Matrix Calculus Cheatsheet

To train machine learning models, we use the standard SGD update rule (or some derived variant) defined as

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \nabla_\theta J(\theta)$$

However, of course, this requires us to compute $\nabla_\theta J(\theta)$, the gradient of the loss function with respect to our model weights/parameters. That is, for each parameter, we will need to calculate

$$\theta_j^{\text{new}} = \theta_j^{\text{old}} - \alpha \frac{\partial J(\theta)}{\partial \theta_j^{\text{old}}}.$$

For the rest of the class, we will discuss how to calculate $\nabla_\theta J(\theta)$ in two different ways:

1. By hand
2. Algorithmically: the backpropagation algorithm

Chain rule

The backbone of backprop is the chain rule which allows us to compute derivatives on compositions of functions.

Example 0.1 (Scalar-valued function)**.**

$$z = 3y$$

$$y = x^2$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = (3)(2x) = 6x$$

Example 0.2 (Vector-valued function)**.**

$$h = f(z)$$

$$z = Wx + b$$

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial z}\frac{\partial z}{\partial x} = ...$$

Example 0.3 (Derivation of neural network gradient)**.** Say we have a one layer neural net where we take from input vector x and want to produce a singular output neuron called the score. Diagram:

key realization: let's break up the equations into simpler pieces use placeholder variables:

$$h = f(z)$$

where

$$z = Wx + b$$

Now say we want $\frac{\partial s}{\partial b}$ all we got to do is apply the chain rule on

for which we get

$$\frac{\partial s}{\partial b} = \frac{\partial s}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial b} = u^T \operatorname{diag}\left(f'(z)\right) I = u^T \odot f'(z).$$

Now try calculating $\frac{\partial s}{\partial W}$. Answer:

$$\frac{\partial s}{\partial W} = \frac{\partial s}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial W}$$

Aside: if we need to calculate both $\frac{\partial s}{\partial W}$ and $\frac{\partial s}{\partial b}$, then we are redudant:

to resolve this redundancy, we can define a variable $\delta = \frac{\partial s}{\partial h}\frac{\partial h}{\partial z}$. This memoization concept is key in backpropogation.

Function shapes

See here.

Remark (**Derivative shape**). Let

$$f : \mathbb{R}^{(\text{input shape})} \to \mathbb{R}^{(\text{output shape})} .$$

Then its derivative will be a function of shape

$$\nabla f : \mathbb{R}^{(\text{input shape})} \to \mathbb{R}^{(\text{output shape})\times(\text{input shape})} .$$

Each extra added dimension on the output corresponds to taking partial derivatives with respect to the all of the input dimension (think of the gradient, for example). Meaning, for each output, we need to take the derivative with respect to all inputs seperately.

- Function of a scalar, returning a scalar: $\mathbb{R} \to \mathbb{R}$
  - Example: $f(x) = ax + b$
  - Derivative shape: $\frac{df}{dx} : \mathbb{R} \to \mathbb{R}$ (derivative)
- Function of a scalar, returning a vector: $\mathbb{R} \to \mathbb{R}^n$
  - Example: $f(x) = xv$
  - Derivative shape: $\frac{df}{dx} : \mathbb{R} \to \mathbb{R}^n$ (vector-valued derivative)
- Function of a vector, returning a scalar: $\mathbb{R}^n \to \mathbb{R}$
  - Example: $f(x) = v^\top x$
  - Derivative shape: $\nabla f(x) : \mathbb{R}^n \to \mathbb{R}^{1\times n}$ (gradient)
- Function of a vector, returning a vector: $\mathbb{R}^n \to \mathbb{R}^n$
  - Example: $f(x) = Mx$
  - Derivative shape:$\mathrm{J}(f(x)) : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ (jacobian)
- Function of a vector, returning a matrix: $\mathbb{R}^n \to \mathbb{R}^{n\times n}$
  - Example: $F(x) = xx^\top$
  - Derivative shape: $\nabla F(x) : \mathbb{R}^n \to \mathbb{R}^{n\times n\times n}$ (generalized Jacobian)

Many more examples

see cos 302 slides.

Neural network output shape

You'll notice a flaw in our machinery. The above shapes are true in pure mathematics. But in neural network land, we want to do the following SGD update:

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha\nabla_\theta J(\theta)$$

However, if our $\nabla_\theta J(\theta)$ shape is different from our old parameter shape θ^{old} , we can't possibly do this update–the shapes don't match. So we define our new shape convention as **the shape of the gradient is the shape of the**

parameters θ. Example: $\frac{\partial s}{\partial W}$ will be of shape $n \times m$:

$$\begin{bmatrix} \frac{\partial s}{\partial W_{11}} & \cdots & \frac{\partial s}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial s}{\partial W_{n1}} & \cdots & \frac{\partial s}{\partial W_{nm}} \end{bmatrix}.$$

This leads us into reshaping and tranpose: two tools we'll need to get the SGD update to work shape-wise.

Example 0.4. Using our neural network example from above, we want to compute $\frac{\partial s}{\partial W}$. We know that the shape should be the shape of W which is $n \times m$. So calculate our derivative as normal:

$$\frac{\partial s}{\partial W} = \delta \frac{\partial z}{\partial W}$$

where $\delta = \frac{\partial s}{\partial h} \frac{\partial h}{\partial z} = u^T \circ f'(z)$ (which is of shape $1 \times n$). So our answer is:

$$\frac{\partial s}{\partial W} = \delta x$$

But these dimensions don't give us our desired $n \times m$ shape. So let's use the tranpose to get an *outer product* (a hacky trick):

$$\frac{\partial s}{\partial W} = \delta^T x^T$$

Breaking down the shapes:

This shape convention idea is all tied to neural networks because each derivative component in the outputted derivative is with respect to one edge between the current layer and the next (we need a component for each edge):

Backpropogation

How can we do everything we discussed above (matrix calculus), but in an **efficient** manner for many layer deep neural networks with millions of parameters? Backpropogation is an algorithm that accomplishes these two feats.

The main ideas behind backprop include memoization of upstream gradients and computation graphs.

Definition 0.1 (Computation graph)**.** A directed acyclic graph whose root node represents the final mathematical expression and each node represents intermediate subexpressions. "forward propogation".

Example 0.5 (Forward propogation)**.** For:

we have:

Two things:

- Nodes represent the operations (besides input/output nodes)
- Edges pass along result of the operation

Example 0.6 (Back propogation)**.** Now calculate the gradients with respect to each node/edge needed of the final expression (s) with respect to the parameter you are updating (go backwards along the edges):

Each node has a **local gradient**: The gradient of its output with respect to its input (i.e., the edge going into the operation/node and the edge going out of the operation/node). Each node receives (from the right hand side since we are going backwards) an "**upstream gradient**". The goal then is to calculate and pass along the correct **downstream gradient** where `[downstream gradient] = [upstream gradient] x [local gradient]`.

Then use the **chain rule** to piece together each downstream gradient until you arrive at the desired gradient:

$$\frac{\partial s}{\partial b} = \frac{\partial s}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial b}$$

Remark: notice that each downstream gradient is a derivative of the final expression s (i.e., the global gradient). However, each local gradient which comprises the downstream global gradient is a derivative with respect to the local function.

Example 0.7 (Back propogation with multiple inputs)**.** Of course, neural network layers have many inputs for each hidden layer node. A simplified version of the neural network hidder layer evaluation expression is:

$$z = Wx.$$

The concept in the previous example generalizes for multiple inputs; just calculate the downstream gradient for each respective input! Then follow along the path of the desired downstream gradient.

Let's do another example of a simple function to solidy our understanding.

Example 0.8 (Forward pass computation graph construction)**.** Take the following function $f : \mathbb{R}^3 \to \mathbb{R}$.

$$f(x, y, z) = (x + y) \max(y, z)$$

We want to construct a computation graph for the forward pass.

1. Break down each "operation" into its own function and define that as a node.

$$\begin{aligned} a &= x + y \\ b &= \max(y, z) \\ f &= ab \end{aligned}$$

2. Formulate the graph where the edges are the inputs and the nodes are the sub-operation functions.

Appendix A: Useful Identities

Here we enumerate a bunch of helpful derivatives, simplifications, identities etc. in this section.

Definition 0.2 (Gradient)**.** yo

Definition 0.3 (Jacobian)**.** yo

Definition 0.4 (Various jacobian derivatives)**.**

$$\frac{\partial}{\partial x}(Wx + b) = W$$

$$\frac{\partial}{\partial b}(Wx + b) = I$$

$$\frac{\partial}{\partial u}\left(u^T h\right) = h^T$$

$$\frac{\partial}{\partial z}(f(z)) = \operatorname{diag}\left(f'(z)\right)$$

Definition 0.5 (Sigmoid derivative)**.**

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial}{\partial \theta_j} \sigma(z) = \sigma(z)[1 - \sigma(z)]$$

Appendix B: More matrix calculus examples

Taking gradients of loss functions with respect to matrix weight parameters is a key concept used in many machine learning classes–so you'll want to nail this concept down. To this end, we will provide many examples of taking gradients of loss functions with the hope of picking up useful patterns and identities (i.e., gradients of log).

To be updated eventually.

Example sources:

- CS 229
- CS 224N/CS231N
- CS 221 exams

Appendix C: Useful resources

- COS 302 slides
- Matrix cookbook
- https://explained.ai/matrix-calculus/