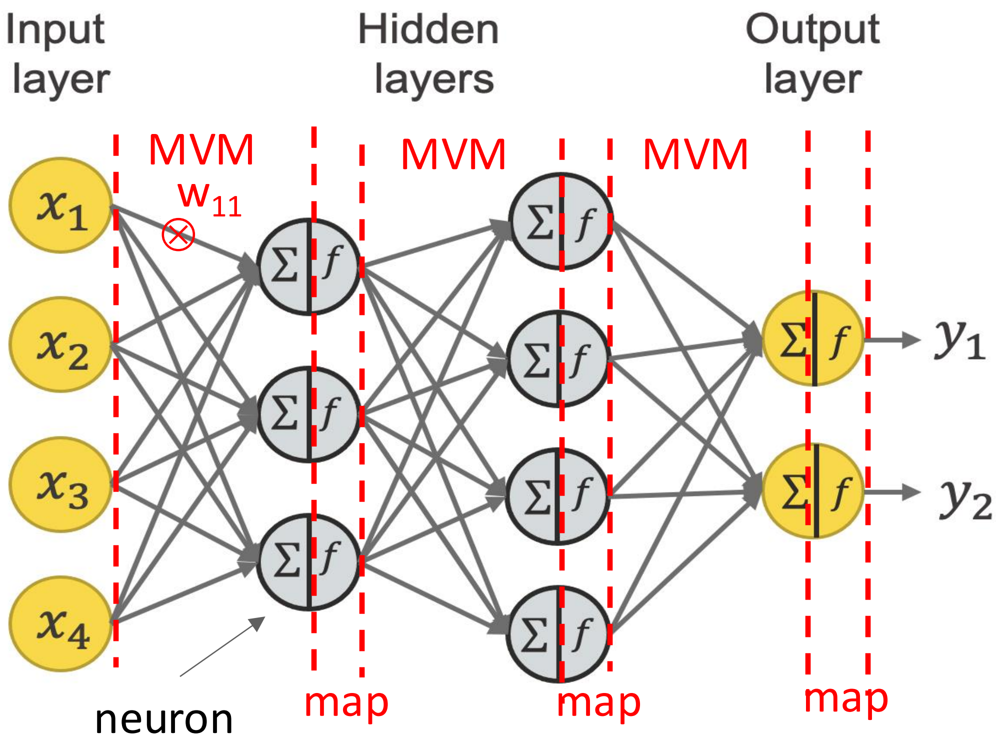
Back Propagation Made Easy

Keshav Pingali
The University of Texas at Austin

Organization

- Problem
  - Parameter estimation in neural networks
  - Non-linear optimization problem
  - Solved using gradient descent
  - Usual presentations of gradient descent are hard to understand and difficult to extend to irregular network connections
- Parameterized programs
  - Abstraction for neural networks
- Gradient computation in parameterized programs
  - Backward dataflow analysis
- Advantages
  - Compositional algorithm for gradient computation
  - Handles weight-sharing
  - Easy to extend to skip connections and other irregular networks
  - Natural extension to higher dimensional data such as tensors

Multilayer Perceptron (MLP) Example



Frank Rosenblatt (Cornell)
Inventor of Perceptron

- **Type**: Inputs: $x_1..x_4$, outputs: y_1,y_2 (all $\Re$)
- **Scalar view**:
  - Each edge performs a multiplication with a real-valued **parameter**
  - Each neuron adds its input values and applies a non-linear operation *f* such as tanh, ReLU etc. (known as **activation functions**)
- **Vector view**:
  - Each layer performs a dense matrix vector multiplication
  - Followed by pointwise (map) non-linear operation *f*
- 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
  - **Composed from**
    - **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)
- Notation: capital letters for variable names, small letters for variable values
  - 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

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

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

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

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

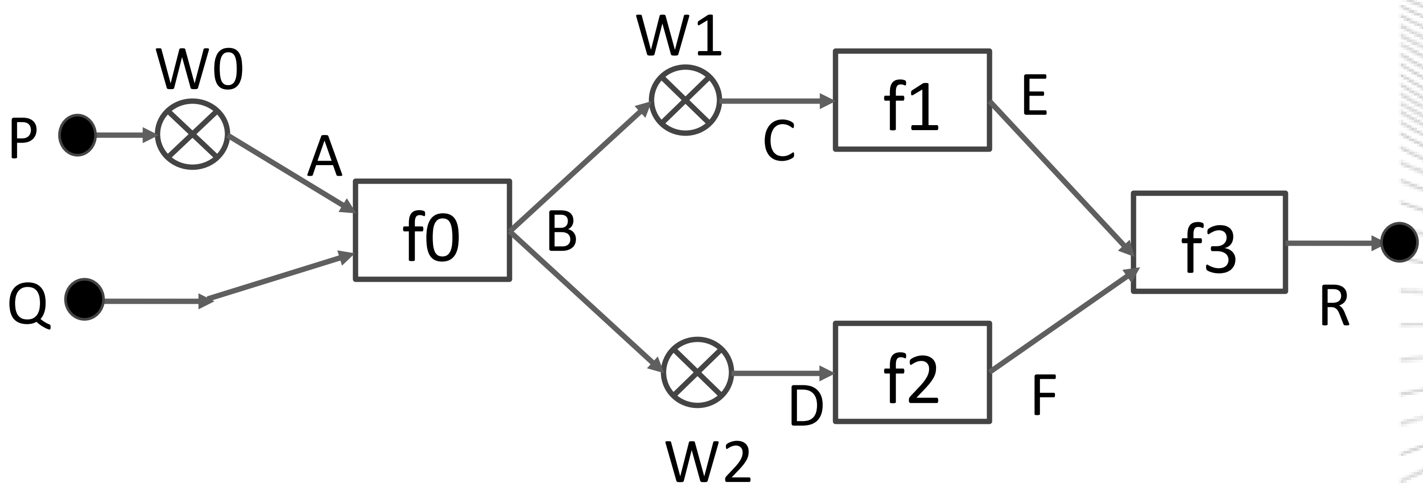
$$\nabla_W R(w; p_i, q_i)$$

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 complicated, non-linear functions so use gradient-descent for parameter optimization

Parameterized program as flow graph

```
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}
```

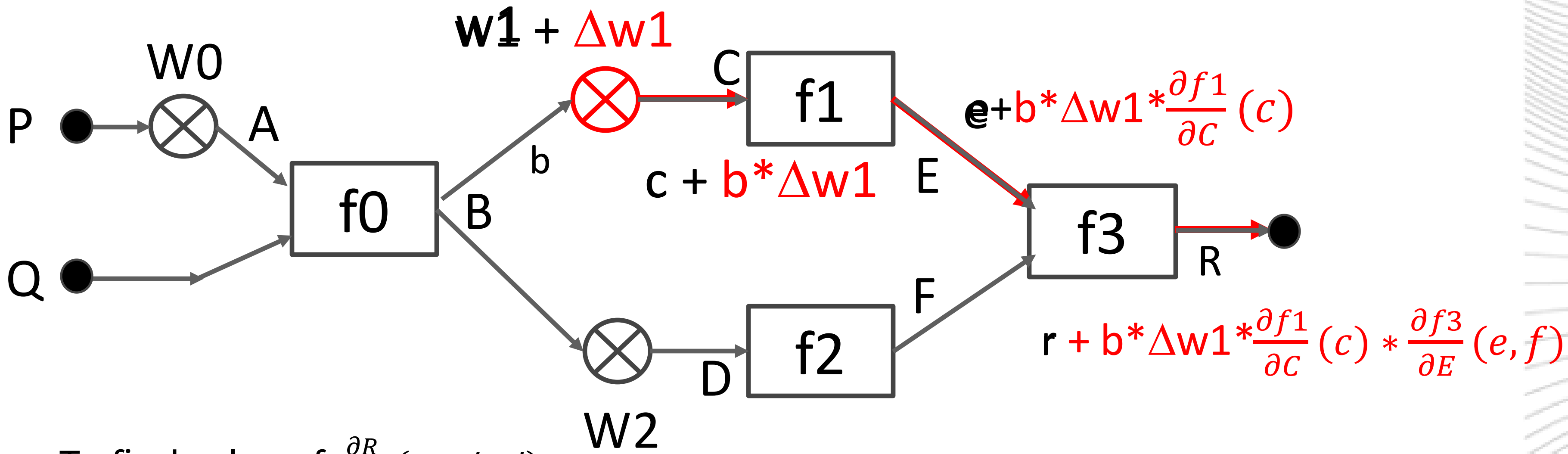


- Useful to represent multiplication by weights differently from functions fi
  - Functions fi are fixed but weights change during training

- Execution models
  - Sequential:
    - Execute nodes sequentally in any topological order
  - Parallel:
    - Asynchronous dataflow: node executes when inputs are available
    - **Forward propagation**: level-by-level schedule of vertices

- All values at intermediate points (A,B,C,...) are stored
  - Needed for gradient computations

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



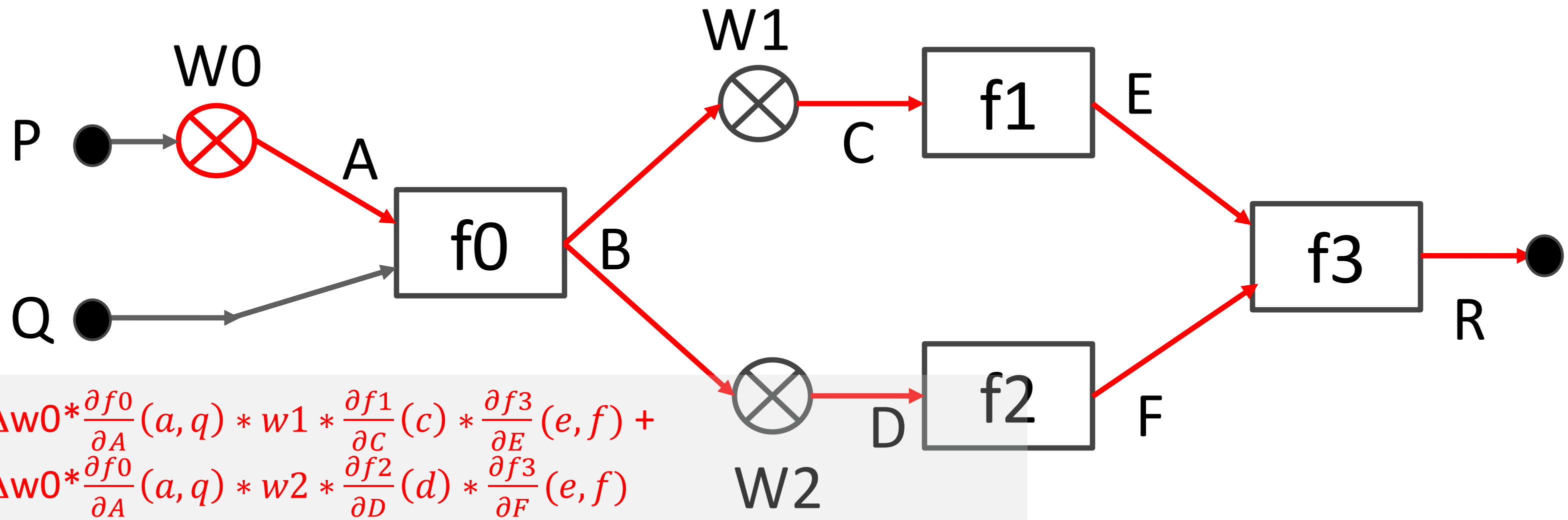
- To find value of $\frac{\partial R}{\partial W1}(w; pi, qi)$
  - Multiply values of partial derivatives of all vertices on path from W1 to R
  - Multiply result by value of input to W1 (i.e., b)
  - Result: b*$\frac{\partial f1}{\partial C}(c) * \frac{\partial f3}{\partial E}(e, f)$
  - Compute the product either forwards or backwards along path

- In general, given path $\rho : X \xrightarrow{*} Y$
  - $\pi(\rho)$ = product of derivatives of nodes on path excluding X and Y (*path derivative*)

    = 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 $\frac{\partial R}{\partial W0}(w; pi, qi)$



$p * \Delta w0 * \frac{\partial f0}{\partial A}(a, q) * w1 * \frac{\partial f1}{\partial C}(c) * \frac{\partial f3}{\partial E}(e, f) +$

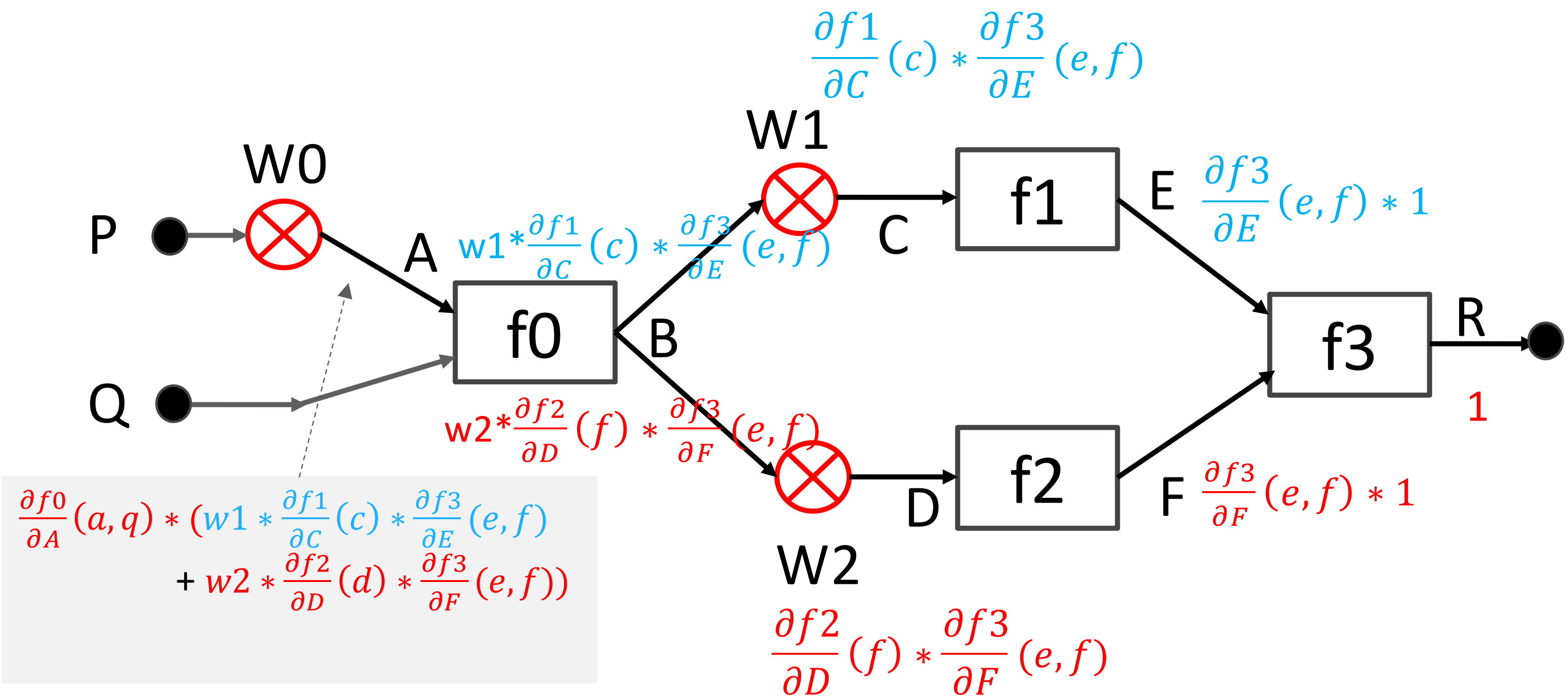
$p * \Delta w0 * \frac{\partial f0}{\partial A}(a, q) * w2 * \frac{\partial f2}{\partial D}(d) * \frac{\partial f3}{\partial F}(e, f)$

- In general, there is a DAG from weight to the output
- Value of partial derivative:
  - Enumerate all paths from weight to output and add up the contributions of all paths

    $\frac{\partial R}{\partial W_0}(w_0; pi, qi) = pi * \sum_{\rho \in \mathcal{P}(W_0)} \pi(\rho)$

    (where $\mathcal{P}(W_0)$ is the set of paths from W_0 to exit)
  - 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



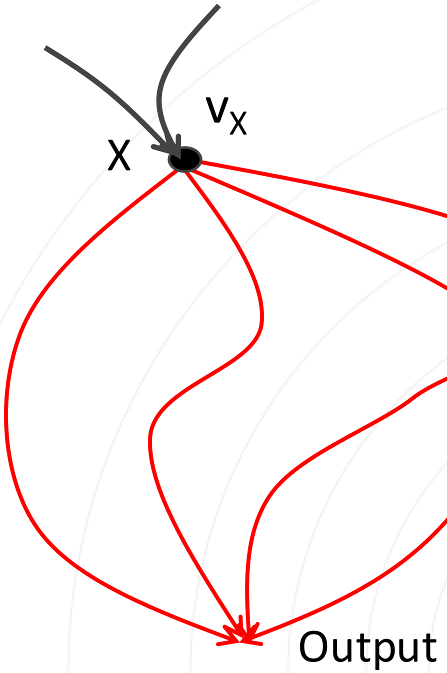
- Compute derivative of output wrt *every* variable/edge ($\frac{\partial R}{\partial A}, \frac{\partial R}{\partial C}, \frac{\partial R}{\partial F}..$)
  - Real number on each edge
  - Derivatives of output wrt weights can be computed from this
- Traverse DAG in reverse topological order of variables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

Summary of derivatives computation

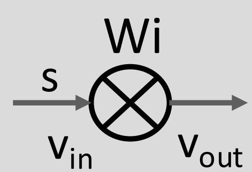
- At each point X, compute v_X: $\Re$ where

$$v_x = \frac{\partial Output}{\partial X}(w; pi, qi) = \sum_{\rho \in \mathcal{P}(X)} \pi(\rho)$$

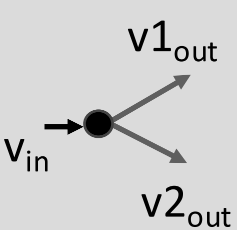
- Small tweak to handle weight-sharing
- Called back-propagation in ML literature



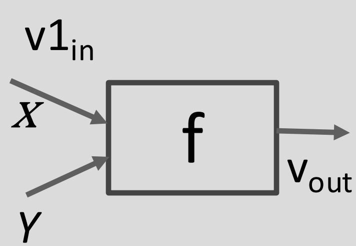
$v_{in} = 1$



$v_{in} = wi * v_{out}$ and $\frac{\partial\ Output}{\partial Wi} = v_{out} * s$



$v_{in} = (v1_{out} + v2_{out})$



$v1_{in} = \frac{\partial f}{\partial X}(x, y) * v_{out}$

Back to running example

- Optimization problem: choose Wi values to minimize loss

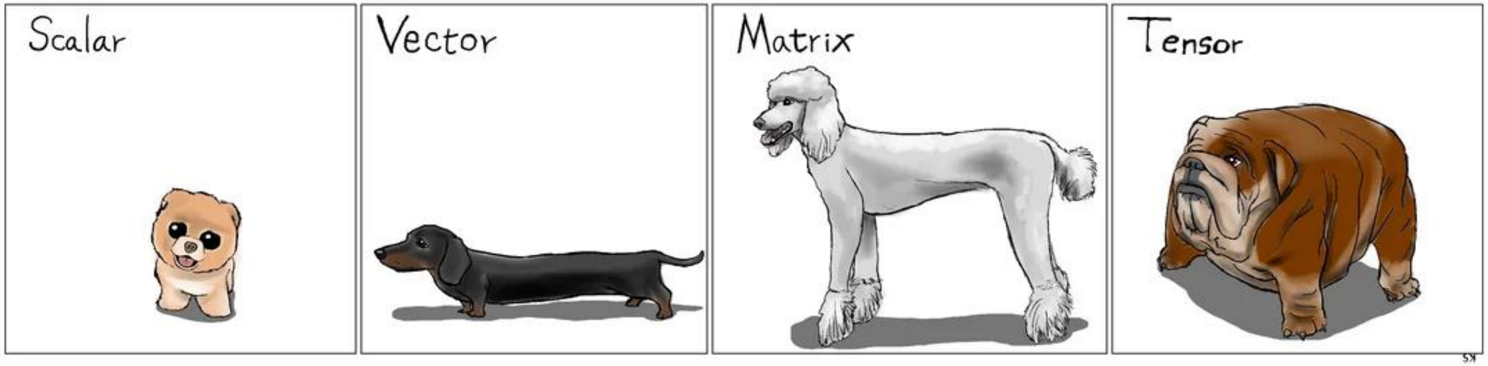
  Loss(w0,w1,w2) = $\frac{1}{N}\sum_{i=1}^{N}(ti - R(w;pi,qi))^2$

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

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
}

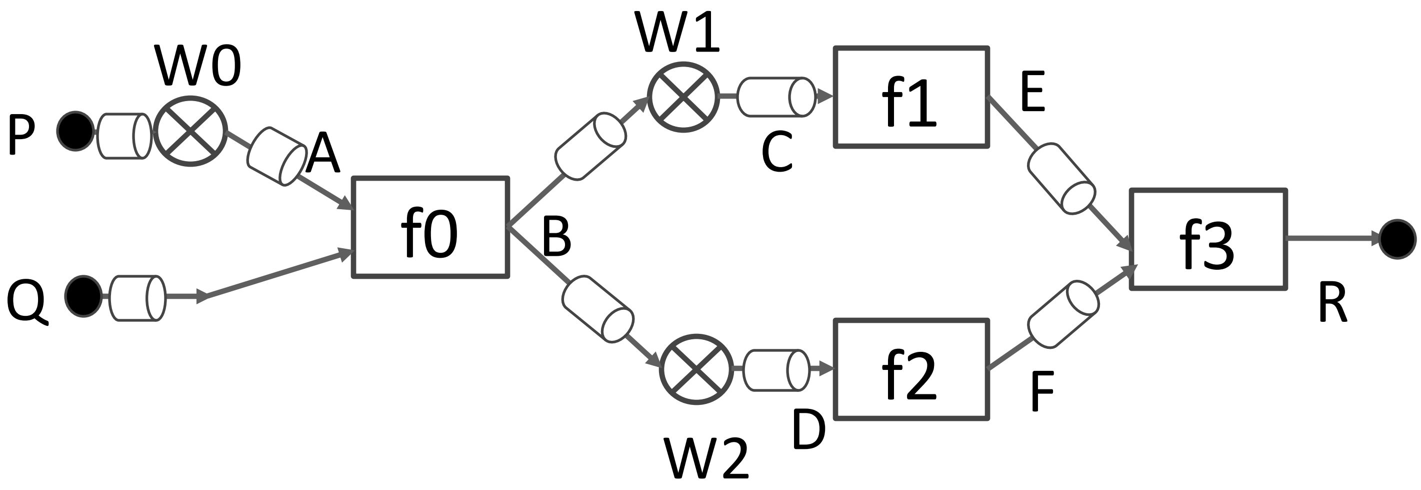
Function R(P,Q) {
*A = W0*P*
B = f0(A,Q)
*C = W1*B*
*D = W2*B*
E = f1(C)
F = f2(D)
R = f3(E,F)
return R}

Generalization to vectors, matrices, tensors



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

Generalization to vectors (matrices, tensors are similar)

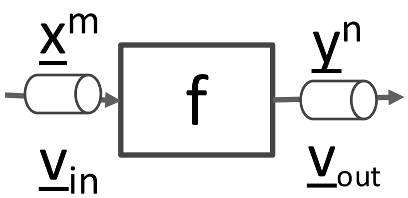


- Programs
  - Inputs can be scalars or vectors, but output is still a scalar (loss)
  - Weights ➔ Weight matrices
    - Weight matrix need not be square
  - Function type: vector → vector, vector x vector → vector, etc.

- 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



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

- 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

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

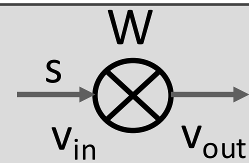
$$\frac{\partial(\mathrm{Output})}{\partial W} = \underline{v}_{out} \otimes \underline{x}$$ (⊗ is outer-product)

$$\frac{\partial(\mathrm{Output})}{\partial W}(i,j) = \underline{v}_{out}(i) * \underline{x}(j)$$ (If w(i,j) changes a small amount, how much does the output change?)

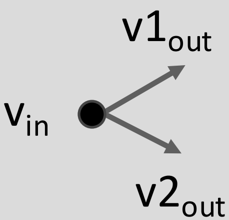
Transfer Functions (scalar and vector)

Scalar case

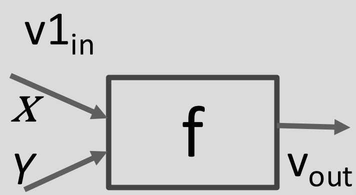
$v_{in} = 1$



$v_{in} = w * v_{out}$ $\qquad \frac{\partial\,(Output)}{\partial Wi} = v_{out} * s$



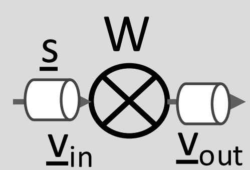
$v_{in} = (v1_{out} + v2_{out})$



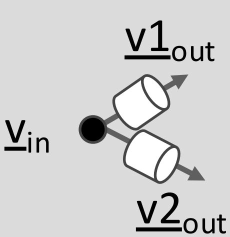
$v1_{in} = \frac{\partial f}{\partial X}(x, y) * v_{out}$

Vector case

$v_{in} = 1$



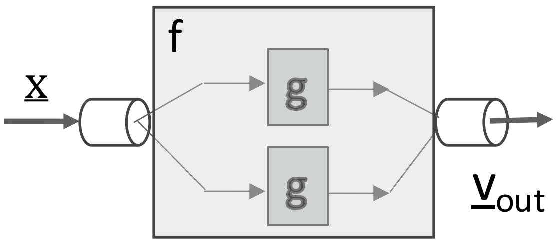
$\underline{v}_{in} = W^{T} * \underline{v}_{out}$ $\qquad \frac{\partial\,(Output)}{\partial W} = \underline{v}_{out} \otimes \underline{s}$



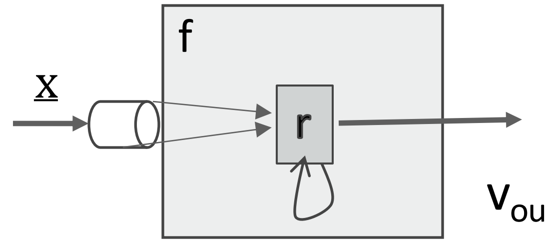
$\underline{v}_{in} = (\underline{v1}_{out} + \underline{v2}_{out})$

$\underline{v}_{in} = J_f * \underline{v}_{out}$

Important special cases



map



reduce

- f: map (g: $\Re \rightarrow \Re$)
  - J_f = diagonal matrix with $J_f(i,i) = g'(x(i))$

$$\underline{\mathbf{v}}_{in} = \begin{pmatrix} g'(\underline{\mathbf{x}}(1)) & 0 \\ 0 & g'(\underline{\mathbf{x}}(2)) \end{pmatrix} \underline{\mathbf{v}}_{out}$$

- f: reduce (r: $\Re \mathrm{x} \Re \rightarrow \Re$)
  - Called **pooling** in ML literature
  - r = +, *, ...
    - > J_f = $(1,1,\ldots,1)^T$ for +
  - r = max, min, ...
    - > If max(x) = x(j), J_f = I(j) (where I(j) is the indicator vector with 1 in j^{th} position)

Summary

- Standard presentations of gradients and back propagation
  - Biological metaphors like neurons and synapses come in the way
  - Properties of activation functions are distraction: enough to know we can compute value and gradient at any point
- Presentation in this lecture
  - **Abstraction** for neural networks: parameterized programs
  - Gradient computation
    - Abstractly (what?): **sum over paths** (sum of products)
    - Efficient computation (how?): compositional algorithm on dataflow graph representation
  - Handles complex neural networks with weight-sharing and irregular interconnections (such as “skip connections”) smoothly