Backpropagation

He He
Slides based on Lecture 12b from David Rosenberg's course materials
(https://github.com/davidrosenberg/mlcourse)

CDS, NYU

April 20, 2021

Back-propagation

A brief history of artificial neural networks

early 1940s–late 1960s

- Initial idea from neuroscience: create a computational model of neural networks.
- Development: perceptron [Rosenblatt, 1958], networks with many layers.
- Optimization: automatic differentiation [Linnainmaa, 1970].

late 1960s–late 1980s

- Computers didn't have enough processing power [Minsky and Papert, 1969].
- Back-propagation invented [Werbos, 1975] (but still hard to train).
- AI research focused on expert systems and symbolic systems.

late 1980s–early 2000s

- SVMs and linear models dominated ML.

Example: MLP Regression

- **Input space**: $\mathcal{X} = \mathbf{R}$
- **Action Space / Output space**: $\mathcal{A} = \mathcal{Y} = \mathbf{R}$
- **Hypothesis space**: MLPs with a single 3-node hidden layer:

$$f(x) = w_0 + w_1 h_1(x) + w_2 h_2(x) + w_3 h_3(x),$$

where

$$h_i(x) = \sigma(v_i x + b_i) \text{ for } i = 1, 2, 3,$$

for some fixed activation function $\sigma : \mathbf{R} \to \mathbf{R}$.

- What are the parameters we need to fit?

$$b_1, b_2, b_3, v_1, v_2, v_3, w_0, w_1, w_2, w_3 \in \mathbf{R}$$

How to choose the best hypothesis?

- As usual, choose our prediction function using empirical risk minimization.
- Our hypothesis space is parameterized by

$$\theta = (b_1, b_2, b_3, v_1, v_2, v_3, w_0, w_1, w_2, w_3) \in \Theta = \mathbf{R}^{10}$$

- For a training set $(x_1, y_1), \ldots, (x_n, y_n)$, find

$$\hat{\theta} = \arg\min_{\theta \in \mathbf{R}^{10}} \frac{1}{n} \sum_{i=1}^{n} (f(x_i; \theta) - y_i)^2 .$$

- Gradient descent:
  - Is it differentiable w.r.t. θ? $f(x) = w_0 + \sum_{i=1}^{3} w_i \tanh(v_i x + b_i)$.
  - Is it convex in θ? Might converge to a local minimum.

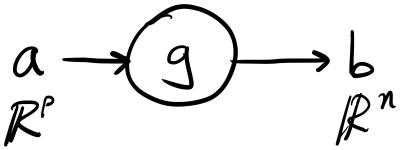
Gradient descent for (large) neural networks

- Mathematically, it's just *partial derivatives*, which you can compute by hand using the *chain rule*.
  - In practice, this could be time-consuming and error-prone.
- How do we compute gradients in a systematic and efficient way?
  - *Back-propagation* (a special case of automatic differentiation).
  - Not limited to neural networks.
- Visualize with *computation graphs*.
  - Avoid long equations.
  - Structure of the computation (modularity and dependency), which allows for modern computation frameworks such as Tensorflow/Pytorch/MXNet/etc.

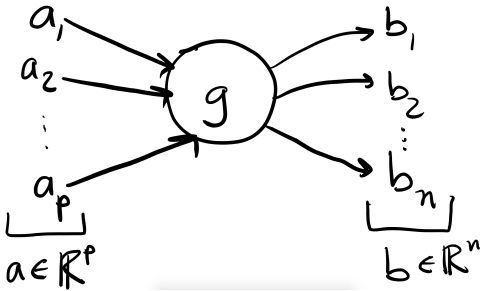
Function as a graph

- Function as a *node* that takes in a set of *inputs* and produces a set of *outputs*.
- Example: $g : \mathbf{R}^p \to \mathbf{R}^n$.

- Typical computation graph:

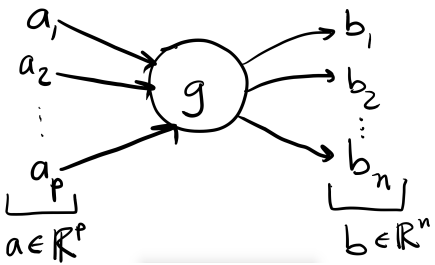


- Broken out into components:



Partial Derivatives of an affine function

- Define the affine function $g(x) = Mx + c$, for $M \in \mathbf{R}^{n \times p}$ and $c \in \mathbf{R}$.



- Let $b = g(a) = Ma + c$. What is b_i?
- b_i depends on the ith row of M:

$$b_i = \sum_{k=1}^{p} M_{ik} a_k + c_i.$$

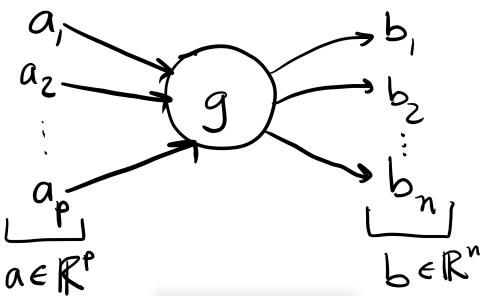
- If $a_j \leftarrow a_j + \delta$, what is b_i?

$$b_i \leftarrow b_i + M_{ij}\delta.$$

Partial derivative/gradient measures *sensitivity*: If we perturb an input a little bit, how much does an output change?

Partial Derivatives in general

- Consider a function $g : \mathbf{R}^p \to \mathbf{R}^n$.



- Partial derivative $\frac{\partial b_i}{\partial a_j}$ is the instantaneous rate of change of b_i as we change a_j.
- If we change a_j slightly to

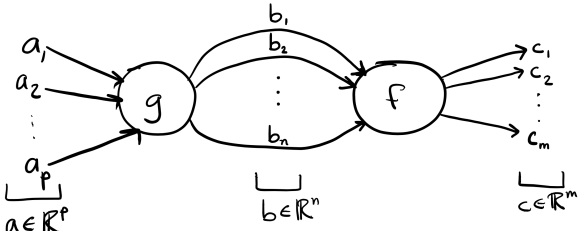
$$a_j + \delta,$$

- Then (for small δ), b_i changes to approximately

$$b_i + \frac{\partial b_i}{\partial a_j}\delta.$$

Compose multiple functions

- Compose two functions $g : \mathbf{R}^p \to \mathbf{R}^n$ and $f : \mathbf{R}^n \to \mathbf{R}^m$.
- $b = g(a)$, $c = f(b)$.



- How does change in a_j affect c_i?
- Visualize **chain rule**:
  - Sum changes induced on all paths from a_j to c_i.
  - Changes on one path is the product of changes on each edge along the path.

$$\frac{\partial c_i}{\partial a_j} = \sum_{k=1}^{n} \frac{\partial c_i}{\partial b_k} \frac{\partial b_k}{\partial a_j}.$$

Example: Linear least squares

- Hypothesis space $\left\{ f(x) = w^T x + b \mid w \in \mathbf{R}^d, b \in \mathbf{R} \right\}$.
- Data set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \mathbf{R}$.
- Define

$$\ell_i(w, b) = \left[\left(w^T x_i + b \right) - y_i \right]^2.$$

- In SGD, in each round we'd choose a random index $i \in 1, \ldots, n$ and take a gradient step

$$\begin{aligned} w_j &\leftarrow w_j - \eta \frac{\partial \ell_i(w, b)}{\partial w_j}, \text{ for } j = 1, \ldots, d \\ b &\leftarrow b - \eta \frac{\partial \ell_i(w, b)}{\partial b}, \end{aligned}$$

  for some step size $\eta > 0$.

- Let's see how to calculate these partial derivatives on a computation graph.

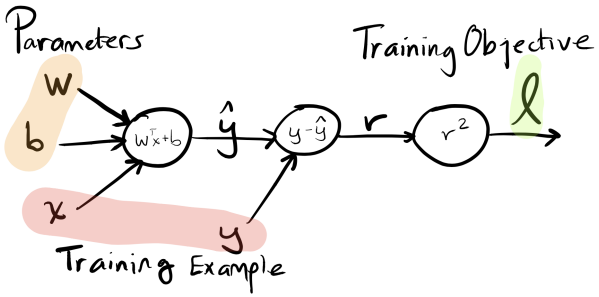
Computation Graph and Intermediate Variables

- For a generic training point (x, y), denote the loss by

$$\ell(w, b) = \left[\left(w^T x + b\right) - y\right]^2.$$

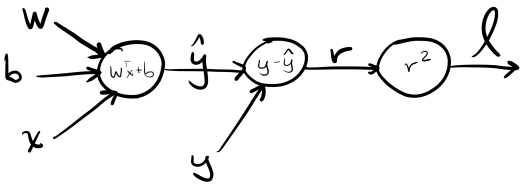
- Let's break this down into some intermediate computations:

$$\begin{aligned} \text{(prediction) } \hat{y} &= \sum_{j=1}^{d} w_j x_j + b \\ \text{(residual) } r &= y - \hat{y} \\ \text{(loss) } \ell &= r^2 \end{aligned}$$



Partial Derivatives on Computation Graph

- We'll work our way from graph output ℓ back to the parameters w and b:



$$
\begin{aligned}
\frac{\partial \ell}{\partial r} &= 2r \\
\frac{\partial \ell}{\partial \hat{y}} &= \frac{\partial \ell}{\partial r}\frac{\partial r}{\partial \hat{y}} = (2r)(-1) = -2r \\
\frac{\partial \ell}{\partial b} &= \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r \\
\frac{\partial \ell}{\partial w_j} &= \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_j} = (-2r)\,x_j = -2rx_j
\end{aligned}
$$

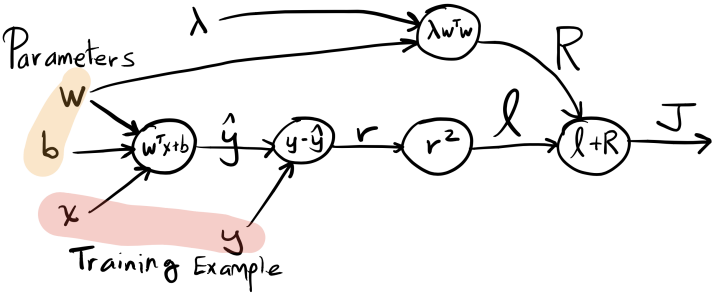
Example: Ridge Regression

- For training point (x, y), the ℓ_2-regularized objective function is

$$J(w, b) = \left[\left(w^T x + b\right) - y\right]^2 + \lambda w^T w.$$

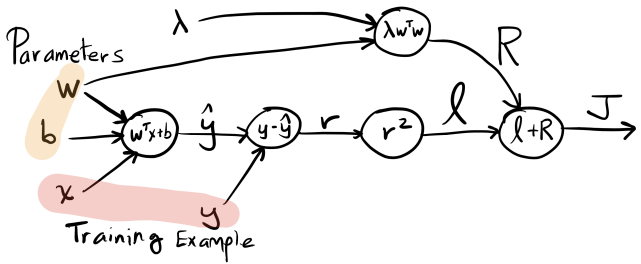
- Let's break this down into some intermediate computations:

$$\begin{aligned}
\text{(prediction) } \hat{y} &= \sum_{j=1}^{d} w_j x_j + b \\
\text{(residual) } r &= y - \hat{y} \\
\text{(loss) } \ell &= r^2 \\
\text{(regularization) } R &= \lambda w^T w \\
\text{(objective) } J &= \ell + R
\end{aligned}$$



Partial Derivatives on Computation Graph

- We'll work our way from graph output ℓ back to the parameters w and b:



$$\frac{\partial J}{\partial \ell} = \frac{\partial J}{\partial R} = 1$$

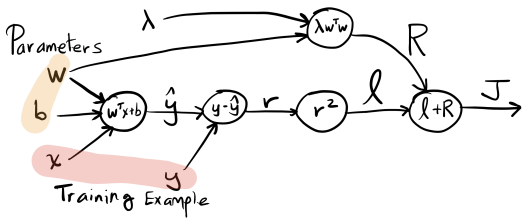
$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial J}{\partial \ell}\frac{\partial \ell}{\partial r}\frac{\partial r}{\partial \hat{y}} = (1)(2r)(-1) = -2r$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r$$

$$\frac{\partial J}{\partial w_j} = \text{Exercise}$$

Backpropagation overview

- **Learning**: run gradient descent to find the parameters that minimize our objective J.
- Backpropagation: compute gradient w.r.t. each (trainable) parameter $\frac{\partial J}{\partial \theta_i}$.



Forward pass: Compute intermediate function values, i.e. output of each node

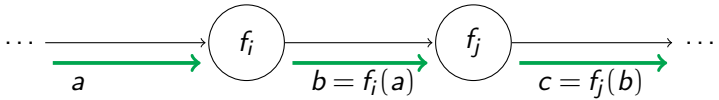
Backward pass: Compute the partial derivative of J w.r.t. all intermediate variables and the model parameters

How to save computation?

- Path sharing: each node needs to *cache the intermediate results*.
- Think dynamic programming.

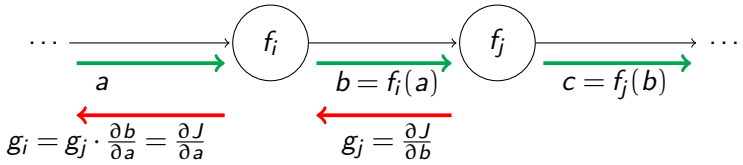
Forward pass

- Order nodes by **topological sort** (every node appears before its children)
- For each node, compute the output given the input (output of its parents).
- Forward at intermediate node f_i and f_j:



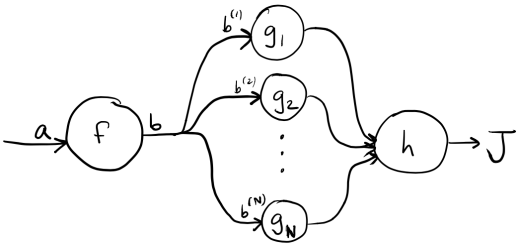
Backward pass

- Order nodes in **reverse topological order** (every node appear after its children)
- For each node, compute the partial derivative of its output w.r.t. its input, multiplied by the partial derivative from its children (chain rule).
- Backward at intermediate node f_i:



Multiple children

- First sum partial derivatives from all children, then multiply.



- Backprop for node f:
- **Input**: $\frac{\partial J}{\partial b^{(1)}}, \ldots, \frac{\partial J}{\partial b^{(N)}}$
  (Partials w.r.t. inputs to all children)
- **Output**:

$$\frac{\partial J}{\partial b} = \sum_{k=1}^{N} \frac{\partial J}{\partial b^{(k)}}$$

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial b}\frac{\partial b}{\partial a}$$

Backpropagation in practice

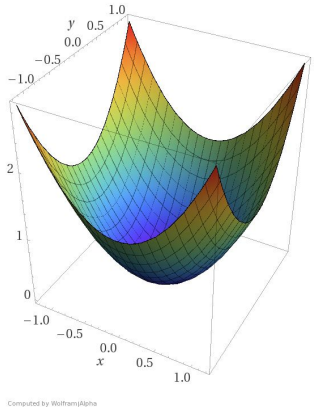
- Inputs and outputs of nodes are generally *vectorized* (efficient to compute on GPUs).
- Computation graphs can be composed from a set of *basic operation nodes*, e.g., addition/multiplication, dot product, logistic function etc.
- Programming paradigms:

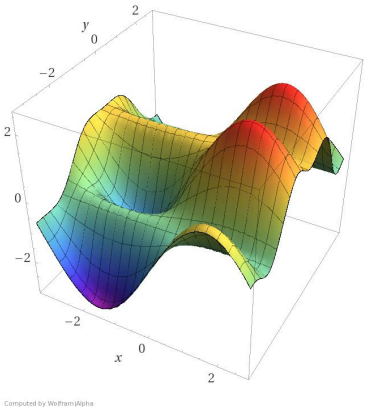
  Symbolic Specify all computation before data—efficient, e.g., Tensorflow.

  Imperative Specify the computation step by step—flexible/easier to write, e.g., Pytorch.

  Hybrid Can use either paradigm for computation subgraphs, e.g., MXNet.

Non-convex optimization



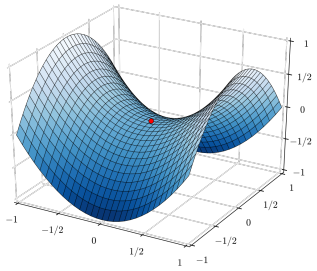


- Left: convex loss function. Right: non-convex loss function.

Non-convex optimization: challenges

Optimization of neural networks is generally hard.

- Converge to a bad local minimum.
  - Try different initialization and rerun.
- Saddle point.
  - Doesn't often happen with SGD.
  - Second partial derivative test.
- "Flat" region: low gradient magnitude
  - Use ReLU instead of sigmoid as activation functions.
- High curvature: high gradient magnitude
  - Gradient clipping.
  - Adaptive step size.



Review

- Backpropagation is an algorithm to compute gradient (partial derivatives + chain rule) efficiently.
- It is used in gradient descent optimization with neural networks.
- Key idea: function composition and dynamic programming
- In practice, efficient software exists (backpropagation, neural network building blocks, optimization algorithms etc.).