
x1, x2, h = numpy.array([2.,5.,1e-5], dtype=numpy.float32)
d_f_x2 = (f(x1,x2+h)-f(x1,x2))/h
print(numpy.absolute(x1-numpy.cos(x2) - d_f_x2))
```

The code above prints `0.09509146`, this means that the numerical approximation, in this simple example, is already off by one decimal point. This is not acceptable, not to mention that to find a gradient of n elements it requires $O(n)$ evaluations of the function which may be very expensive.

Another way to automate the calculation of the gradient is to use *symbolic differentiation*. That is to treat the expression of the loss as an data structure[1] and write an algorithm that applies the rules of calculus for us. Some people may tell you that this approach does not scale due to *expression swell*, that is to say that the naïve application of calculus rules can lead to an explosion in the size of the expression and in a lot of redundant calculations. E.g $\frac{\partial \sin(e^x)}{\partial x} = e^x \cos(e^x)$, as you can see in a naïve implementation we have to evaluate and store the node representing e^x two times. This problem can be solved with *common sub-expression elimination*. The real problem lies in the fact that we have to use another language to specify our neural network and in how to differentiate through control-flow[2].

In the rest of this document we are going to introduce the *back-propagation* algorithm which is a special case of *reverse-mode automatic differentiation*, which is one of the differentiation modes studied by the field of *automatic differentiation* [1, 2].

2 Scalar Back-propagation

To recap, we need a an efficient way to calculate gradient, at *machine precision*, of a function written in a programming language, supporting all of its control-flow constructs. This task may seem daunting but we will show how it can be done, starting from the general idea, then describing how it is usually implemented and how this gives us for free support for deriving through control-flow.

Let us start by considering equation 1, in particular how we can use the chain rule to calculate the two elements of its gradient. But first a bit of notation, we are going to decompose equation 1 in its elementary operations as shown of the left side of figure 2. We can then express the two elements of the gradient as

$$\begin{aligned}
\frac{\partial f(x_1, x_2)}{\partial x_1} &= \frac{\partial v_5}{\partial v_4}\left(\frac{\partial v_4}{\partial v_1}\frac{\partial v_1}{\partial v_{-1}} + \frac{\partial v_4}{\partial v_2}\frac{\partial v_2}{\partial v_{-1}}\right) \\
&= \frac{\partial v_5}{\partial v_4}\frac{\partial v_4}{\partial v_1}\frac{\partial v_1}{\partial v_{-1}} + \frac{\partial v_5}{\partial v_4}\frac{\partial v_4}{\partial v_2}\frac{\partial v_2}{\partial v_{-1}} \\
\frac{\partial f(x_1, x_2)}{\partial x_2} &= \frac{\partial v_5}{\partial v_3}\frac{\partial v_3}{\partial v_0} + \frac{\partial v_5}{\partial v_4}\frac{\partial v_4}{\partial v_2}\frac{\partial v_2}{\partial v_0}
\end{aligned}$$

[1] A directed acyclic graph.

[2] Again this is not strictly true. We could use our own programming language as the language to specify the expression that we want to differentiate and than use *reflection* to transform it in a suitable data structure. With this approach we can also make control-flow structures work but if you get this far you have to make some philosophy about what you mean by symbolic differentiation... [3].

$$
\begin{aligned}
v_{-1} &= x_1 \\
v_0 &= x_2 \\
v_1 &= \log(v_{-1}) \\
v_2 &= v_{-1} v_0 \\
v_3 &= \sin(v_0) \\
v_4 &= v_1 + v_2 \\
v_5 &= v_4 - v_3 \\
f(x_1, x_2) &= v_5,
\end{aligned}
\qquad
\begin{aligned}
\bar{v}_5 &= 1 \\
\bar{v}_4 &= \bar{v}_5 \tfrac{\partial v_5}{\partial v_4} \\
\bar{v}_3 &= \bar{v}_5 \tfrac{\partial v_5}{\partial v_3} \\
\bar{v}_1 &= \bar{v}_4 \tfrac{\partial v_4}{\partial v_1} \\
\bar{v}_2 &= \bar{v}_4 \tfrac{\partial v_4}{\partial v_2} \\
\bar{v}_0 &= \bar{v}_3 \tfrac{\partial v_3}{\partial v_0} \\
\bar{v}_{-1} &= \bar{v}_2 \tfrac{\partial v_2}{\partial v_{-1}} \\
\bar{v}_0 &\mathrel{+}= \bar{v}_2 \tfrac{\partial v_2}{\partial v_0} \\
\bar{v}_{-1} &\mathrel{+}= \bar{v}_1 \tfrac{\partial v_1}{\partial v_{-1}}
\end{aligned}
$$

Figure 1: Left forward propagation. Right back-propagation of the gradient.

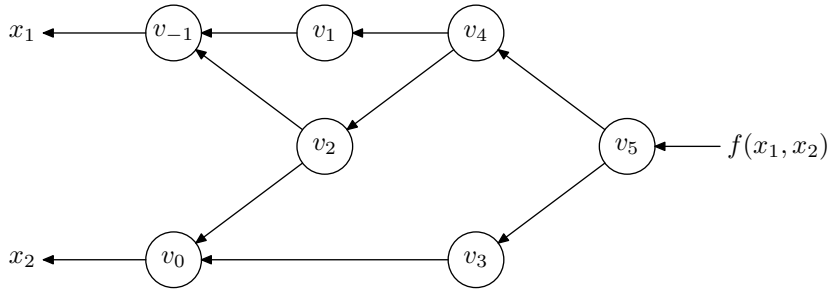

Figure 2: Graph representing the expression.

and in figure 2 we have the dependencies among the variables used to evaluate the expression.

We can start by noting a few things. The paths in the graph from the source node v_5 to the sinks v_{-1} and v_0 correspond to the addends in the chain rule, e.g. $\frac{\partial v_5}{\partial v_3}\frac{\partial v_3}{\partial v_0}$ correspond to the path $((v_5, v_3), (v_3, v_0))$. We also have that we can evaluate a path (or addend of a chain rule), from left to right, e.g.

$$\frac{\partial v_5}{\partial v_4}\frac{\partial v_4}{\partial v_1}\frac{\partial v_1}{\partial v_{-1}} = \frac{\partial v_5}{\partial v_1}\frac{\partial v_1}{\partial v_{-1}} = \frac{\partial v_5}{\partial v_{-1}},$$

that is we can evaluate it by left-multiplying the derivatives of v_5 w.r.t. every intermediate variable v_i along the path.

This are the foundamental ideas behing the back-propagation algorithm. We can calculate the gradient incrementally, starting at the source node and going backward, storing at each node v_i its derivative with respect to the source $\frac{\partial v_5}{\partial v_i}$. If we let $\bar{v}_i = \frac{\partial v_5}{\partial v_i}$ we can see how this calculation is performed on the right side of figure 2.

The way in which this is usually implemented in a programming language is by keeping track of all the variables and operations used on them so that at the end of the computation of the function we have the graph that we can use to back-propagate the gradient through. In the case of Python this is done by creating a class like the one below.

```
import dataclasses, enum
class Operation(enum.Enum): ...
@dataclasses.dataclass
class Scalar:
    value: float
    gradient: float
    operation: Operation
    parent1: Scalar
    parent2: Scalar
```

To make the generation of the graph more ergonomic one can implement the dunder methods to use the standard mathematical operation on `Scalar` objects. It is important that before the back-propagation step the graph is topologically sorted.

We will illustrate the way in which this enables us to back-propagate through contol-flow with an example: when you train a neural network you update its weight in a loop, so if your network is represented by a static expression the graph that is used to back propagate has always the same form[3], if in the code that we use to specify the neural network we have an `if` that take one of its branches based on a calculated value the graph that is going to be generated is going to differ from one of the iteration of the loop to another. E.g. `x = a+b if a > 0 else a+c`. The back-propagation algorithm is therefore going to propagate the gradient though a different graph every time but it does not care about how it was generated, if by an `if`, a `for` or an exception.

3 Back-propagation for Vectors and Matrices

We now have an algorithm that it is capable of differentiating any code, saddly this is not enough for our needs... There are two main problems with this approach. We are now going to illustrate them first for the case in which the gradient is back-propagated through vector operations then for the more general case of matrix operations.

The first one is a memory problem, to train neural networks we often do computations with big matrices, multiplying two matrices of size $n \times n$ requires at leats $\Omega(n^2)$ mathematical operations[4]. If you consider that a single precision floating point number occupies 4 bytes, and that to keep track of an operation

[3] Of course the values stored in it are going to change.

[4] The naive matrix multiplication algorithm requires $O(n^3)$ operations, there are better ones but they cannot require less than $\Omega(n^2)$, so we are going to make a very rosey assumption here.

you are going to need at least 16 bytes[5] so there are 8 bytes of overhead per operation so if you les $n = 1024$ we would be $8 \cdot 1024^2$, i.e. 8MiB, of overhead for just a single operation.

The second problem is one of performance, if we the values of our operations all arround in memory it not possible to use optimized BLAS routines to implement the linear algebra operations we care about, also due to how computer works it is not really possible to implement them ourselves. Since what we care about are mostly matrix operations it makes sense to perform operations directly on vectors (and matrices) and not perform them at the granular scalar level, we will need to add a few more bytes to keep track of the dimension of the vector/matrix but it is going to be worth it.

Doing this introduces another problem, to see what is is let us start with a quick refresh of multivariable calculus let $\mathbf{y} = \mathbf{f}(\mathbf{x})$, where $\mathbf{f}$ is a function of vector argument and value. Its derivative is a matrix called the Jacobian with the following contents

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}.$$

So back propagating the gradient now requires multiplying the gradient vector $\mathbf{g}$ with the Jacobian at that node.

The problem is going to become evident with the following example, let $\mathbf{f}(\mathbf{x}) = \mathbf{a} \odot \mathbf{x}$, where $\odot$ is the element-wise roduct. Its Jacobian is

$$J = \begin{bmatrix} a_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & a_n \end{bmatrix},$$

that is a diagonal matrix with $\mathbf{a}$ as its diagonal. Multiplying the gradient with this matrix is the same as doing an element-wise multiplication, that is

$$J\mathbf{g} = \mathbf{a} \odot \mathbf{g}.$$

This is because J is *sparse*, i.e. most of its elements are 0. We can therefore implement matrix vector multiplication in a more space and time efficient way by just storing $\mathbf{a}$ instead of J. This is called Jacobian-vector product (vJp) and can often be calculated implicitly in an efficient way.

It is important to note that not all Jacobians are sparse, if we let $\mathbf{f}(\mathbf{x}) = A\mathbf{x}$, where A is a matrix, then $\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = A'$, where A' is A transposed.

So the Jacobians of many function is sparse and we can make some space and time savings by being smart in the way in wich we store and operate on this sparse Jacobians.

[5] For a binary operation say we allocate 4 bytes for pointers to the 2 parents, 4 bytes for the value of the operation and another 4 for its gradient, ignoring the fact that we also need a way to distinguish a among the various kinds of operations.

Neural networks are often trained in batches so the multiplications performed with matrices are not in the form of a matrix and a vector but of a matrix and a matrix and a matrix. Let $F(X) = AX$, what is $\frac{\partial F(X)}{\partial X}$?

The answare is: its debated [4]. Why is this important? Matrix multiplication is arguably the most important operation in neural networks, therefore it is very important to be able to back propagate through it. Where is the problem, whatever $\frac{\partial F(X)}{\partial X}$ is it needs to contain al possible partial derivatives of the input w.r.t. the output, for simplicity say that the input and and the output have the same shape and are both square so $\frac{\partial F(X)}{\partial X}$ needs to contain $n \times n \times n \times n$ numbers in it, now if $n = 1024$ then $\frac{\partial F(X)}{\partial X}$, for single precision floatin point numbers, needs to occupy at least 4TiB, which as of today no GPU on earth can store. Luckly it is sparse, we just need a way to implicitly perform the vJp.

We are going to need a few pieces, first let $\langle \mathbf{x}, \mathbf{y} \rangle$ be the inner product of the vectors $\mathbf{x}$ and $\mathbf{y}$. It can be proven that $\frac{\partial \langle \mathbf{f}(\mathbf{x}), \mathbf{g}(\mathbf{x}) \rangle}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} \mathbf{g}(\mathbf{x}) + \frac{\partial \mathbf{g}(\mathbf{x})}{\partial \mathbf{x}} \mathbf{f}(\mathbf{x})$, so if we keep one of the two arguments costant deriving the inner product gives us the vJp. E.g. $\frac{\partial \langle \mathbf{a} \odot \mathbf{x}, \mathbf{g} \rangle}{\partial \mathbf{x}} = \mathbf{a} \odot \mathbf{g}$.

Let $\mathbf{vec}(A)$ be a vector made by stacking the columns of the matrix A one on top of the other. In this way we can generalize our our defition of inner product to matrices by letting $\langle A, B \rangle = \langle \mathbf{vec}(A), \mathbf{vec}(B) \rangle$. Let $\mathrm{tr}(A)$ be the sum of the diagonal entries of a square matrix, we have that $\langle A, B \rangle = \mathrm{tr}(A'B)$. Finally it can be prooven that $\frac{\partial \mathrm{tr}(X'A)}{\partial X} = A$ [6, 5].

We are now ready to find the vJp of the matrix-matrix multiplication AX, let G be the matrix containing the gradients of the batch in its columns, then:

$$\frac{\partial \langle AX, G \rangle}{\partial X} = \frac{\partial \mathrm{tr}((AX)'G)}{\partial X} = \frac{\partial \mathrm{tr}(X'A'G)}{\partial X} = A'G.$$

References

[1] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, January 2017.

[2] Andreas Griewank and Andrea Walther. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, second edition, 2008.

[3] Sören Laue. On the equivalence of forward mode automatic differentiation and symbolic differentiation. *CoRR*, abs/1904.02990, 2019.

[4] Jan R. Magnus. On the concept of matrix derivative. *Journal of Multivariate Analysis*, 101(9):2200–2206, 2010.

[5] Jan R. Magnus and Heinz Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics*.

[6] K. B. Petersen and M. S. Pedersen. The matrix cookbook, nov 2012. Version 20121115.