

COMS 4776 Lecture 5: Backpropagation

Richard Zemel

Overview

- We've seen that multilayer neural networks are powerful. But how can we actually learn them?
- Backpropagation is the central algorithm in this course.
 - It's is an algorithm for computing gradients.
 - Really it's an instance of reverse mode automatic differentiation, which is much more broadly applicable than just neural nets.
 - This is “just” a clever and efficient use of the Chain Rule for derivatives.
- Soon you will see how to implement an automatic differentiation library:
 - build the computation graph
 - vector-Jacobian products (VJP) for primitive ops
 - the backwards pass
- We'll cover Autograd, a lightweight autodiff tool. PyTorch's implementation is very similar.
 - You will probably never have to implement autodiff yourself but it is good to know its inner workings.

Confusing Terminology

- **Automatic differentiation (autodiff)** refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value.
- **Backpropagation** is the special case of autodiff applied to neural nets
 - But in machine learning, we often use backprop synonymously with autodiff
- **Autograd** is the name of a particular autodiff library we will cover in this lecture. There are many others, e.g. PyTorch, TensorFlow.

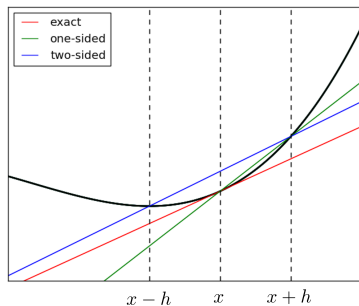
What Autodiff Is Not: Finite Differences

- We often use finite differences to check our gradient calculations.
- One-sided version:

$$\frac{\partial}{\partial x_i} f(x_1, \dots, x_N) \approx \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i, \dots, x_N)}{h}$$

- Two-sided version:

$$\frac{\partial}{\partial x_i} f(x_1, \dots, x_N) \approx \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i - h, \dots, x_N)}{2h}$$

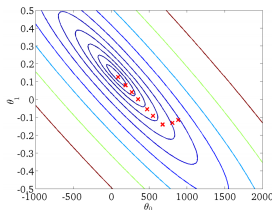


What Autodiff Is Not: Finite Differences

- Autodiff is not finite differences.
 - Finite differences are expensive, since you need to do a forward pass for *each* derivative.
 - It also induces huge numerical error.
 - Normally, we only use it for testing.
- Autodiff is both efficient (linear in the cost of computing the value) and numerically stable.

Recap: Gradient Descent

- **Recall:** gradient descent moves opposite the gradient (the direction of steepest descent)



- Weight space for a multilayer neural net: one coordinate for each weight or bias of the network, in *all* the layers
- Conceptually, not any different from what we've seen so far — just higher dimensional and harder to visualize!
- We want to compute the cost gradient $d\mathcal{J}/d\mathbf{w}$, which is the vector of partial derivatives.
 - This is the average of $d\mathcal{L}/d\mathbf{w}$ over all the training examples, so in this lecture we focus on computing $d\mathcal{L}/d\mathbf{w}$.

Univariate Chain Rule

- We've already been using the univariate Chain Rule.
- Recall: if $f(x)$ and $x(t)$ are univariate functions, then

$$\frac{d}{dt} f(x(t)) = \frac{df}{dx} \frac{dx}{dt}.$$

Univariate Chain Rule

Recall: Univariate logistic least squares model

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Let's compute the loss derivatives.

Univariate Chain Rule

How you would have done it in calculus class

$$\begin{aligned}\mathcal{L} &= \frac{1}{2}(\sigma(wx + b) - t)^2 \\ \frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w} \left[\frac{1}{2}(\sigma(wx + b) - t)^2 \right] \\ &= \frac{1}{2} \frac{\partial}{\partial w} (\sigma(wx + b) - t)^2 \\ &= (\sigma(wx + b) - t) \frac{\partial}{\partial w} (\sigma(wx + b) - t) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial w} (wx + b) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) x\end{aligned}$$
$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial}{\partial b} \left[\frac{1}{2}(\sigma(wx + b) - t)^2 \right] \\ &= \frac{1}{2} \frac{\partial}{\partial b} (\sigma(wx + b) - t)^2 \\ &= (\sigma(wx + b) - t) \frac{\partial}{\partial b} (\sigma(wx + b) - t) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial b} (wx + b) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b)\end{aligned}$$

What are the disadvantages of this approach?

Univariate Chain Rule

A more structured way to do it

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\frac{d\mathcal{L}}{dy} = y - t$$

$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{dy} \sigma'(z)$$

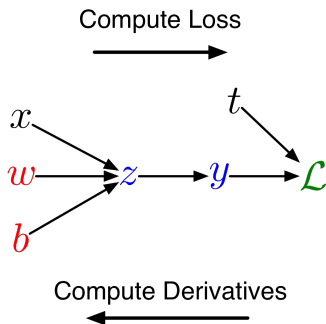
$$\frac{\partial \mathcal{L}}{\partial w} = \frac{d\mathcal{L}}{dz} x$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{d\mathcal{L}}{dz}$$

Remember, the goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Univariate Chain Rule

- We can diagram out the computations using a **computation graph**.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.



Univariate Chain Rule

A slightly more convenient notation:

- Use \bar{y} to denote the derivative $d\mathcal{L}/dy$, sometimes called the **error signal**.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).
- This is not a standard notation, but I couldn't find another one that I liked.

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\bar{y} = y - t$$

$$\bar{z} = \bar{y} \sigma'(z)$$

$$\bar{w} = \bar{z} x$$

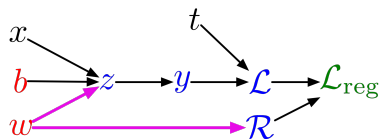
$$\bar{b} = \bar{z}$$

Multivariate Chain Rule

Problem: what if the computation graph has **fan-out** > 1 ?

This requires the **multivariate Chain Rule**!

L_2 -Regularized regression



$$z = wx + b$$

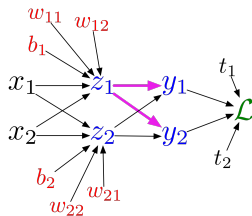
$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\mathcal{R} = \frac{1}{2}w^2$$

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \mathcal{R}$$

Multiclass logistic regression



$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$

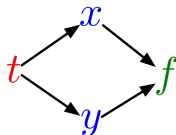
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$

$$\mathcal{L} = - \sum_k t_k \log y_k$$

Multivariate Chain Rule

- Suppose we have a function $f(x, y)$ and functions $x(t)$ and $y(t)$. (All the variables here are scalar-valued.) Then

$$\frac{d}{dt}f(x(t), y(t)) = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$



- Example:

$$f(x, y) = y + e^{xy}$$

$$x(t) = \cos t$$

$$y(t) = t^2$$

- Plug in to Chain Rule:

$$\begin{aligned} \frac{df}{dt} &= \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt} \\ &= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t \end{aligned}$$

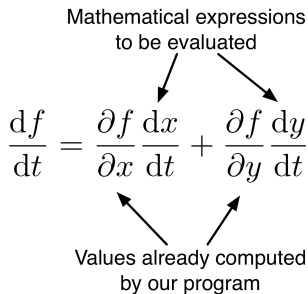
Multivariable Chain Rule

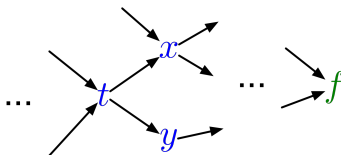
- In the context of backpropagation:

Mathematical expressions
to be evaluated

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

Values already computed
by our program





- In our notation:

$$\bar{t} = \bar{x} \frac{dx}{dt} + \bar{y} \frac{dy}{dt}$$

Backpropagation

Full backpropagation algorithm:

Let v_1, \dots, v_N be a **topological ordering** of the computation graph (i.e. parents come before children.)

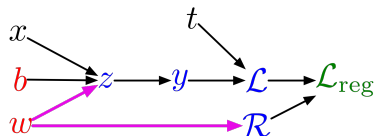
v_N denotes the variable we're trying to compute derivatives of (e.g. loss).

forward pass $\left[\begin{array}{l} \text{For } i = 1, \dots, N \\ \text{Compute } v_i \text{ as a function of } \text{Pa}(v_i) \end{array} \right.$

backward pass $\left[\begin{array}{l} \overline{v_N} = 1 \\ \text{For } i = N - 1, \dots, 1 \\ \overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i} \end{array} \right.$

Backpropagation

Example: univariate logistic least squares regression



Forward pass:

$$\begin{aligned}z &= wx + b \\y &= \sigma(z) \\ \mathcal{L} &= \frac{1}{2}(y - t)^2 \\ \mathcal{R} &= \frac{1}{2}w^2 \\ \mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda \mathcal{R}\end{aligned}$$

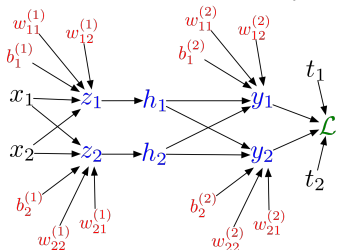
Backward pass:

$$\begin{aligned}\overline{\mathcal{L}_{\text{reg}}} &= 1 \\ \overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{R}} \\ &= \overline{\mathcal{L}_{\text{reg}}} \lambda \\ \overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{L}} \\ &= \overline{\mathcal{L}_{\text{reg}}} \\ \overline{y} &= \overline{\mathcal{L}} \frac{d\mathcal{L}}{dy} \\ &= \overline{\mathcal{L}} (y - t)\end{aligned}$$

$$\begin{aligned}\overline{z} &= \overline{y} \frac{dy}{dz} \\ &= \overline{y} \sigma'(z) \\ \overline{w} &= \overline{z} \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{d\mathcal{R}}{dw} \\ &= \overline{z} x + \overline{\mathcal{R}} w \\ \overline{b} &= \overline{z} \frac{\partial z}{\partial b} \\ &= \overline{z}\end{aligned}$$

Backpropagation

Multilayer Perceptron (multiple outputs):



Forward pass:

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2$$

Backward pass:

$$\bar{\mathcal{L}} = 1$$

$$\bar{y}_k = \bar{\mathcal{L}} (y_k - t_k)$$

$$\bar{w}_{ki}^{(2)} = \bar{y}_k h_i$$

$$\bar{b}_k^{(2)} = \bar{y}_k$$

$$\bar{h}_i = \sum_k \bar{y}_k w_{ki}^{(2)}$$

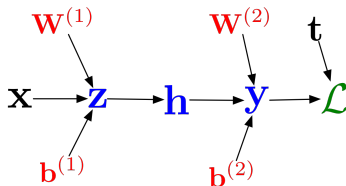
$$\bar{z}_i = \bar{h}_i \sigma'(z_i)$$

$$\bar{w}_{ij}^{(1)} = \bar{z}_i x_j$$

$$\bar{b}_i^{(1)} = \bar{z}_i$$

Vector Form

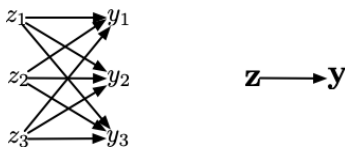
- Computation graphs showing individual units are cumbersome.
- As you might have guessed, we typically draw graphs over the vectorized variables.



- We pass messages back analogous to the ones for scalar-valued nodes.

Vector Form

- Consider this computation graph:



- Backprop rules:

$$\bar{z}_j = \sum_k \bar{y}_k \frac{\partial y_k}{\partial z_j} \qquad \bar{\mathbf{z}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}}^\top \bar{\mathbf{y}},$$

where $\partial \mathbf{y} / \partial \mathbf{z}$ is the **Jacobian matrix**:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \frac{\partial y_1}{\partial z_1} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \cdots & \frac{\partial y_m}{\partial z_n} \end{pmatrix}$$

Vector Form

Examples

- Matrix-vector product

$$\mathbf{z} = \mathbf{W}\mathbf{x} \quad \frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{W} \quad \bar{\mathbf{x}} = \mathbf{W}^\top \bar{\mathbf{z}}$$

- Elementwise operations

$$\mathbf{y} = \exp(\mathbf{z}) \quad \frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix} \quad \bar{\mathbf{z}} = \exp(\mathbf{z}) \circ \bar{\mathbf{y}}$$

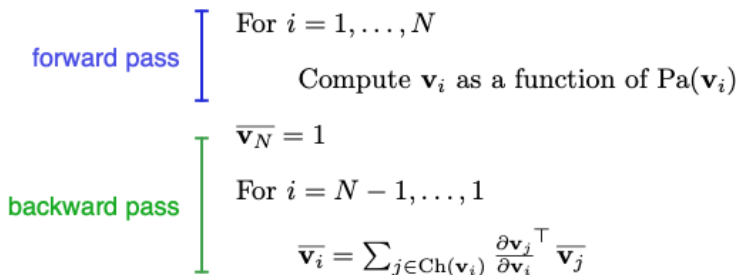
- Note: we never explicitly construct the Jacobian. It's usually simpler and more efficient to compute the VJP directly.

Vector Form

Full backpropagation algorithm (vector form):

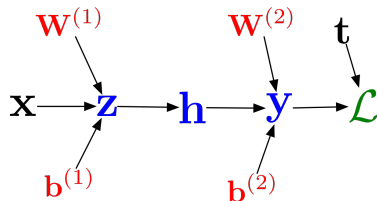
Let $\mathbf{v}_1, \dots, \mathbf{v}_N$ be a **topological ordering** of the computation graph (i.e. parents come before children.)

\mathbf{v}_N denotes the variable we're trying to compute derivatives of (e.g. loss). It's a scalar, which we can treat as a 1-D vector.



Vector Form

MLP example in vectorized form:



Forward pass:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

$$\mathbf{h} = \sigma(\mathbf{z})$$

$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \|\mathbf{t} - \mathbf{y}\|^2$$

Backward pass:

$$\bar{\mathcal{L}} = 1$$

$$\bar{\mathbf{y}} = \bar{\mathcal{L}}(\mathbf{y} - \mathbf{t})$$

$$\overline{\mathbf{W}^{(2)}} = \bar{\mathbf{y}}\mathbf{h}^\top$$

$$\overline{\mathbf{b}^{(2)}} = \bar{\mathbf{y}}$$

$$\bar{\mathbf{h}} = \mathbf{W}^{(2)\top} \bar{\mathbf{y}}$$

$$\bar{\mathbf{z}} = \bar{\mathbf{h}} \circ \sigma'(\mathbf{z})$$

$$\overline{\mathbf{W}^{(1)}} = \bar{\mathbf{z}}\mathbf{x}^\top$$

$$\overline{\mathbf{b}^{(1)}} = \bar{\mathbf{z}}$$

Computational Cost

- Computational cost of forward pass: one **add-multiply operation** per weight

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

- Computational cost of backward pass: two add-multiply operations per weight

$$\begin{aligned}\overline{w_{ki}^{(2)}} &= \overline{y_k} h_i \\ \overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)}\end{aligned}$$

- Rule of thumb: the backward pass is about as expensive as two forward passes.
- For a multilayer perceptron, this means the cost is linear in the number of layers, quadratic in the number of units per layer.

Closing Thoughts

- Backprop is used to train the overwhelming majority of neural nets today.
 - Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.
- Despite its practical success, backprop is believed to be neurally implausible.
 - No evidence for biological signals analogous to error derivatives.
 - All the biologically plausible alternatives we know about learn much more slowly (on computers).
 - So how on earth does the brain learn?

Closing Thoughts

The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.

– Don Knuth

- By now, we've seen three different ways of looking at gradients:
 - **Geometric:** visualization of gradient in weight space
 - **Algebraic:** mechanics of computing the derivatives
 - **Implementational:** efficient implementation on the computer
- When thinking about neural nets, it's important to be able to shift between these different perspectives!