Backpropagation

- Chain rule for the derivatives
- Backpropagation graphs
- Examples

Motivation for backpropagation

To do gradient descent, we need to calculate the gradient of the objective with respect to the parameters, θ. However, in a neural network, some parameters are not directly connected to the output. How do we calculate then the gradient of the objective with respect to these parameters? Backpropagation answers this question, and is a general application of the chain rule for derivatives.

Nomenclature

Forward propagation:

- Forward propagation is the act of calculating the values of the hidden and output units of the neural network given an input.
- It involves taking input $\mathbf{x}$, propagating it to through each hidden unit sequentially, until you arrive at the output $\mathbf{y}$. From forward propagation, we can also calculate the cost function $J(\theta)$.
- In this manner, the forward propagated signals are the activations.
- With the input as the "start" and the output as the "end," information propagates in a forward manner.

Backpropagation (colloquially called backprop):

- As its name suggests, now information is passed backward through the network, from the cost function and outputs to the inputs.
- The signal that is backpropagated are the gradients.
- It enables the calculation of gradients at every stage going back to the input layer.

Why do we need backpropagation?

It is possible in many cases to calculate an analytical gradient. So why use backpropagation?

- Evaluating analytical gradients may be computationally expensive.
- Backpropagation is generalizable and is often inexpensive.

A few further notes on backpropagation.

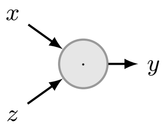
- Backpropagation is not the learning algorithm. It's the method of computing gradients.
- Backpropagation is not specific to multilayer neural networks, but a general way to compute derivatives of functions.

Computational graphs

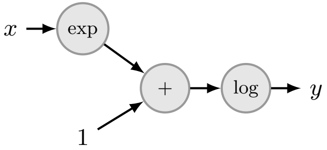
Computational graphs help to visualize and contextualize how we aim to backpropagate derivatives. In this class, we will let each inner node denote an operation.

Examples:

- $y = xz$. (Note: we let $\cdot$ denote dot product.)

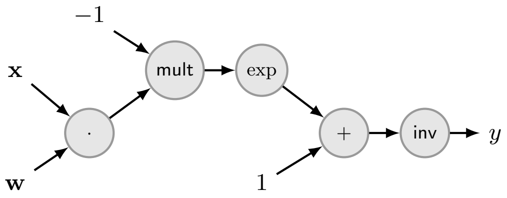


- $y = \text{softplus}(x)$.

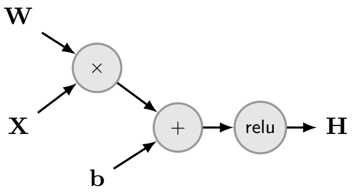


Computational graphs (cont)

- $y = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$. (Note, we let $\cdot$ denote dot product.)



- $\mathbf{H} = \mathrm{ReLU}(\mathbf{WX} + \mathbf{b})$. (Note: we let $\times$ denote matrix multiplication with appropriate order.)



Calculus chain rule

- Scalar version.
  Let $x, y, z \in \mathbb{R}$ and $f(\cdot), g(\cdot)$ be $\mathbb{R} \to \mathbb{R}$ functions. Suppose $y = g(x)$ and $z = f(x)$. Then $z = f(g(x))$. The calculus chain rule states:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- Vector version.
  Let $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$. Suppose $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. Suppose $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$. Then $z = f(g(\mathbf{x}))$. The multivariate calculus chain rule states:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

  This can be equivalently stated as:

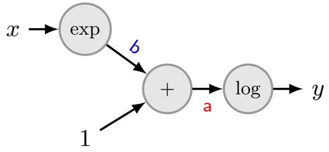
$$\nabla_{\mathbf{x}} z = \mathbf{J}(g)(\mathbf{x})^T \nabla_{\mathbf{y}} z$$

Backpropagation example: softplus

- $y = \text{softplus}(x) = \log(1 + \exp(x))$. We can analytically calculate the derivative.

$$\frac{dy}{dx} = \frac{\exp(x)}{1 + \exp(x)}$$

Let's now calculate it via backpropagation.

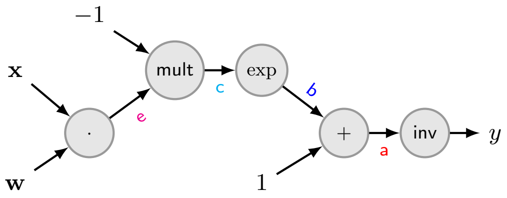


Here, we have that $b = \exp(x)$ and that $a = 1 + \exp(x)$. Applying the chain rule, we have that:

$$\begin{aligned} \frac{dy}{da} &= \frac{d}{da}\log a = \frac{1}{a} \\ \frac{dy}{db} &= \frac{da}{db}\frac{dy}{da} = \frac{dy}{da} \\ \frac{dy}{dx} &= \frac{db}{dx}\frac{dy}{db} = \exp(x)\frac{1}{a} \end{aligned}$$

Backpropagation example: sigmoid

- $y = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$.



Apply the chain rule:

$$\begin{aligned}
\frac{dy}{da} &= -\frac{1}{a^2} \\
\frac{dy}{db} &= \frac{dy}{da} \\
\frac{dy}{dc} &= \exp(c)\frac{dy}{db} \\
\frac{dy}{de} &= -\frac{dy}{dc}
\end{aligned}$$

Backpropagation example: sigmoid (cont.)

Finally, we have that:

$$\begin{aligned} \nabla_{\mathbf{x}} y &= \nabla_{\mathbf{x}} e \frac{dy}{de} \\ \nabla_{\mathbf{w}} y &= \nabla_{\mathbf{w}} e \frac{dy}{de} \end{aligned}$$

Putting it all together, we have that:

$$\begin{aligned} \nabla_{\mathbf{w}} y &= \mathbf{x} \frac{dy}{de} \\ &= -\mathbf{x} \frac{dy}{dc} \\ &= -\exp(-\mathbf{w}^T \mathbf{x}) \mathbf{x} \frac{dy}{db} \\ &= -\exp(-\mathbf{w}^T \mathbf{x}) \mathbf{x} \frac{dy}{da} \\ &= -\frac{\exp(-\mathbf{w}^T \mathbf{x})}{(1 + \exp(-\mathbf{w}^T \mathbf{x}))^2} \mathbf{x} \end{aligned}$$

A similar computation could be done to compute $\nabla_{\mathbf{x}} y$.

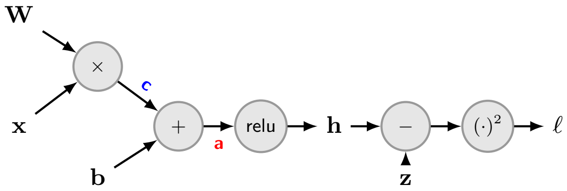
Backpropagation: a few intermediate notes

A few things to note thus far:

- So far, all multiplications have been scalar-scalar or vector-scalar, where order does not matter. It may be the case that we have matrix-vector or matrix-matrix multiplies, in which case the order should be correct. While one could do the rigorous math to determine the order, in general, one quick "trick" to get the order correct with less cognitive effort is to look at the dimensionality.
- Prior to backpropagation, we would have performed a forward pass to calculate intermediate values. These intermediate values were denoted a, b, c, e, etc. in the prior plots. We can cache these and would not have to recalculate their values in doing backpropagation.

Backpropagation: neural network layer

Here, $\mathbf{h} = \mathrm{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $\mathbf{h} \in \mathbb{R}^h$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{h \times m}$, and $\mathbf{b} \in \mathbb{R}^h$. While many output cost functions might be used, let's consider a simple squared-loss output $\ell = (\mathbf{h} - \mathbf{z})^2$ where $\mathbf{z}$ is some target value.



A few things to note:

- Note that $\nabla_{\mathbf{h}} \ell = 2(\mathbf{h} - \mathbf{z})$. We will start backpropagation at $\mathbf{h}$ rather than at ℓ.
- In the following backpropagation, we'll have need for elementwise multiplication. This is formally called a Hadamard product, and we will denote it via $\odot$. Concretely, the ith entry of $\mathbf{x} \odot \mathbf{y}$ is given by $x_i y_i$.

Backpropagation: neural network layer (cont.)

Applying the chain rule, we have:

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{a}} &= \mathbb{I}(\mathbf{a} > 0) \odot \frac{\partial \ell}{\partial \mathbf{h}} \\
\frac{\partial \ell}{\partial \mathbf{c}} &= \frac{\partial \ell}{\partial \mathbf{a}} \\
\frac{\partial \ell}{\partial \mathbf{x}} &= \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \frac{\partial \ell}{\partial \mathbf{c}} \\
&= \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{c}}
\end{aligned}$$

A few notes:

- For $\frac{\partial \mathbf{c}}{\partial \mathbf{x}}$, see example in the Tools notes.
- Why was the chain rule written right to left instead of left to right? This turns out to be a result of our convention of how we defined derivatives (using the "denominator layout notation") in the linear algebra notes.
- Though maybe not the most satisfying answer, you can always check the order of operations is correct by considering non-square matrices, where the dimensionality must be correct.

Backpropagation: neural network layer (cont.)

What about $\frac{\partial \ell}{\partial \mathbf{W}}$? In doing backpropagation, we have to calculate $\frac{\partial \mathbf{c}}{\partial \mathbf{W}}$ which is a 3-dimensional tensor.

However, we also intuit the following (informally):

- $\frac{\partial \ell}{\partial \mathbf{W}}$ is a matrix that is $h \times m$.
- The derivative of $\mathbf{W}\mathbf{x}$ with respect to $\mathbf{W}$ "ought to look like" $\mathbf{x}$.
- Since $\frac{\partial \ell}{\partial \mathbf{c}} \in \mathbb{R}^h$, and $\mathbf{x} \in \mathbb{R}^m$ then, intuitively,

$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \ell}{\partial \mathbf{c}} \mathbf{x}^T$$

This intuition turns out to be correct, and it is common to use this intuition. However, the first time around we should do this rigorously and then see the general pattern of why this intuition works.

Calculating a tensor derivative

$\frac{\partial \mathbf{c}}{\partial \mathbf{W}}$ is an $h \times m \times h$ tensor.

- Letting c_k denote the ith element of $\mathbf{c}$, we see that each $\frac{\partial c_i}{\partial \mathbf{W}}$ is an $h \times m$ matrix, and that there are h of these for each element c_i from $i = 1, \ldots, h$.
- The *matrix*, $\frac{\partial c_i}{\partial \mathbf{W}}$, can be calculated as follows:

$$\frac{\partial c_i}{\partial \mathbf{W}} = \frac{\partial}{\partial \mathbf{W}} \sum_{j=1}^{h} w_{ij} x_j$$

and thus, the (i,j)th element of this matrix is:

$$\left(\frac{\partial c_i}{\partial \mathbf{W}} \right)_{i,j} = x_j$$

It is worth noting that

$$\left(\frac{\partial c_k}{\partial \mathbf{W}} \right)_{i,j} = 0 \qquad \text{for } k \neq i$$

Calculating a tensor derivative (cont.)

Hence, $\frac{\partial c_i}{\partial \mathbf{W}}$ is a matrix where the ith row is $\mathbf{x}^T$ and all other rows are the zeros (we denote the zero vector by $\mathbf{0}$). i.e.,

$$\frac{\partial c_1}{\partial \mathbf{W}} = \begin{bmatrix} -\mathbf{x}^T- \\ -\mathbf{0}^T- \\ \vdots \\ -\mathbf{0}^T- \end{bmatrix} \qquad \frac{\partial c_2}{\partial \mathbf{W}} = \begin{bmatrix} -\mathbf{0}^T- \\ -\mathbf{x}^T- \\ \vdots \\ -\mathbf{0}^T- \end{bmatrix} \qquad \text{etc...}$$

Now applying the chain rule,

$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \mathbf{c}}{\partial \mathbf{W}} \frac{\partial \ell}{\partial \mathbf{c}}$$

is a tensor product between an $(h \times m \times h)$ tensor and an $(h \times 1)$ vector, whose resulting dimensionality is $(h \times m \times 1)$ or equivalently, an $(h \times m)$ matrix.

Calculating a tensor derivative (cont.)

We carry out this tensor-vector multiply in the standard way.

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{W}} &= \frac{\partial \mathbf{c}}{\partial \mathbf{W}} \frac{\partial \ell}{\partial \mathbf{c}} \\
&= \sum_{i=1}^{h} \frac{\partial c_i}{\partial \mathbf{W}} \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_i \\
&= \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_1 \begin{bmatrix} -\mathbf{x}^T- \\ -\mathbf{0}^T- \\ \vdots \\ -\mathbf{0}^T- \end{bmatrix} + \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_2 \begin{bmatrix} -\mathbf{0}^T- \\ -\mathbf{x}^T- \\ \vdots \\ -\mathbf{0}^T- \end{bmatrix} + \cdots + \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_h \begin{bmatrix} -\mathbf{0}^T- \\ -\mathbf{0}^T- \\ \vdots \\ -\mathbf{x}^T- \end{bmatrix} \\
&= \begin{bmatrix} \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_1 \mathbf{x}^T \\ \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_2 \mathbf{x}^T \\ \vdots \\ \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_h \mathbf{x}^T \end{bmatrix} \\
&= \frac{\partial \ell}{\partial \mathbf{c}} \mathbf{x}^T
\end{aligned}$$

A few notes on tensor derivatives

- In general, the simpler rule can be inferred via pattern intuition / looking at the dimensionality of the matrices, and these tensor derivatives need not be explicitly derived. (See intuition three slides prior.)
- Indeed, actually calculating these tensor derivatives, storing them, and then doing e.g., a tensor-vector multiply, is usually not a good idea for both memory and computation. In this example, storing all these zeros and performing the multiplications is unnecessary.
- If we know the end result is simply an outer product of two vectors, we need not even calculate an additional derivative in this step of backpropagation, or store an extra value (assuming the inputs were previously cached).