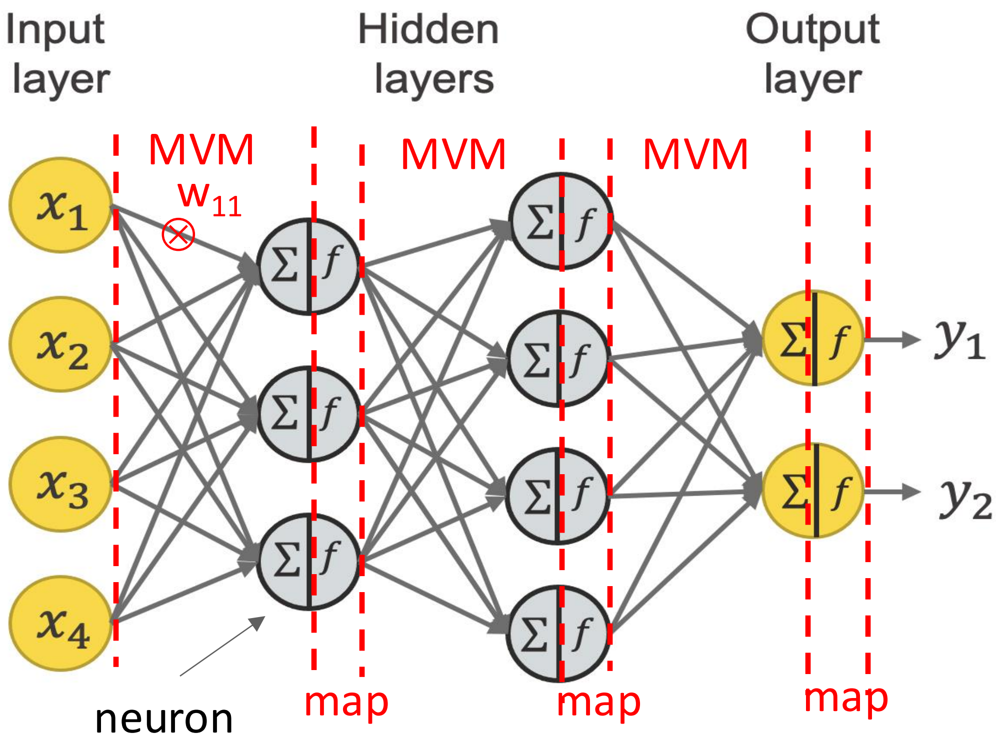
Gradient Computation in Neural Networks and Kolmogorov-Arnold Networks

Keshav Pingali and Lain Mustafaoglu
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
- Extension to Kolmogorov-Arnold Networks (KANs) and Recurrent Neural Networks (RNNs)

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*
- Gradient computation difficult to understand
- 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 invocation 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*

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

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

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

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

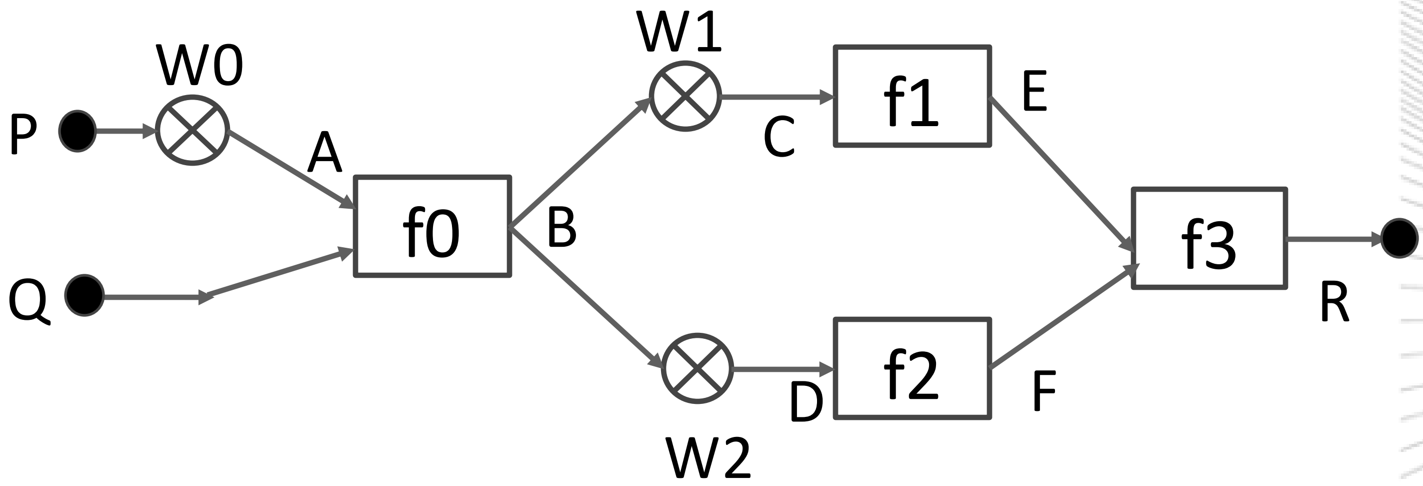
$$\nabla_W R(w; p_i, q_i)$$

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
-> Focus on gradient computation

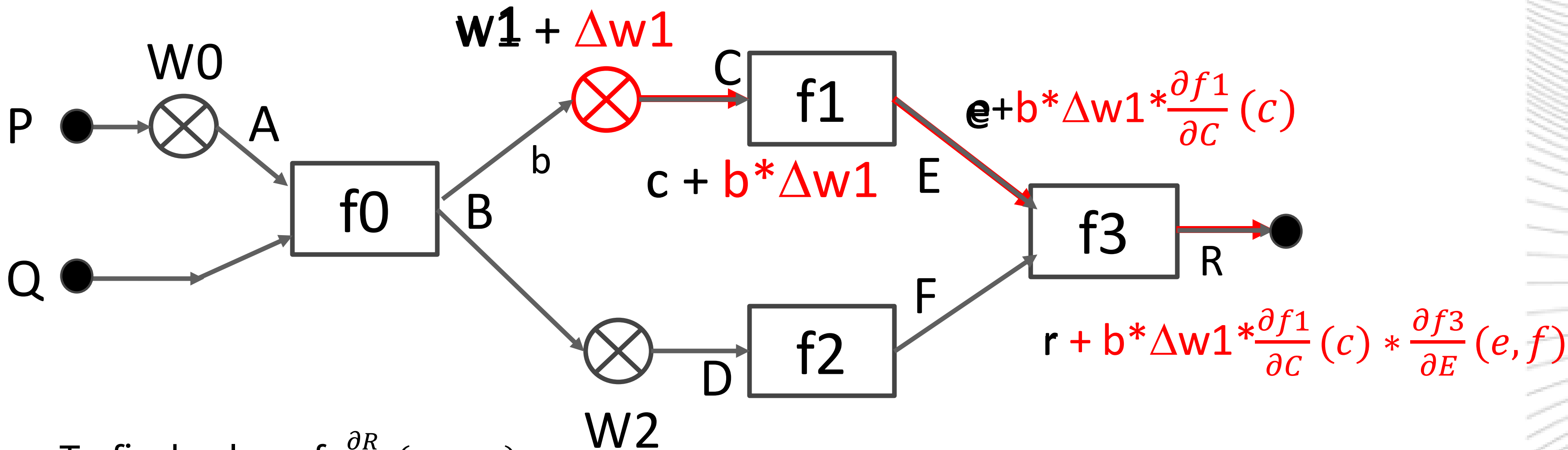
Parameterized program as flow graph

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
  - Weights change during training
- Forward Propagation: execute nodes in any topological order
- All variable values are stored
  - Needed for gradient computations

Value of $\frac{\partial R}{\partial W1}(w; p, q)$



- To find value of $\frac{\partial R}{\partial W1}(w; p, q)$
  - 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

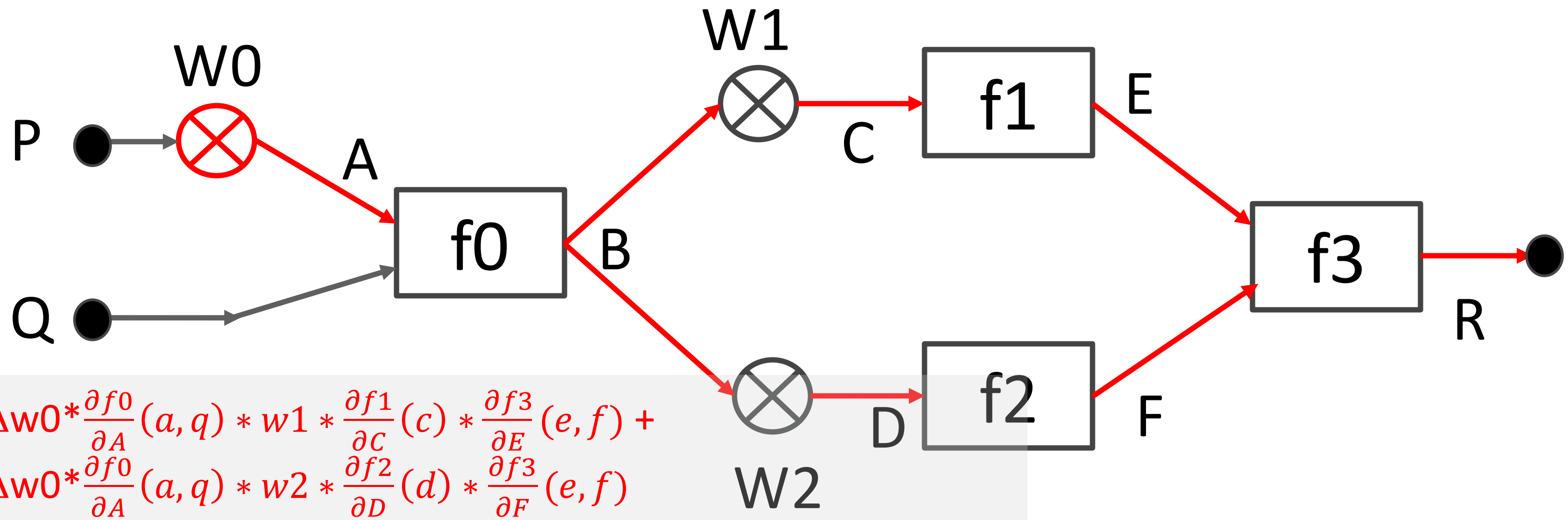
- In general, for path $\rho : \mathrm{X} \xrightarrow{*} \mathrm{Y}$ *path derivative* $\pi(\rho)$

  = product of derivatives of nodes on path excluding X and Y

  = 1 for empty path or if there are no intermediate nodes

- $\frac{\partial R}{\partial W_1}(w; p, q) = b * \pi(W1 \xrightarrow{*} R)$

Value of $\frac{\partial R}{\partial W0}(w; p, q)$



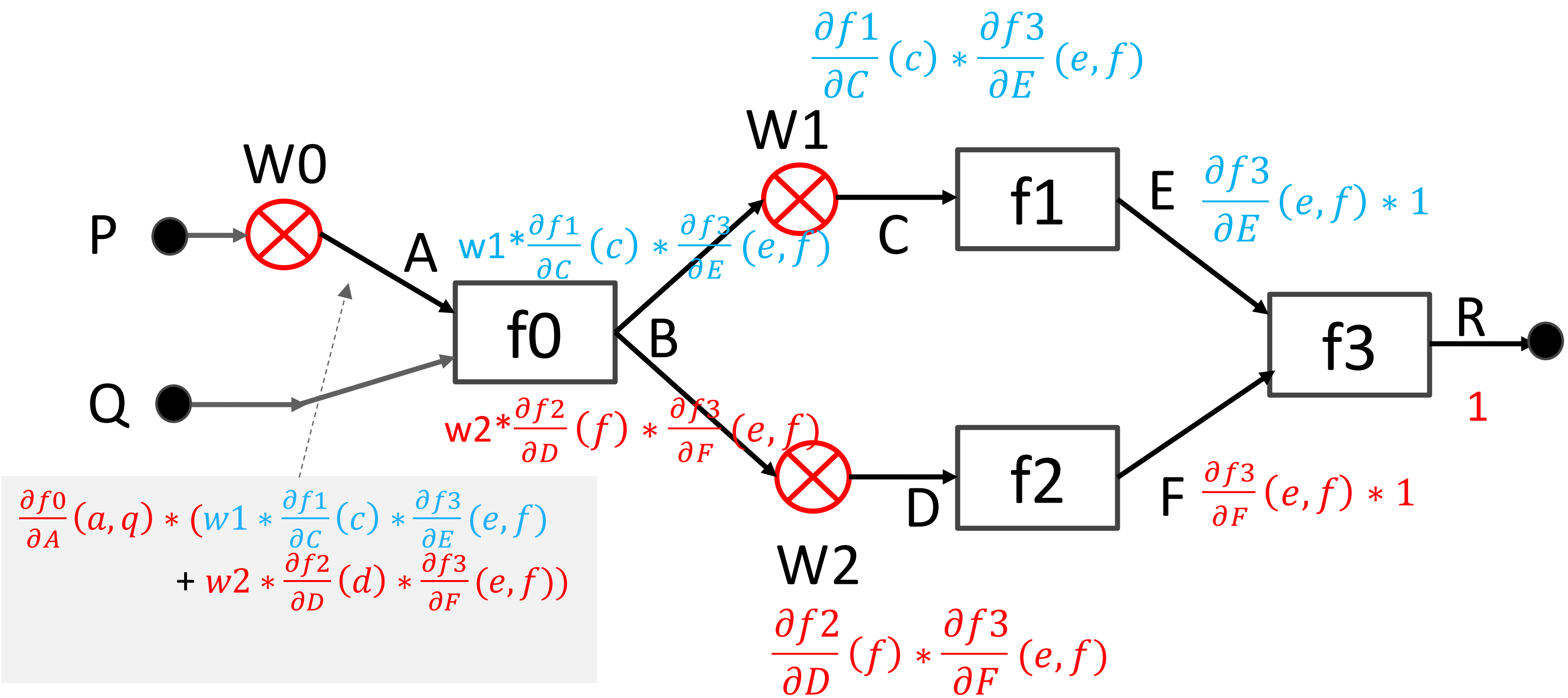
p*Δw0*$\frac{\partial f0}{\partial A}(a,q) * w1 * \frac{\partial f1}{\partial C}(c) * \frac{\partial f3}{\partial E}(e,f)$ +
p*Δw0*$\frac{\partial f0}{\partial A}(a,q) * w2 * \frac{\partial f2}{\partial D}(d) * \frac{\partial f3}{\partial F}(e,f)$

- 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

    $$\frac{\partial R}{\partial W_0}(w; p, q) = p * \sum_{\rho \in \mathcal{P}(W_0)} \pi(\rho)$$
    (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 $\frac{\partial R}{\partial W0}(w; p, q)$, $\frac{\partial R}{\partial W1}(w; p, q)$ and $\frac{\partial R}{\partial W2}(w; p, q)$?

Efficient computation of all derivatives



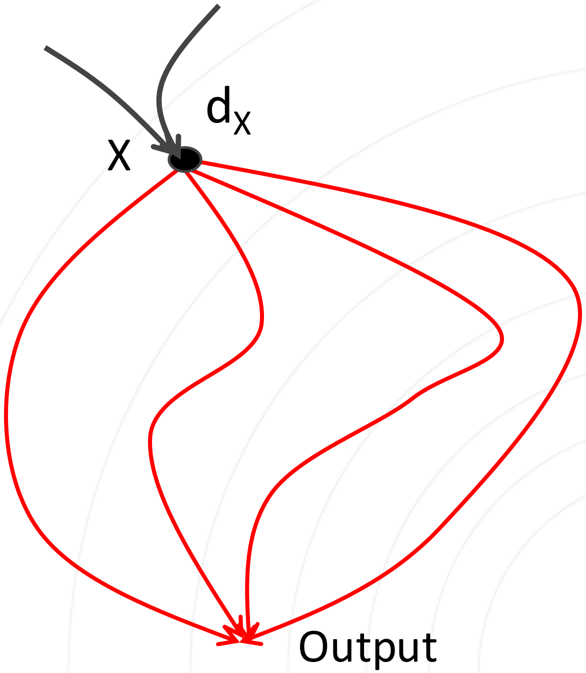
- Compute derivative of output wrt *every* variable/edge ($\frac{\partial R}{\partial A}, \frac{\partial R}{\partial C}, \frac{\partial R}{\partial F}..$)
  - Real number on each edge
  - Derivatives of output wrt weights can be computed from this
- Traverse DAG in reverse topological order of variables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

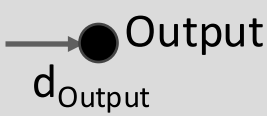
Summary of derivatives computation

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

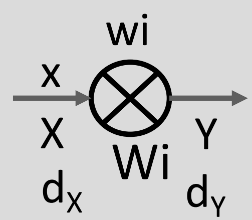
$$d_x = \frac{\partial\ Output}{\partial X}(w; p, q) = \sum_{\rho \in \mathcal{P}(X)} \pi(\rho)$$

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

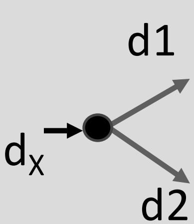




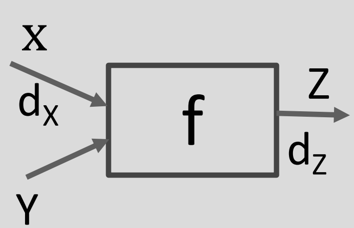
$d_{Output} = 1$



$d_X = wi*d_Y$ and $\frac{\partial\ Output}{\partial Wi} = d_Y*x$



$d_X = d1+d2$



$d_X = \frac{\partial f}{\partial X}(x, y)*d_Z$

(chain rule)

Back to running example

- Optimization problem: choose Wi values to minimize loss

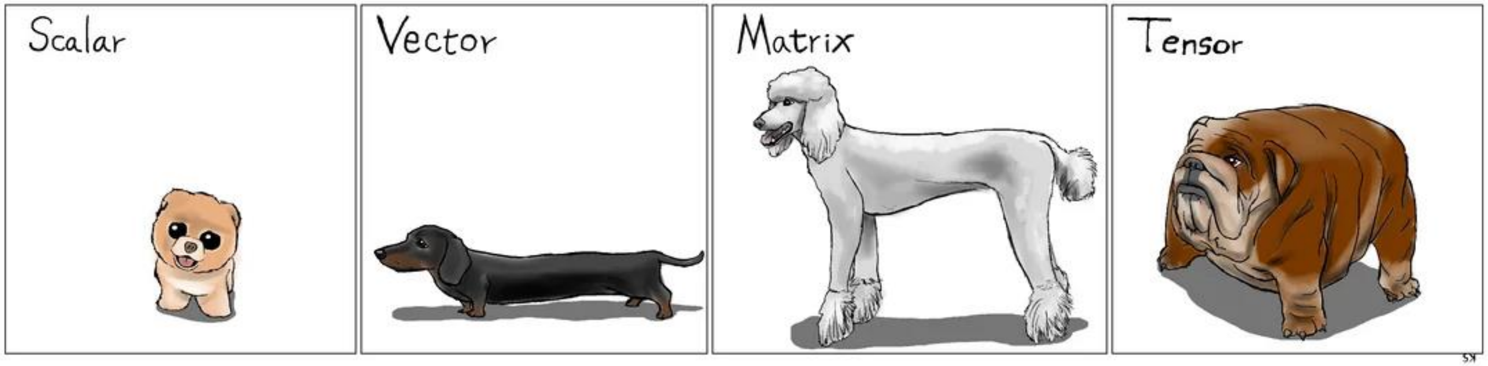
  Loss(w0,w1,w2) = $\frac{1}{N}\sum_{i=1}^{N}(ti - R(w; pi, qi))^2$

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

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}

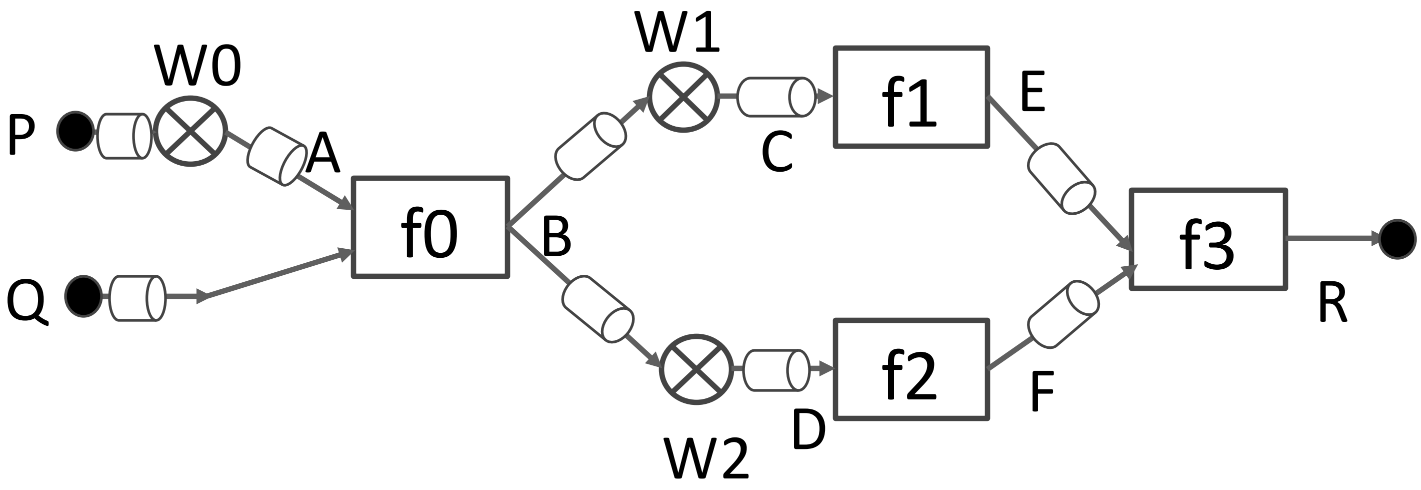
Initialize weights to random values
for #epochs do {
 GradientVector = 0
 for each training sample (pi,qi,ti) do {
  perform forward propagation and compute
  perform backpropagation and compute weight derivatives
  update GradientVector with products}
 scale GradientVector by -2/N
 use GradientVector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

Generalization to vectors, matrices, tensors



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

Generalization to vectors (matrices, tensors are similar)

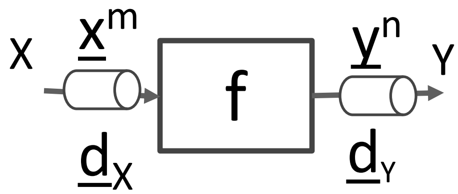


- Programs
  - Inputs can be scalars or vectors, but output is still a scalar (loss)
  - Weights ➔ Weight matrices
    - > Weight matrix need not be square
  - Function type: vector → vector, vector x vector → vector, etc.

- 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



$$J_f = \begin{pmatrix} \frac{\partial Y_1}{\partial X_1} & \frac{\partial Y_2}{\partial X_1} & \cdots & \frac{\partial Y_n}{\partial X_1} \\ \frac{\partial Y_1}{\partial X_2} & \frac{\partial Y_2}{\partial X_2} & \cdots & \frac{\partial Y_n}{\partial X_2} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial Y_1}{\partial X_m} & \frac{\partial Y_2}{\partial X_m} & \cdots & \frac{\partial Y_n}{\partial X_m} \end{pmatrix} \qquad \underline{d}_Y = \begin{pmatrix} \frac{\partial \text{Output}}{\partial Y_1} \\ \frac{\partial \text{Output}}{\partial Y_2} \\ \cdots\cdot \\ \frac{\partial \text{Output}}{\partial Y_n} \end{pmatrix}$$

- General function f
  - $\underline{d}_X = J_f * \underline{d}_Y$
- Linear function W
  - $J_f = W^T$
  - $\underline{d}_X = W^T * \underline{d}_Y$
  - Derivatives of output wrt weights in W

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

$$\frac{\partial \text{ Output}}{\partial W} = \underline{d}_Y \otimes \underline{x}$$

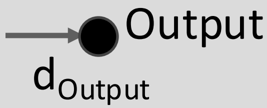
(⊗ is outer-product)

$$\frac{\partial \text{ Output}}{\partial W}(i,j) = \underline{d}_Y(i) * \underline{x}(j)$$

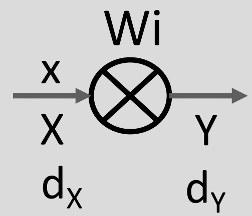
(If w(i,j) changes a small amount, how much does the output change?)

Transfer Functions (scalar and vector)

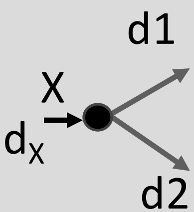
Scalar case



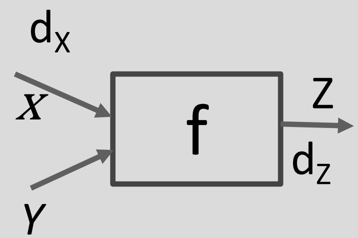
$d_{Output} = 1$



$d_X = wi * d_Y$ and $\frac{\partial\ Output}{\partial Wi} = d_Y * x$

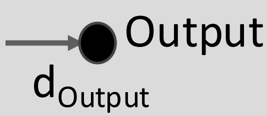


$d_X = d1 + d2$

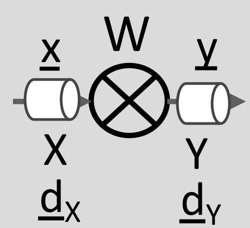


$d_X = \frac{\partial f}{\partial X}(x, y) * d_Z$

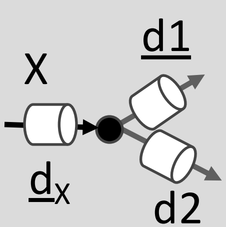
Vector case



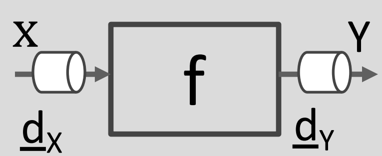
$d_{Output} = 1$



$\underline{d}_X = W^T * \underline{d}_Y$ $\frac{\partial\ Output}{\partial W} = \underline{d}_Y \otimes \underline{x}$

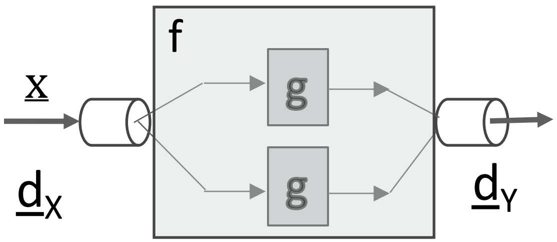


$\underline{d}_X = \underline{d1} + \underline{d2}$

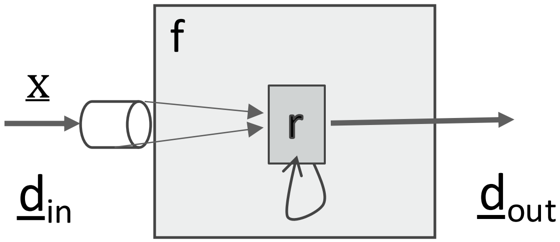


$\underline{d}_X = J_f * \underline{d}_Y$

Important special cases



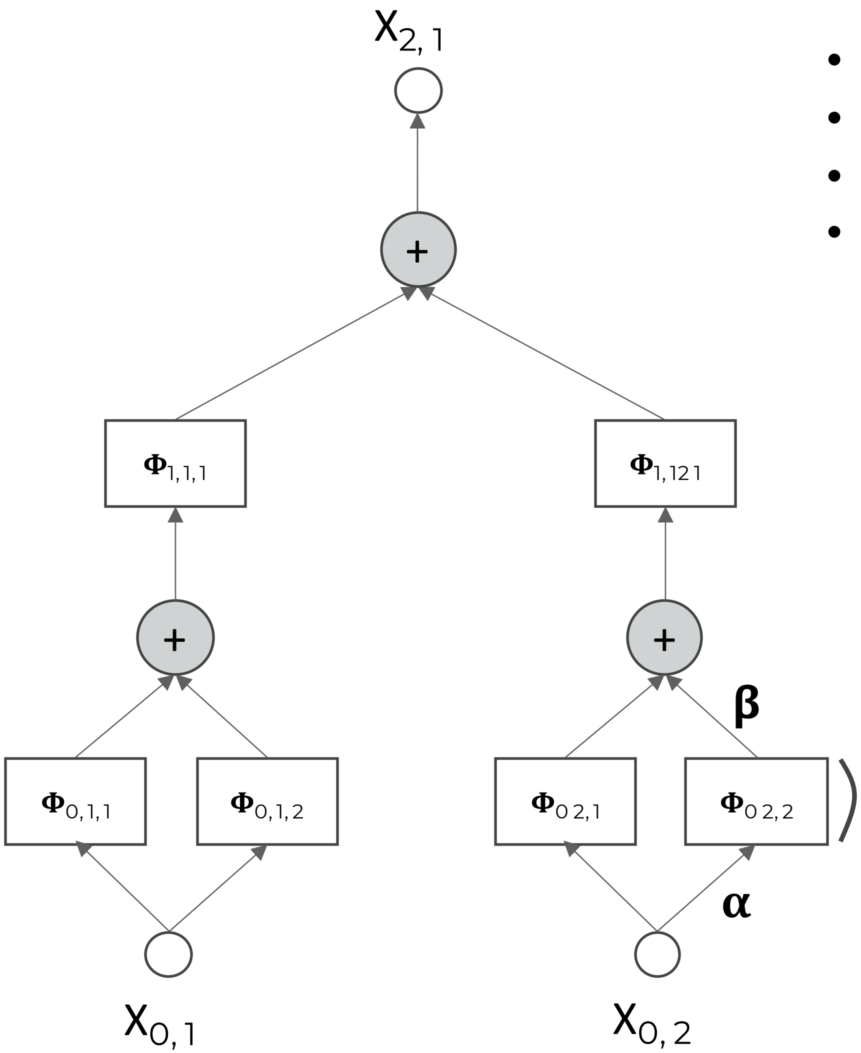
map



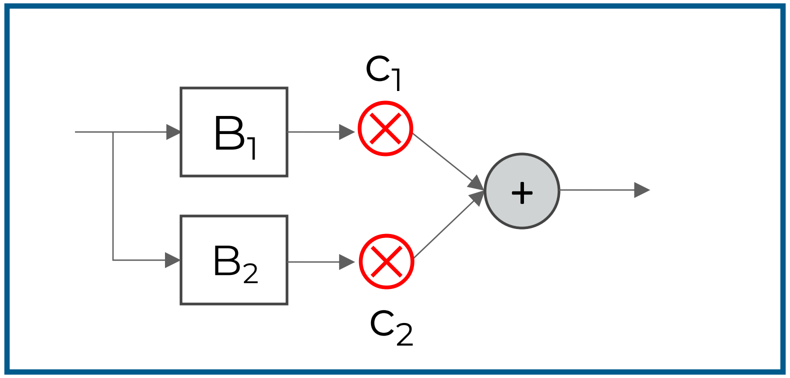
reduce

- f: map (g: $\Re \rightarrow \Re$)
  - J_f = diagonal matrix with $J_f(i,i) = g'(x(i))$ $\qquad \underline{d}_X = \begin{pmatrix} g'(\underline{x}(1)) & 0 \\ 0 & g'(\underline{x}(2)) \end{pmatrix} \underline{d}_Y$
- f: reduce (r: $\Re \times \Re \rightarrow \Re$)
  - 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)

Kolmogorov-Arnold Networks (KANs)



- Alternative to MLPs
- B_i: **B-spline**
- $\boldsymbol{\Phi} = \boldsymbol{\Sigma}_i\, c_i\, B_i$: **spline**
- Training: backpropagation to adjust B-spline control points c_i

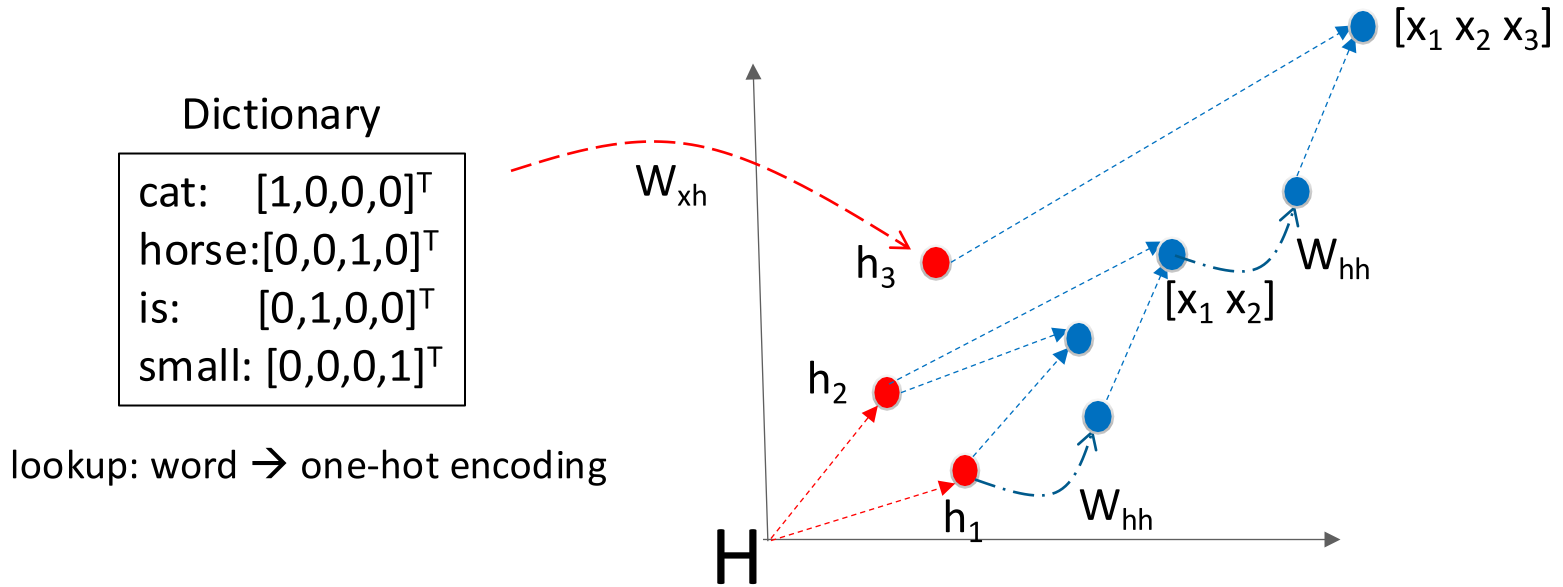


Recurrent Neural Networks (RNNs)

Running example

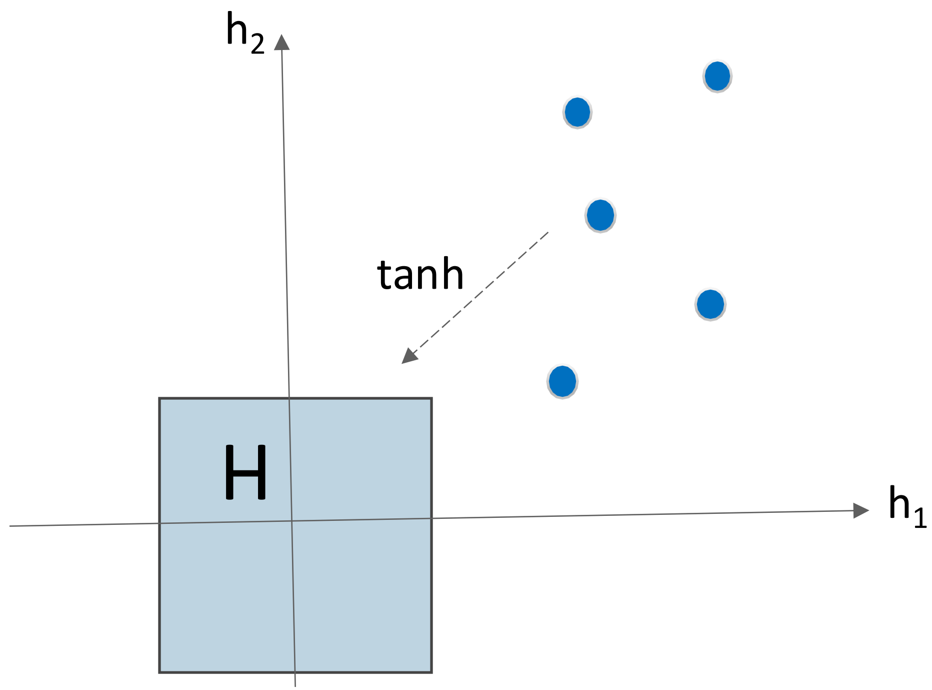
- RNNs: input is sequence
- Machine translation
  - Input: sentence (sequence of words) in English
  - Output: sentence (sequence of words) in French
- Training data
  - Set of sentence pairs: (sentence in English, sentence in French)
- Abstractly we want a function of this type
  - F: $[x_1,x_2,\ldots,x_m] \rightarrow [y_1,y_2,\ldots,y_n]$ (m,n can be different for different sentences)
  - Input sequence can be of arbitrary length
  - Assume m=n for simplicity
- Questions
  - How do we encode (represent) words?
  - How do we encode sequences of words?
  - How do we handle arbitrarily long sequences?
  - How is output produced?

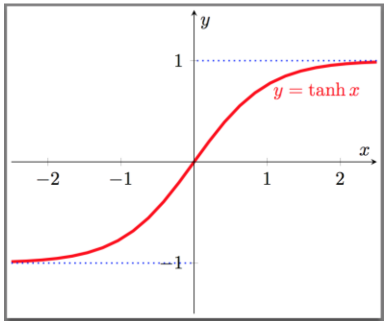
Encoding words and sequences of words



- **Vector space model**: words and sequences of words **embedded** as points in $H = \Re^m$
- Embedding of word x: $h = W_{xh} * \text{lookup}(x)$
  - lookup() uses dictionary to map word x to its one-hot encoding
  - W_{xh} is learned: column m is embedding of word with 1 in the m^{th} position of one-hot encoding
- Embedding of sequence (e.g.) $[x_1\ x_2]$
  - One possibility: add embeddings of x_1 and x_2
    - Drawback: [small cat] will have same embedding as [cat small]
  - Better idea: $W_{hh} * h_1 + h_2$ (where W_{hh} is learned)
- In general, H: sequence of words → $\Re^m$
  - $H([\ ]) = \underline{0}$
  - $H([x_1\ x_2 \ldots\ x_{i-1}\ x_i]) = W_{hh} * H([x_1\ x_2 \ldots\ x_{i-1}]) + W_{xh} * \text{lookup}(x_i)$

In practice



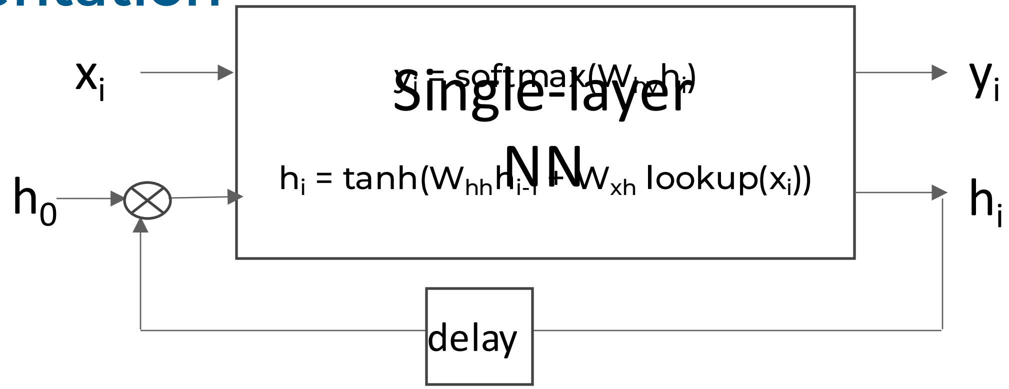


$$\frac{dy}{dx} = 1 - \tanh^2(x)$$

- Use tanh to squash H encodings into unit hypercube to prevent blow-up
  – $H([x_1\ x_2 \ldots.\ x_{i-1}\ x_i]) = \tanh(W_{hh} * H([x_1\ x_2\ \ldots\ x_{i-1}]) + W_{xh} * \mathrm{lookup}(x_i))$

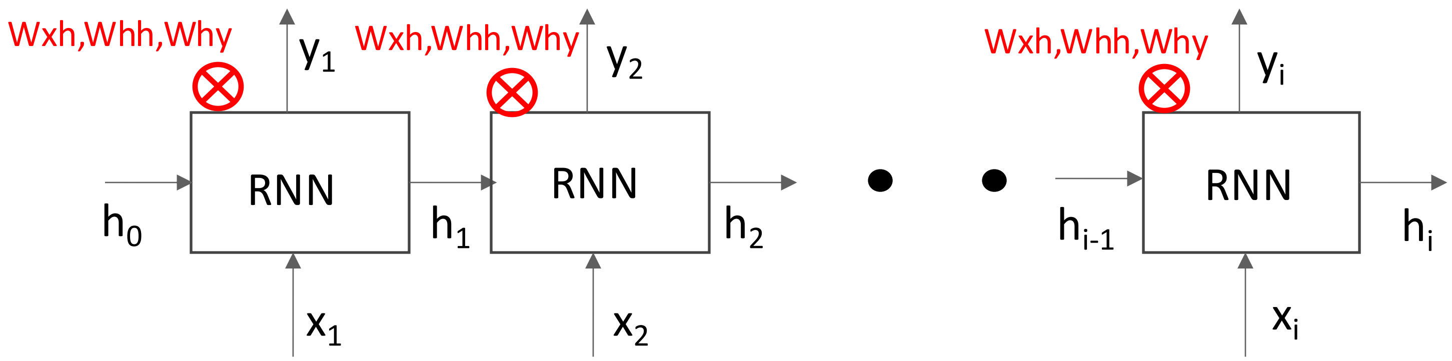
- Output for simple RNN produced "online"
  – y_i depends only on $[x_1,\ldots,x_i]$
  – $y_i \sim \mathrm{softmax}(W_{hy} * H([x_1\ x_2 \ldots.\ x_{i-1}\ x_i]))$
    > Output of softmax = probability vector for next output word
    > y_i is sampled from output distribution of softmax

RNN implementation



- RNN is single-layer neural network with feedback loop
- $H([x_1\ x_2 \ldots .\ x_{i-1}\ x_i])$ represented as vector h_i
  - Fed back to next iteration
- Details
  - W_{xh} can be initialized to embeddings from Word2Vec
  - RNNs can be chained to form “multi-layer” RNNs

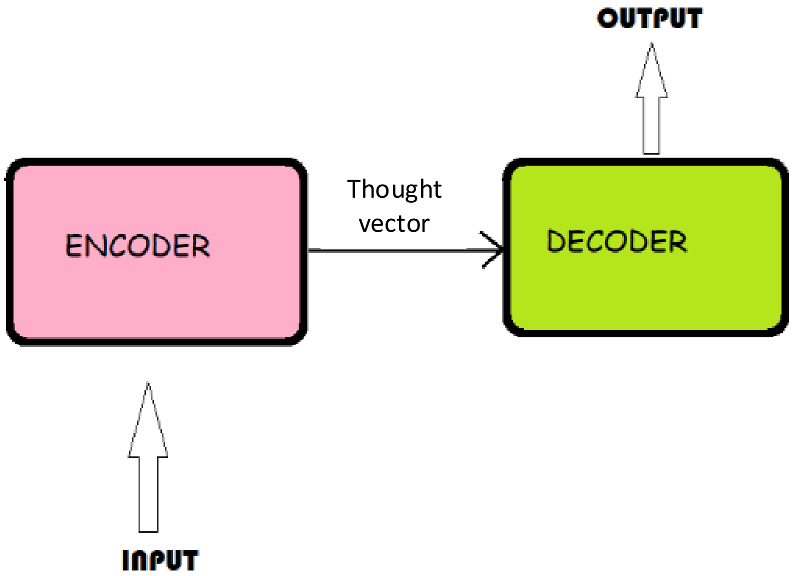
Training RNNs



- Training data: $\{[x_1,x_2,\ldots,x_n] \rightarrow [Y_1,Y_2,\ldots,Y_n]\}$
  - Notation: Y_i is training data, y_i is output produced by RNN during "inference"
- Training: at each step
  - Compute cross-entropy between ground truth Y_i and computed value y_i
    - Strictly speaking, between one-hot encoding of Y_i and output of softmax at step i
  - Back-propagate using weight-sharing to update weights
  - In practice, limit the size of the "look-back" window to 3-4
- Analogy: path-sensitive dataflow analysis

Improving RNNs: Encoder-decoder architectures

- Requiring output to be produced online means
  - No “look-ahead” in input stream is possible when determining how to produce next output word
  - Input and output sequences must have same length
- Solution
  - First encode entire input sequence (encoder)
  - Then produce output one word at a time (decoder)
- Two architectures
  - Baseline encoder-decoder architecture based on RNNs
  - Transformers



Baseline encode-decoder architecture

- We want to learn a function F: $[x_1,x_2,\ldots,x_n,y_1,y_2,\ldots,y_{i-1}] \rightarrow y_i$
- Training input: $[x_1,\ldots,x_n]$, $[Y_1,\ldots,Y_n]$
- Encoder
  - $h_i = f_1(h_{i-1},x_i)$
    - $h_0 = 0$
  - h_n = thought vector (embedding of $[x_1,x_2,\ldots,x_n]$)

$$f_1 = \tanh(W_{hh}*h_{i-1} + W_{xh}*\text{lookup}(x_i))$$

- Decoder (training)
  - $h_{n+i} = f_1(h_{n+i-1},Y_{i-1})$ (embedding of $[x_1,x_2,\ldots,x_n,Y_0,Y_1,\ldots,Y_{i-1}]$)
    - Y_0 = _START
  - $y_i = f_2(h_{n+i})$ (used to compute loss between y_i and Y_i)

$$f_2 \sim \text{softmax}(W_{hy}*h_{n+i})$$

- Decoder (inference)
  - $h_{n+i} = f_1(h_{n+i-1},y_{i-1})$ (embedding of $[x_1,x_2,\ldots,x_n,y_0,y_1,\ldots,y_{i-1}]$)
    - y_0 = _START
  - $y_i = f_2(h_{n+i})$

Remarks on RNNs

- Drawbacks of RNN-based translation

  (1) Encoding and decoding are sequential

  (2) Information loss for long sequences

  > In principle, encoder-decoder RNN architectures allow the decoder to see the entire input sequence before producing any output

  > However, signal from first few words is lost by end of long sequence

  > Experience: RNN-based translation works only for sentences of 4-5 words and if languages are well aligned

- Solution: transformer

  (1) Create encoding of sequence in **parallel**

  (2) **Attention**: pick up important signals for a given word from *anywhere* in input sequence

  > Example: The boy stood on the burning deck whence all but he had fled.

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
- Extension to **Kolmogorov-Arnold Networks** (KANs) and Recurrent Neural Networks (RNNs)