Computer Vision

Lecture 05
Backpropagation and Computational Graphs
Military College of Signals

Asim D. Bakhshi
asim.dilawar@mcs.edu.pk

April 29, 2022

Contents

1 Setting Up the Problem

In this section we will develop expertise with an intuitive understanding of backpropagation [1], which is a way of computing gradients of expressions through recursive application of **chain rule**. Understanding of this process and its subtleties is critical for you to understand, and effectively develop, design and debug neural networks.

1.1 Problem Statement

Here is the core problem: We are given some function $f(\mathbf{x})$ where $\mathbf{x}$ is a vector of inputs and we are interested in computing the gradient of f at $\mathbf{x}$ (i.e. $\nabla f(\mathbf{x})$)

1.2 Motivation

Recall that the primary reason we are interested in this problem is that f corresponds to the loss function (L) and the inputs $\mathbf{x}$ will consist of the training data and the weights. For example, the loss could be the SVM loss function and the inputs are both the training data (x_i,y_i),$i = 1 \cdots N$ and the weights and biases $\mathbf{W}$, b. Note that (as is usually the case in Machine Learning) we think of the training data as given and fixed, and of the weights as variables we have control over. Hence, even though we can easily use backpropagation to compute the gradient on the input examples x_i, in practice we usually only compute the gradient for the parameters (e.g. $\mathbf{W}$, b) so that we can use it to perform a parameter update. However, as we will see later in the class the gradient on x_i can still be useful sometimes, for example for purposes of visualization and interpreting what the Neural Network might be doing.

There are various ways in which backpropagation is viewed and developed for study. We adopt a view of backpropagation as backward flow in real-valued circuits. The insights you'll gain may help you throughout the class.

2 Gradients

2.1 Basic Interpretation of Gradient

Consider a simple multiplication function of two numbers,

$$f(x, y) = xy \tag{1}$$

We can consider x and y as inputs to the function f and it is a matter of simple calculus to derive the partial derivative for either input:

$$f(x, y) = xy \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$$

Keep in mind what the derivatives tell you: They indicate the rate of change of a function with respect to that variable surrounding an infinitesimally small region near a particular point:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

A technical note is that the division sign on the left-hand side is, unlike the division sign on the right-hand side, not a division. Instead, this notation indicates that the operator $\frac{d}{dx}$ is being applied to the function f, and returns a different function (the derivative). A nice way to think about the expression above is that when h is very small, then the function is well-approximated by a straight line, and the derivative is its slope. In other words, the derivative on each variable tells you the sensitivity of the whole expression on its value.

Example. For example, if $x = 4$, $y = -3$ then $f(x, y) = -12$ and the derivative on x $\frac{\partial f}{\partial x} = -3$. This tells us that if we were to increase the value of this variable by a tiny amount, the effect on the whole expression would be to decrease it (due to the negative sign), and by three times that amount. This can be seen by rearranging the above equation ($f(x+h) = f(x) + h\frac{df(x)}{dx}$). Analogously, since $\frac{\partial f}{\partial x} = 4$, we expect that increasing the value of y by some very small amount h would also increase the output of the function (due to the positive sign), and by $4h$.

As mentioned, the gradient ∇f is the vector of partial derivatives, so we have that $\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}] = [y, x]$. Even though the gradient is technically a vector, we will often use terms such as "the gradient on x" instead of the technically correct phrase "the partial derivative on x" for simplicity.

We can also derive the derivatives for the addition operation:

$$f(x, y) = x + y \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = 1 \qquad \frac{\partial f}{\partial y} = 1$$

that is, the derivative on both x, y is 1 regardless of what the values of x, y are. This makes sense, since increasing either x, y would increase the output of f, and the rate of that increase would be independent of what the actual values of x, y are (unlike the case of multiplication above). The last function we'll use quite a bit in the class is the max operation:

$$f(x, y) = \max(x, y) \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = \mathbf{1}(x >= y) \qquad \frac{\partial f}{\partial y} = \mathbf{1}(y >= x)$$

That is, the (sub)gradient is 1 on the input that was larger and 0 on the other input. Intuitively, if the inputs are $x = 4, y = 2$, then the max is 4, and the function is not sensitive to the setting of y. That is, if we were to increase it by a tiny amount h, the function would keep outputting 4, and therefore the gradient is zero: there is no effect. Of course, if we were to change y by a large amount (e.g. larger than 2), then the value of f would change, but the derivatives tell us nothing about the effect of such large changes on the inputs of a function; They are only informative for tiny, infinitesimally small changes on the inputs, as indicated by the $\lim_{h \to 0}$ in its definition.

2.2 Compound Gradient Expressions: Chain Rule

Lets now start to consider more complicated expressions that involve multiple composed functions, such as

$$f(x, y, z) = (x + y)z \tag{2}$$

This expression is still simple enough to differentiate directly, but we'll take a particular approach to it that will be helpful with understanding the intuition behind backpropagation.

In particular, note that this expression can be broken down into two expressions: $q = x + y$ and $f = qz$. Moreover, we know how to compute the derivatives of both expressions separately, as seen in the previous section. f is just multiplication of q and z, so $\frac{\partial f}{\partial q} = z$, $\frac{\partial f}{\partial z} = q$, and q is addition of x and y so $\frac{\partial q}{\partial x} = 1$, $\frac{\partial q}{\partial y} = 1$.

However, we don't necessarily care about the gradient on the intermediate value q - the value of $\frac{\partial f}{\partial q}$ is not useful. Instead, we are ultimately interested in the gradient of f with respect to its inputs x, y, z. The **chain rule** tells us that the correct way to "chain" these gradient expressions together is through multiplication. For example, $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$. In practice this is simply a multiplication of the two numbers that hold the two gradients.

Example Implementation: You can implement forward and backward propagation in python with inputs given by $x = -2, y = 5, z = -4$. In the end, we are left with the gradient vector with three values, which tell us the sensitivity of the variables x, y, z on f. In code we either use the convention [dfdx,dfdy,dfdz] or sometimes a more concise notation that omits the [df] prefix. For example, we may simply write [dq] instead of [dfdq], and always assume that the gradient is computed on the final output.

Computational Graphs. These computations can also be nicely visualized with a circuit diagram called computational graphs. The diagram for the above example is shown in Figure 1.

3 Dynamics of Backpropagation

3.1 Intuitive Understanding

Notice that backpropagation is a beautifully local process. Every gate in a circuit diagram gets some inputs and can right away compute two things:

1. Its output value.
2. The local gradient of its output with respect to its inputs.

Notice that the gates can do this completely independently without being aware of any of the details of the full circuit that they are embedded in. However,

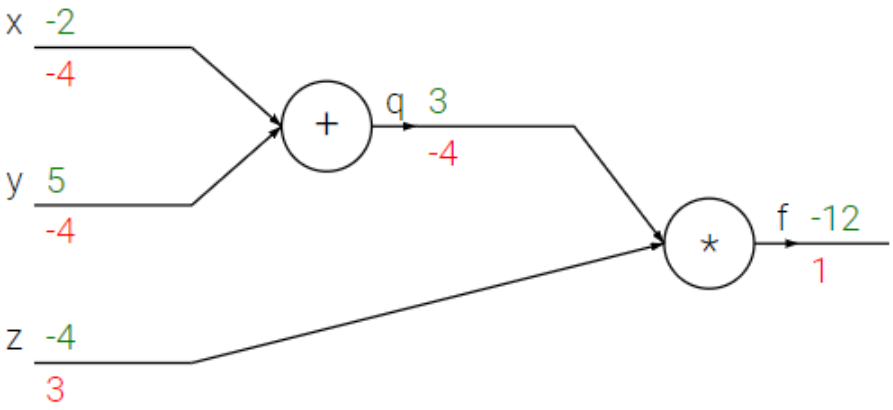

Figure 1: The real-valued "circuit" on left shows the visual representation of the computation. The forward pass computes values from inputs to output (shown in green). The backward pass then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (shown in red) all the way to the inputs of the circuit. The gradients can be thought of as flowing backwards through the circuit.

once the forward pass is over, during backpropagation the gate will eventually learn about the gradient of its output value on the final output of the entire circuit. Chain rule says that the gate should take that gradient and multiply it into every gradient it normally computes for all of its inputs.

Lets get an intuition for how this works by referring again to the example. The add gate received inputs $[-2, 5]$ and computed output 3. Since the gate is computing the addition operation, its local gradient for both of its inputs is $+1$. The rest of the circuit computed the final value, which is -12. During the backward pass in which the chain rule is applied recursively backwards through the circuit, the add gate (which is an input to the multiply gate) learns that the gradient for its output was -4. If we anthropomorphize the circuit as wanting to output a higher value (which can help with intuition), then we can think of the circuit as "wanting" the output of the add gate to be lower (due to negative sign), and with a force of 4. To continue the recurrence and to chain the gradient, the add gate takes that gradient and multiplies it to all of the local gradients for its inputs (making the gradient on both x and y $1 \times -4 = -4$). Notice that this has the desired effect: If x, y were to decrease (responding to their negative gradient) then the add gate's output would decrease, which in turn makes the multiply gate's output increase.

Backpropagation can thus be thought of as gates communicating to each other (through the gradient signal) whether they want their outputs to increase

or decrease (and how strongly), so as to make the final output value higher.

3.2 Modularity: Sigmoid Example

The gates we introduced above are relatively arbitrary. Any kind of differentiable function can act as a gate, and we can group multiple gates into a single gate, or decompose a function into multiple gates whenever it is convenient. Lets look at another expression that illustrates this point:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}} \tag{3}$$

as we will see later in the class, this expression describes a 2-dimensional neuron (with inputs $\mathbf{x}$ and weights $\mathbf{w}$) that uses the sigmoid activation function. But for now lets think of this very simply as just a function from inputs w, x to a single number. The function is made up of multiple gates. In addition to the ones described already above (add, mul, max), there are four more:

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

Where the functions f_c, f_a translate the input by a constant of c and scale the input by a constant of a, respectively. These are technically special cases of addition and multiplication, but we introduce them as new gates here since we do not need the gradients for the constants c, a. The full circuit then looks as in Figure

In the example above, we see a long chain of function applications that operates on the result of the dot product between $\mathbf{w}, \mathbf{x}$. The function that these operations implement is called the sigmoid function $\sigma(x)$ given by

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4}$$

It turns out that the derivative of the sigmoid function with respect to its input simplifies if you perform the derivation (after a fun tricky part where we have to add and subtract a 1 in the numerator):

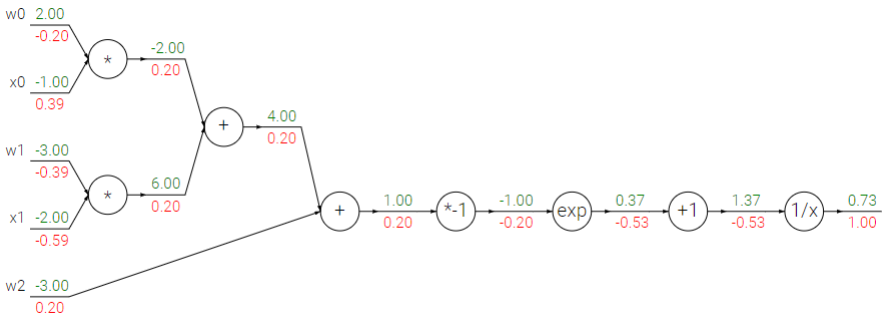

Figure 2: Example circuit for a 2D neuron with a sigmoid activation function. The inputs are [x0,x1] and the (learnable) weights of the neuron are [w0,w1,w2]. As we will see later, the neuron computes a dot product with the input and then its activation is softly squashed by the sigmoid function to be in range from 0 to 1.

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

As we see, the gradient turns out to simplify and becomes surprisingly simple. For example, the sigmoid expression receives the input 1.0 and computes the output 0.73 during the forward pass. The derivation above shows that the local gradient would simply be $(1 - 0.73) \times 0.73 = 0.1971$, as the circuit computed before (see the Figure 2), except this way it would be done with a single, simple and efficient expression (and with less numerical issues). Therefore, in any real practical application it would be very useful to group these operations into a single gate. Lets see if you can do this simple implementation of the backprop for this neuron in code.

3.3 Backprop in Practice: Staged Computation

Lets see this with another example. Suppose that we have a function of the form:

$$f(x,y) = \frac{x+\sigma(y)}{\sigma(x)+(x+y)^2} \tag{5}$$

To be clear, this function is completely useless and it's not clear why you would ever want to compute its gradient, except for the fact that it is a good example of backpropagation in practice. It is very important to stress that if you were to launch into performing the differentiation with respect to either x or y, you would end up with very large and complex expressions. However, it turns out that doing so is completely unnecessary because we don't need to have

an explicit function written down that evaluates the gradient. We only have to know how to compute it. Let us see how we would do this in code.

3.4 Patterns in Back Flow

It is interesting to note that in many cases the backward-flowing gradient can be interpreted on an intuitive level. For example, the three most commonly used gates in neural networks (add, mul, max), all have very simple interpretations in terms of how they act during backpropagation. Consider this example circuit in Figure 3.

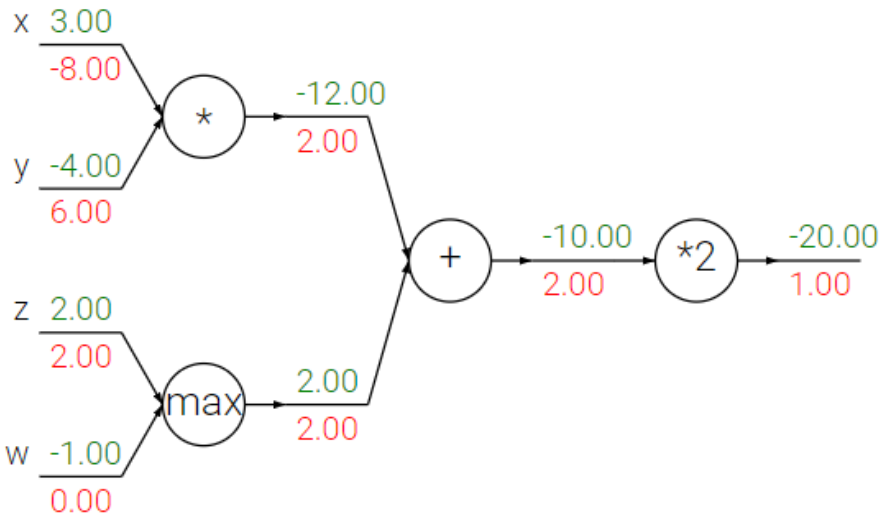

Figure 3: An example circuit demonstrating the intuition behind the operations that backpropagation performs during the backward pass in order to compute the gradients on the inputs. Sum operation distributes gradients equally to all its inputs. Max operation routes the gradient to the higher input. Multiply gate takes the input activations, swaps them and multiplies by its gradient.

Looking at the diagram above as an example, we can see that:

The **add gate** always takes the gradient on its output and distributes it equally to all of its inputs, regardless of what their values were during the forward pass. This follows from the fact that the local gradient for the add operation is simply $+1.0$, so the gradients on all inputs will exactly equal the gradients on the output because it will be multiplied by $\times 1.0$ (and remain unchanged). In the example circuit above, note that the $+$ gate routed the gradient of 2.00 to both of its inputs, equally and unchanged.

The **max gate** routes the gradient. Unlike the add gate which distributed the gradient unchanged to all its inputs, the max gate distributes the gradient

(unchanged) to exactly one of its inputs (the input that had the highest value during the forward pass). This is because the local gradient for a max gate is 1.0 for the highest value, and 0.0 for all other values. In the example circuit above, the max operation routed the gradient of 2.00 to the z variable, which had a higher value than w, and the gradient on w remains zero.

The **multiply gate** is a little less easy to interpret. Its local gradients are the input values (except switched), and this is multiplied by the gradient on its output during the chain rule. In the example above, the gradient on x is -8.00, which is -4.00×2.00.

Unintuitive Effects and their Consequences. Notice that if one of the inputs to the multiply gate is very small and the other is very big, then the multiply gate will do something slightly unintuitive: it will assign a relatively huge gradient to the small input and a tiny gradient to the large input. Note that in linear classifiers where the weights are dot producted $w^T x_i$ (multiplied) with the inputs, this implies that the scale of the data has an effect on the magnitude of the gradient for the weights. For example, if you multiplied all input data examples x_i by 1000 during preprocessing, then the gradient on the weights will be 1000 times larger, and you'd have to lower the learning rate by that factor to compensate. This is why preprocessing matters a lot, sometimes in subtle ways! And having intuitive understanding for how the gradients flow can help you debug some of these cases.

3.5 Gradients for Vectorized Operations

The above sections were concerned with single variables, but all concepts extend in a straight-forward manner to matrix and vector operations. However, one must pay closer attention to dimensions and transpose operations.

Optimization in general and gradients in particular is very vast subject with lots of nuances [2] which you may like to visit at some point in your research. You may also like to refer to this amazingly simple blog post about calculus on computational graphs. Here is another great resource. Finally here is a classical lecture from Prof. Patrick Winston.

References

[1] Paul J Werbos. "Backpropagation through time: what it does and how to do it". *Proceedings of the IEEE* 78, pp. 1550–1560, 1990.

[2] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". *Journal of machine learning research* 18, 2018.