CS395T: Foundations of Machine Learning for Systems Researchers

Fall 2025

Lecture 2: Gradient Computation in Abstract Neural Networks

Optimization of DNNs: Turing Award citations (2018)

Backpropagation: In a 1986 paper, “Learning Internal Representations by Error Propagation,” co-authored with David Rumelhart and Ronald Williams, Hinton demonstrated that the backpropagation algorithm allowed neural nets to discover their own internal representations of data, making it possible to use neural nets to solve problems that had previously been thought to be beyond their reach. The backpropagation algorithm is standard in most neural networks today.

Improving backpropagation algorithms: LeCun proposed an early version of the backpropagation algorithm (backprop) and gave a clean derivation of it based on variational principles. His work to speed up backpropagation algorithms included describing two simple methods to accelerate learning time.

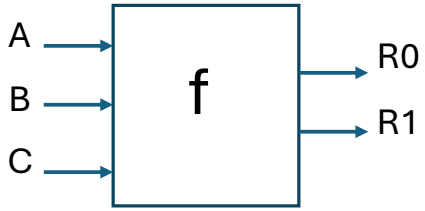
Goal of this lecture

- Standard presentations of back propagation in neural networks
  - Biological metaphors like neurons and synapses come in the way
  - Properties of activation functions are distraction
  - Chain rule of calculus: not obvious how to use it for complicated networks
- Presentation in this lecture
  - Abstraction for neural networks: *parameterized programs*
  - Gradient computation
    - Abstraction (what?): sum over paths
    - Implementation (how?): compositional algorithm on dataflow graph representation
  - Handles complicated neural networks
    - Nonlinear functions like sin(x), cos(x), sin(cos(x)), $e^{\cos(x)}$
    - Arbitrary DAGs including skip connections and weight sharing

Organization

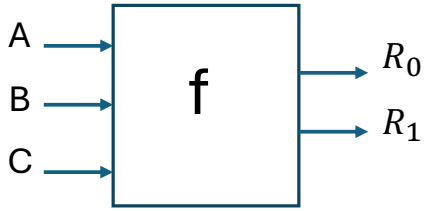
- **Basics:**
  - Partial derivatives, gradients, Jacobians
  - Linear regression:
    - Parameter optimization: solving linear systems
  - Neural networks: parameter optimization requires solving non-linear systems
    - Gradient descent
- **Parameterized programs:** abstraction for neural networks
  - Dataflow graph representation
  - Function evaluation: forward propagation
  - Computing gradients: back propagation
  - Parameter optimization using gradient descent

Basic concepts:
Partial derivatives, gradients, Jacobians (I)



- Convention:
  - variables are written as capital letters (A,B,R0)
  - variable values are written as corresponding small letters (a,b,r0)
- Partial derivative of R0 wrt A: $\frac{\partial R0}{\partial A}$
  - Function that tells you how much R0 changes if A is changed a small amount
  - Value of partial derivative depends on values of A,B,C (consider $f(x) = x^2$)
  - Notation: $\frac{\partial R0}{\partial A}(a, b, c)$ is "value of partial derivative of R0 wrt A when input is (a, b, c)"
- If value of A changes from *a* to *(a+Δa)*, value of R0 changes by $\frac{\partial R0}{\partial A}(a, b, c) * \Delta a$

Basic concepts:
Partial derivatives, gradients, Jacobians (II)



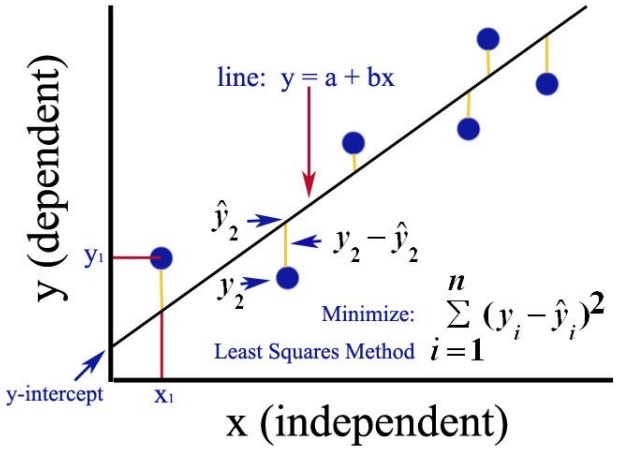
- **Gradient** of R0: ∇R0
  - Column vector containing partial derivatives of R0 $\begin{pmatrix} \frac{\partial \mathrm{R0}}{\partial \mathrm{A}} \\ \frac{\partial \mathrm{R0}}{\partial \mathrm{B}} \\ \frac{\partial \mathrm{R0}}{\partial \mathrm{C}} \end{pmatrix}$
  - If input changes from *(a,b,c)* to *(a+Δa, b+Δb, c+Δc),* R0 changes by $(\nabla R0)^{\mathsf{T}}(a,b,c) * \begin{pmatrix} \Delta a \\ \Delta b \\ \Delta c \end{pmatrix}$
- **Jacobian of f: J_f**
  - Matrix whose columns are gradients of outputs $\mathrm{J_f} = \begin{pmatrix} \frac{\partial \mathrm{R0}}{\partial \mathrm{A}} & \frac{\partial \mathrm{R1}}{\partial \mathrm{A}} \\ \frac{\partial R0}{\partial \mathrm{B}} & \frac{\partial \mathrm{R1}}{\partial \mathrm{B}} \\ \frac{\partial \mathrm{R0}}{\partial \mathrm{C}} & \frac{\partial \mathrm{R1}}{\partial \mathrm{C}} \end{pmatrix}$
  - If inputs change from *(a,b,c)* to to *(a+Δa, b+Δb, c+Δc),* outputs change by $\mathrm{J_f}^{\mathsf{T}}(a,b,c) * \begin{pmatrix} \Delta a \\ \Delta b \\ \Delta c \end{pmatrix}$

Parameter optimization: linear regression

- Given set of *n* training data samples {(x_i,y_i)}, find "best" line Y = a+b*X for given data
  - **Model** for training data
  - **Model parameters:** a and b
  - **Generalization**: model lets you predict y for x not in training sample
- Scoring a proposal (a,b): **loss function**
  - Compute total square error/residual
  - Detail: we use mean square error (L2 loss)

$$Loss(a,b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (a+b*x_i))^2$$

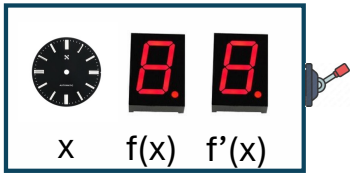
- Parameter values that minimize loss
  - Conceptually, a big search problem
  - Better solution: use gradients
    - > Set ∇ Loss(a,b) to zero
    - > Solve two **linear** equations
- Neural networks are more complicated
  - Parameter optimization requires solving **non-linear** equations
  - Popular method: **gradient descent**



Karl Friedrich Gauss

$$n * a + (\Sigma_i x_i) * b = \Sigma_i y_i$$
$$(\Sigma_i x_i) * a + (\Sigma_i x_i^2) * b = \Sigma_i x_i y_i$$

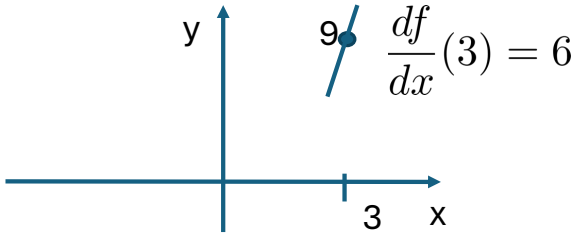
Function minimization using gradient descent (I)

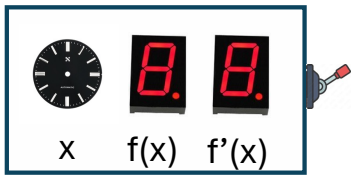


- Problem: given function f(x), find x that minimizes f. Assumptions:
  - f has minimum and first derivative, but no closed-form expression is available for function or derivative
  - However, we can ask for the value of the function and its derivative at any point
- In general, need iterative search

Function minimization using gradient descent (I)

- Pick some value of x (say 3) and find f(x) (say 9)
  - To see if we have a minimum, find gradient (say 6)
  - Gradient not zero, so not at minimum
  - Good point to try next?
- Gradient gives us two pieces of information
  - Sign:
    - gradient is positive, so increasing x increases function value $\Rightarrow$ we should decrease x.
  - Magnitude:
    - if the magnitude of gradient is small, close to the minimum $\Rightarrow$ step size should be small
    - if magnitude of gradient is large, may be far from minimum $\Rightarrow$ step size should be larger
- This suggests an iterative scheme of the form $x_{i+1} = x_i - \frac{df}{dx}(x_i)$

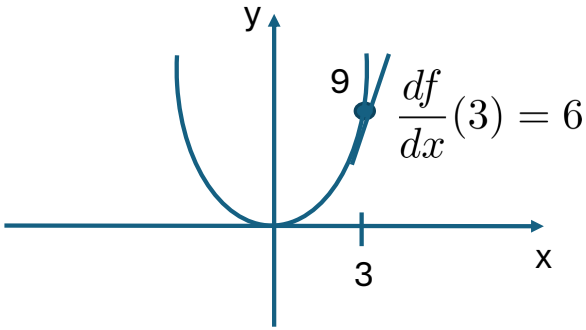


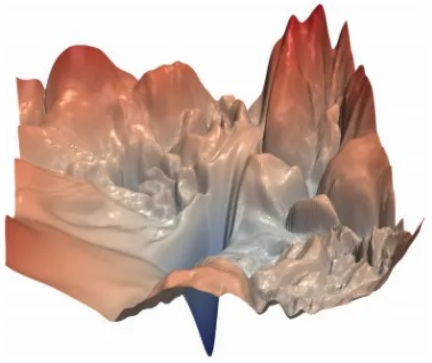


Function minimization using gradient descent (II)

- **Problem:** when magnitude of derivative is large, step size will be big, and we may overshoot minimum
  - Example: function is $y = x^2$ and x_0 is 3.
  - May never converge
- **Solution:** change iterative scheme to $x_{i+1} = x_i - \alpha_i \frac{df}{dx}(x_i)$
  - $0<\alpha_i<1$ is a sequence of numbers called *learning rate*
  - Convergence to minimum if sequence satisfies Robbins-Munro conditions

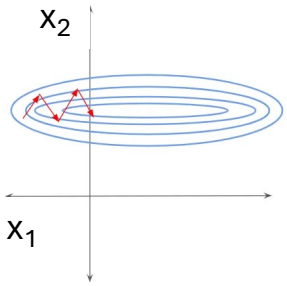
    $\sum_{i=0}^{\infty} \alpha_n = \infty \qquad \sum_{i=0}^{\infty} \alpha_n^2 < \infty$
  - Safe but slow solution: set α_i to $\frac{1}{i}$
- Iterative scheme for multiple variable function: $\underline{x}_{i+1} = \underline{x}_i - \alpha_i \nabla f(\underline{x}_i)$
- Gradient descent finds local minimum, may not be global minimum

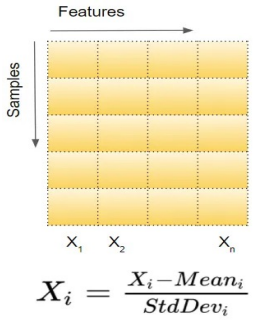


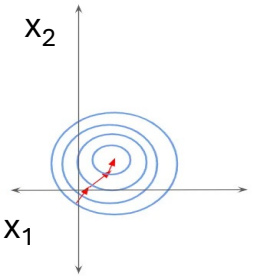


Hao Li et al.

Variations on theme (I)





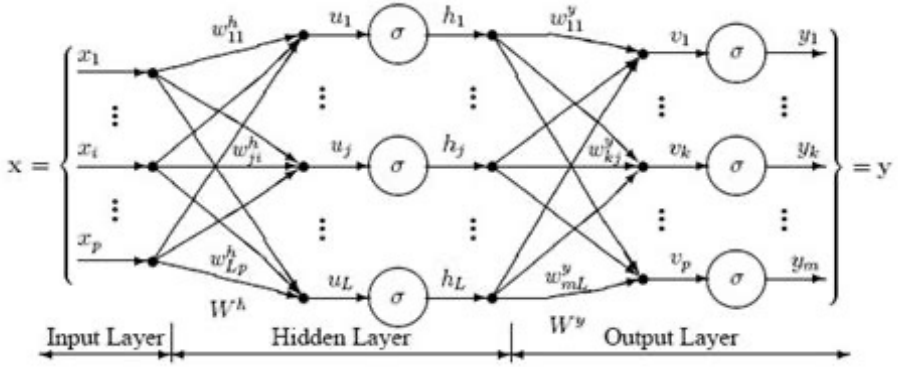


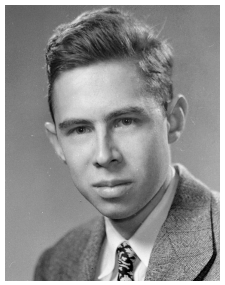
Ketan Doshi

- Rate of convergence
  - How fast is gradient changing?
    - Curvature: second-derivative (Hessian in higher dimensions)
  - Noisy values and gradients: penalty for inaccuracy
- Batch normalization (BN) Ioffe & Szegedy 2015
  - Reshape optimization landscape to avoid “valleys”
- Other iterative schemes (optimizers): Adagrad, Nesterov momentum, AdamW, ….
  - https://www.slideshare.net/SebastianRuder/optimization-for-deep-learning

Computing Gradients in Programs

Multilayer Perceptron (MLP) Example

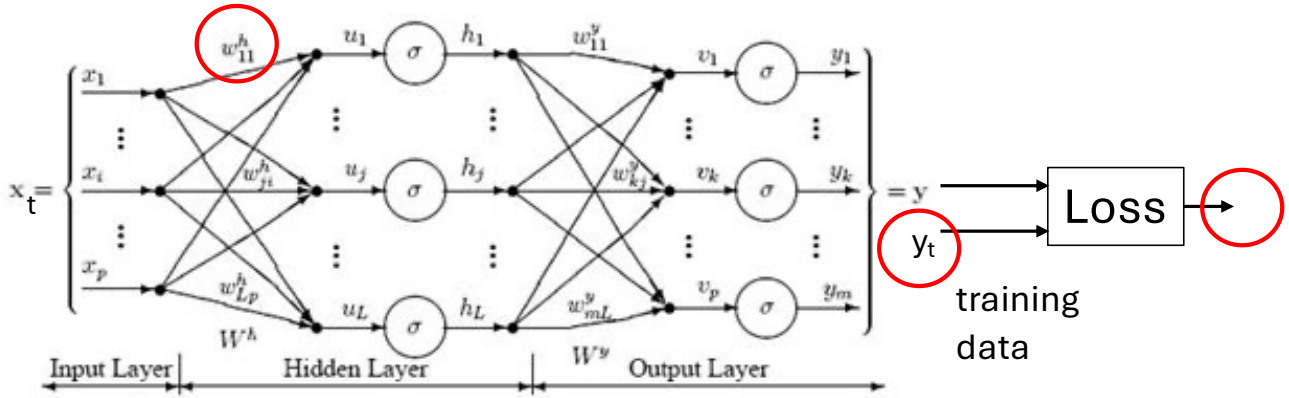




Frank Rosenblatt (Cornell)
Inventor of Perceptron

- **Type**: Inputs: $x_1..x_p$, outputs: $y_1..y_m$ (all $\Re$)
- **Scalar view**:
  - Each edge w_{ij} performs a multiplication with a real-valued **parameter**
  - These values are added followed by non-linear operation σ such as tanh, ReLU etc. (known as **activation functions**)
- **Vector view**:
  - Input and output are vectors
  - Each layer performs a dense matrix vector multiplication followed by pointwise (map) non-linear operation σ

Training MLPs



- Optimization problem *f(W; x,y)*
  - What values of weights w_{ij} minimize loss for training data?
- Gradient descent
  - Compute derivative of loss w.r.t each weight
  - Use an optimizer from earlier discussion
- Usual presentation of gradient computation: chain rule
- Abstraction of MLP and more complex neural networks: ***parameterized programs***

Parameterized program: running example

- Type of desired function: real x real → real
- Training data: set of N 3-tuples {(pi,qi,ti)}
- Model
  - **Composition** of
    - > **base functions** *fi* (may be **nonlinear** such as tanh, sigmoid, sin, cos, …)
    - > **parameters** *Wi: real;* assume no weight sharing so each weight occurs just once (VGGNet: 138 million parameters)
  - Function written as R(w; pi,qi) where w is (w0,w1,w2)
- Parameter optimization
  - Square error for training sample (pi,qi,ti) = $(ti - R(w;pi,qi))^2$
  - Goal: choose (w0,w1,w2) to minimize mean square error (L2 loss)

    Loss(w0,w1,w2) = $\frac{1}{N}\sum_{i=1}^{N}(ti - R(w;pi,qi))^2$

Function R(P,Q) {
*A = W0*P*
B = f0(A,Q)
*C = W1*B*
*D = W2*B*
E = f1(C)
F = f2(D)
R = f3(E,F)
return R}

Parameter optimization

- Find derivatives of Loss wrt *W0,W1,W2:*

Loss(*w0,w1,w2*) = $\frac{1}{N}\sum_{i=1}^{N}(ti - R(w;\ pi, qi))^2$

$$\frac{\partial Loss}{\partial W_0}(w_0, w_1, w_2) = -\frac{2}{N}\sum_{i=1}^{N}(t_i - R(w; p_i, q_i))\frac{\partial R}{\partial W_0}(w; p_i, q_i)$$

$$\frac{\partial Loss}{\partial W_1}(w_0, w_1, w_2) = -\frac{2}{N}\sum_{i=1}^{N}(t_i - R(w; p_i, q_i))\frac{\partial R}{\partial W_1}(w; p_i, q_i)$$

$$\frac{\partial Loss}{\partial W_2}(w_0, w_1, w_2) = -\frac{2}{N}\sum_{i=1}^{N}(t_i - R(w; p_i, q_i))\frac{\partial R}{\partial W_2}(w; p_i, q_i)$$

$$\nabla_W R(w; p_i, q_i)$$

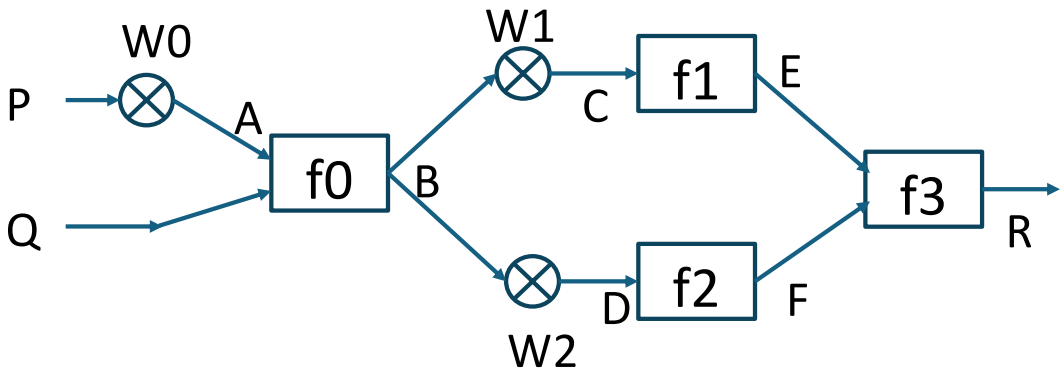
Function R(P,Q) {
*A = W0*P*
B = f0(A,Q)
*C = W1*B*
*D = W2*B*
E = f1(C)
F = f2(D)
R = f3(E,F)
return R}

Derivatives are complex, non-linear functions
so use gradient-descent for parameter optimization

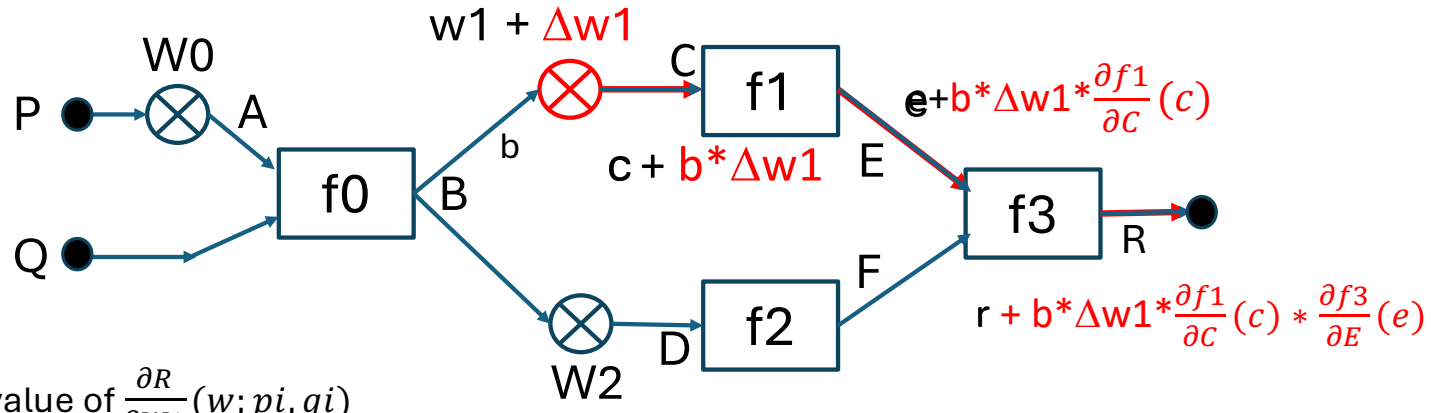
Parameterized program as flow graph

- Useful to represent multiplication by weights differently from functions fi
  - Functions fi are fixed but weights change during training
- Execution (**forward propagation**)
  - Sequential: execute nodes in any topological order
  - Parallel dataflow: node executes when inputs are available
- All values at intermediate points (A,B,C,…) are stored
  - Needed for gradient computations

Function R(P,Q) {
*A = W0*P*
B = f0(A,Q)
*C = W1*B*
*D = W2*B*
E = f1(C)
F = f2(D)
R = f3(E,F)
return R}

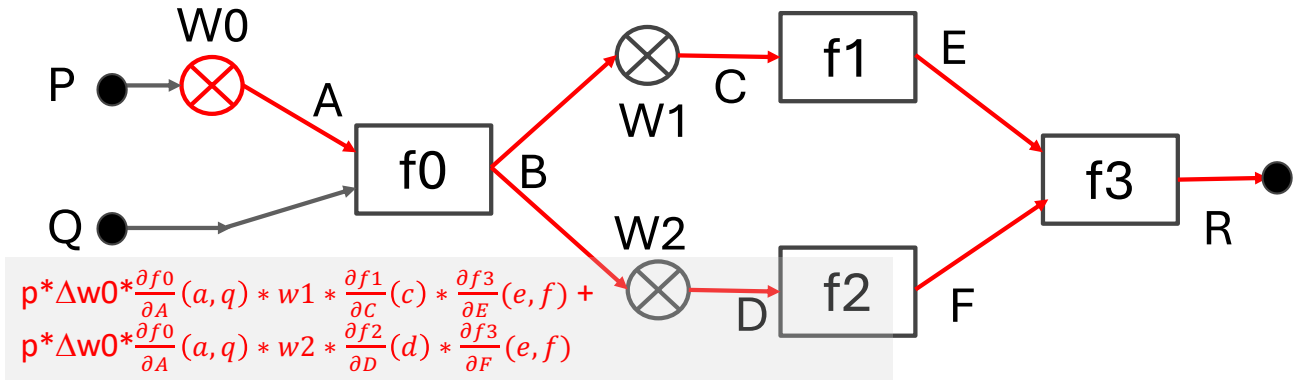


Value of $\frac{\partial R}{\partial W1}(w; pi, qi)$



- To find value of $\frac{\partial R}{\partial W1}(w; pi, qi)$
  - Multiply values of partial derivatives of all vertices on path from W1 to R
  - Multiply result by value of input to W1 (i.e., b)
  - Result: b* $\frac{\partial f1}{\partial C}(c) * \frac{\partial f3}{\partial E}(e)$
  - Compute the product either forwards or backwards along path
- In general, given path $\mathrm{h} : \mathrm{X} \xrightarrow{*} \mathrm{Y}$
  - π(h) = product of derivative values of nodes on h excluding X and Y
  - π(h) = 1 for empty path or if there are no intermediate nodes
- $\frac{\partial R}{\partial W_1}(w; pi, qi) = b * \pi(W1 \xrightarrow{*} R)$

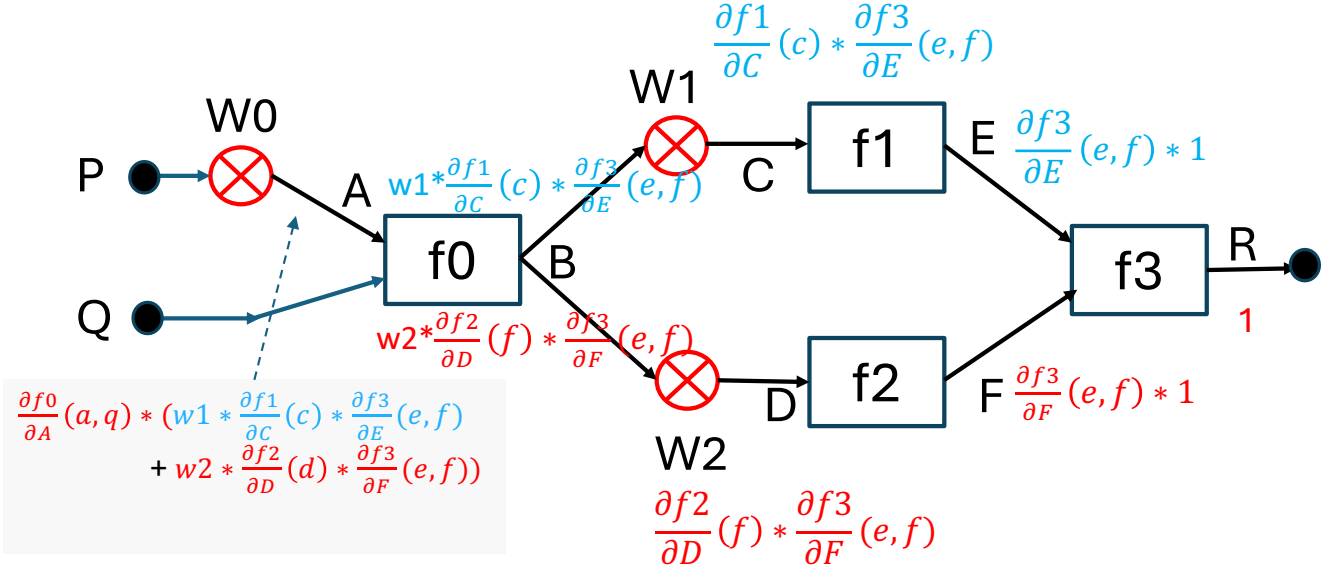
Value of $\frac{\partial R}{\partial W1}(w; pi, qi)$



- In general, there is a DAG from vertex representing weight to the output
  - Example: weight W0
- Value of partial derivative: sum over paths from vertex to output
  - Enumerate all paths from weight to output and add up the contributions of all paths
    - Notation: H(n): set of paths from node n to output

    $\frac{\partial R}{\partial W0}(W; pi, qi) = pi * \sum_{h \epsilon H(W0)} \pi(h)$
  - Intuition: derivatives make this a linear problem, so *superposition of paths* works
- Problems:
  - Treats DAG like tree so could do exponential computation in size of DAG. More efficient solution?
  - What order should we compute $\frac{\partial R}{\partial (W0)}(w; pi, qi)$, $\frac{\partial R}{\partial (W1)}(w; pi, qi)$ and $\frac{\partial R}{\partial (W2)}(w; pi, qi)$?

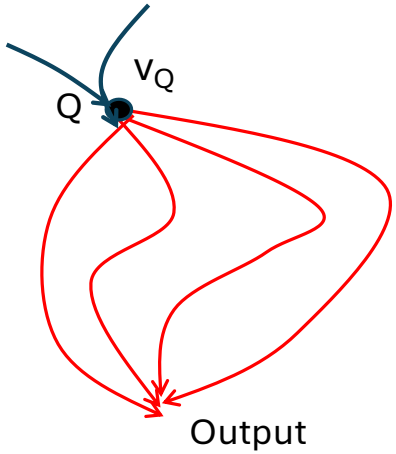
Efficient computation of all derivatives

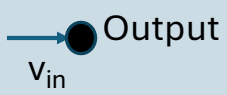
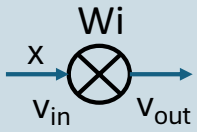
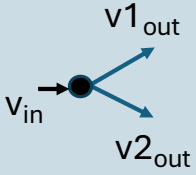
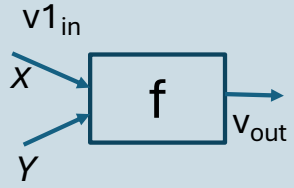


- Compute derivative of output wrt *every* variable/edge ($\frac{\partial R}{\partial A}, \frac{\partial R}{\partial C}, \frac{\partial R}{\partial F}$..)
  - Real number on each edge
  - Derivatives of output wrt weights can be computed from this
- Traverse DAG in reverse topological order of variables for computation
  - $\frac{\partial R}{\partial R} = 1$
  - **Transfer functions** to propagate derivative from function output to its inputs

Summary of derivatives computation

- At each point Q, compute v_Q: $\Re$ where
  $v_Q = \frac{\partial Output}{\partial Q}(W; pi, qi) = \sum_{h \epsilon H(Q)} \pi(h)$
- Small tweak to handle weight-sharing
- Called back-propagation in ML literature
- Vanishing and exploding gradients
  - Multiplying sequence of small or large numbers



 $v_{in} = 1$	 $v_{in} = wi * v_{out}$ and $\frac{\partial\ Output}{\partial Wi} = v_{out} * x$
 $v_{in} = (v1_{out} + v2_{out})$	 $v1_{in} = \frac{\partial f}{\partial X}(x, y) * v_{out}$

Back to running example

- Optimization problem: choose Wi values to minimize loss

  Loss(w0,w1,w2) = $\frac{1}{N}\sum_{i=1}^{N}(ti - R(w;pi,qi))^2$

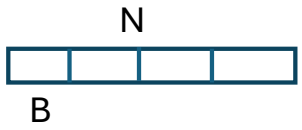
$$\frac{\partial Loss}{\partial W_0}(w_0,w_1,w_2) = -\frac{2}{N}\sum_{i=1}^{N}(t_i - R(w;p_i,q_i))\frac{\partial R}{\partial W_0}(w;p_i,q_i)$$

Initialize weights to random values
for #epochs do {
  GradientVector = 0
  for each training sample (pi,qi,ti) do {
    perform forward propagation and compute
    perform backpropagation and compute weight derivatives
    update GradientVector with products}
  scale GradientVector by -2/N
  use GradientVector to update weights using gradient descent step
}

Function R(P,Q) {
*A = W0*P*
B = f0(A,Q)
*C = W1*B*
*D = W2*B*
E = f1(C)
F = f2(D)
R = f3(E,F)
return R}

Variations on theme (II)

- How many training data samples should we use before updating weights?



  - (Batch) Gradient-descent: use entire training data set to compute gradient.
    - Disadvantage: learn only after all training data has been processed.
  - (Mini-)Batching: divide training data into subsets of size B, update weights after each subset is processed and use as initial weights for next subset.
    - Disadvantage: choosing B? 50-256 is common.
    - Useful to randomize training data set
  - Stochastic gradient-descent (SGD): B= 1. Can be noisier than previous approaches.
- Initialization: we assumed random initialization
  - Can also exploit prior (domain knowledge)
  - He initialization
  - Xavier (Glorot) initialization
  - Good discussion: https://arxiv.org/abs/1704.08863

Variations on theme (III)

Other loss functions

- Regularization: add a term that penalizes weights that are not “desirable”

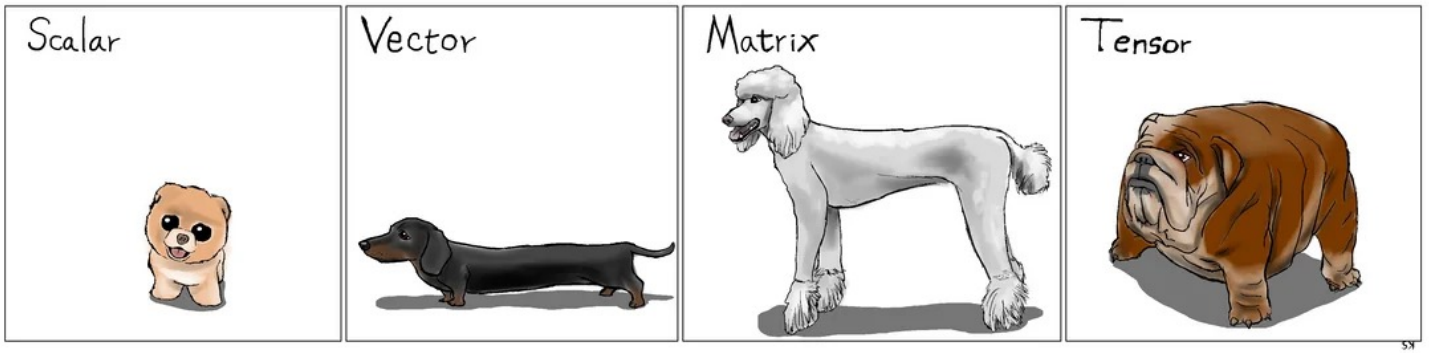
  Example: Loss(w0,w1,w2) = $\frac{1}{N}\sum_{i=1}^{N}(ti - R(W; pi, qi))^2 + \lambda \left|\left|W\right|\right|^2$

  Called *ridge regularization*: penalizes large weight values

  Many other forms of regularization

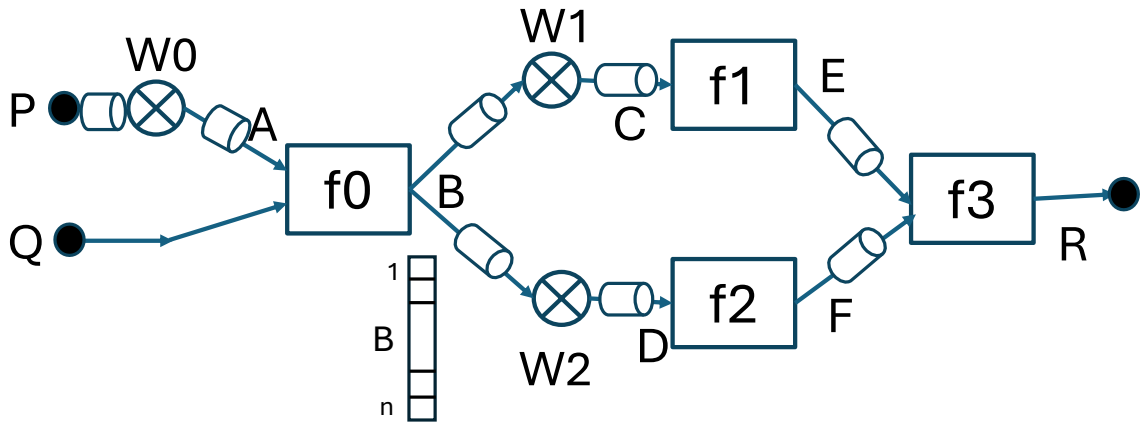
- Loss for classification problems
  - KL-divergence, cross-entropy loss (see later)

Generalization to vectors, matrices, tensors



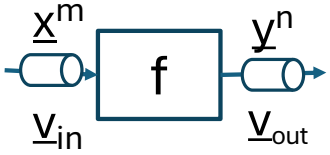
How to understand data with dogs
Karl Stratos (Reddit post)

Generalization to vectors (matrices, tensors are similar)



- Programs
  - Inputs can be scalars or vectors, but output is still a scalar (loss)
  - Weight matrices (need not be square)
  - Function type: vector → vector
- Compute gradients instead of derivatives
  - Variable is vector of size n → gradient of output wrt this vector is also vector of size n
    - Intuition: each dimension of gradient vector is derivative of output wrt value in that dimension
  - Transfer functions: Jacobians instead of derivatives
- Useful to know matrix derivatives notation

Transfer functions



$$J_f = \begin{pmatrix} \frac{\partial Y_1}{\partial X_1} & \frac{\partial Y_2}{\partial X_1} & \cdots & \frac{\partial Y_n}{\partial X_1} \\ \frac{\partial Y_1}{\partial X_2} & \frac{\partial Y_2}{\partial X_2} & \cdots & \frac{\partial Y_n}{\partial X_2} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial Y_1}{\partial X_m} & \frac{\partial Y_2}{\partial X_m} & \cdots & \frac{\partial Y_n}{\partial X_m} \end{pmatrix} \quad \underline{v}_{out} = \begin{pmatrix} \frac{\partial Output}{\partial Y_1} \\ \frac{\partial Output}{\partial Y_2} \\ .... \\ \frac{\partial Output}{\partial Y_n} \end{pmatrix}$$

- General function f
  - $\underline{v}_{in} = J_f * \underline{v}_{out}$
- Linear function W
  - $J_f = W^T$
  - $\underline{v}_{in} = W^T * \underline{v}_{out}$
  - Derivatives of output wrt weights in W (will be a matrix)

$$y_1 = w_{11}x_1 + w_{12}x_2 + \ldots + w_{1m}x_m$$
$$y_2 = w_{21}x_1 + w_{22}x_2 + \ldots + w_{2m}x_m$$
$$\ldots\ldots$$

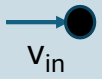
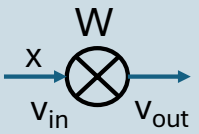
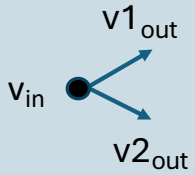
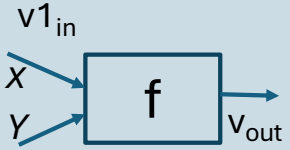
$$\frac{\partial(\text{Output})}{\partial W} = \underline{v}_{out} \otimes \underline{x} \qquad (\otimes \text{ is outer-product})$$

$$\frac{\partial(\text{Output})}{\partial W}(i,j) = \underline{v}_{out}(i) * \underline{x}(j)$$

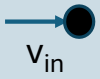
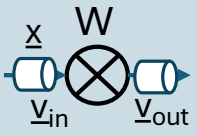
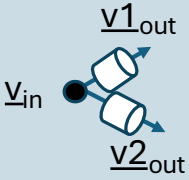
(If w(i,j) changes a small amount, how much does the output change?)

Transfer Functions (scalar and vector)

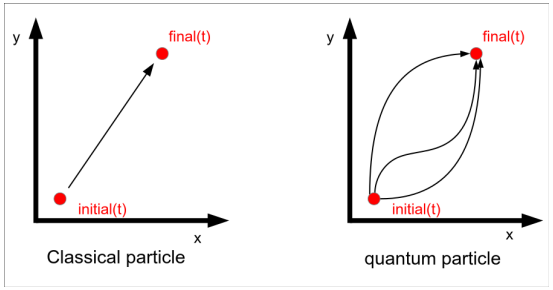
Scalar case

 $v_{in} = 1$	 $v_{in} = w * v_{out}$ $\frac{\partial\,(Output)}{\partial Wi} = v_{out} * x$
 $v_{in} = (v1_{out} + v2_{out})$	 $v1_{in} = \frac{\partial f}{\partial X}(x, y) * v_{out}$

Vector case

 $v_{in} = 1$	 $\underline{v}_{in} = W^{T} * \underline{v}_{out}$ $\frac{\partial\,(Output)}{\partial W} = \underline{v}_{out} \otimes \underline{x}$
 $\underline{v}_{in} = (\underline{v1}_{out} + \underline{v2}_{out})$	 $\underline{v}_{in} = J_f * \underline{v}_{out}$

Summary



Path integral formulation
of quantum mechanics (Feynman)

Abstraction for neural networks: parameterized programs

Gradient computation

- Abstractly (what?): sum over paths (important idea in Physics)
- Efficient computation (how?): compositional algorithm on dataflow graph representation

Handles complex nonlinear functions like sin(x), sin(cos(x)), $e^{\cos(x)}$

Handles complex neural networks with weight-sharing and irregular interconnections (such as “skip connections”) smoothly

With small caveat, handles conditional and loops as well

Example of backward dataflow analysis used in compilers