Back Propagation and Other Differentiation Algorithms

Sargur N. Srihari
srihari@cedar.buffalo.edu

Topics (Deep Feedforward Networks)

- Overview

1. Example: Learning XOR
2. Gradient-Based Learning
3. Hidden Units
4. Architecture Design
5. Backpropagation and Other Differentiation Algorithms
6. Historical Notes

Topics in Backpropagation

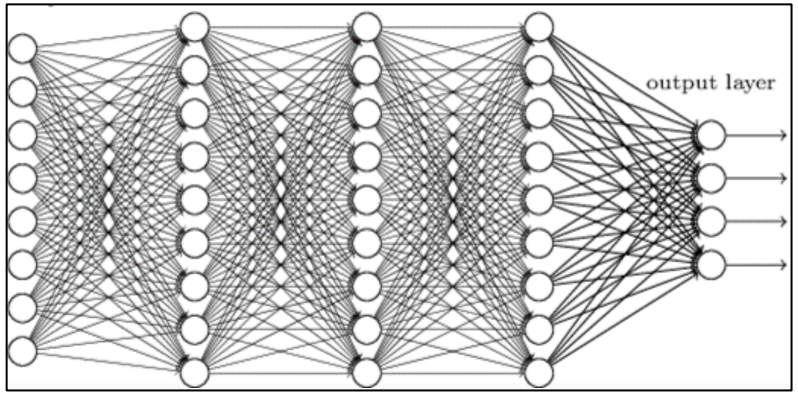
1. Overview
2. Computational Graphs
3. Chain Rule of Calculus
4. Recursively applying the chain rule to obtain backprop
5. Backpropagation computation in fully-connected MLP
6. Symbol-to-symbol derivatived
7. General backpropagation
8. Ex: backpropagation for MLP training
9. Complications
10. Differentiation outside the deep learning community
11. Higher-order derivatives

Overview of Backpropagation

Forward Propagation

- Producing an output from input
  - When we *use* a Feed-Forward Network to accept an input $\boldsymbol{x}$ and produce an output $\hat{\boldsymbol{y}}$ information $\boldsymbol{x}$ propagates to hidden units at each layer and finally produces $\hat{\boldsymbol{y}}$
  - This is called forward propagation
- During training (quality of result is evaluated):
  - forward propagation can continue onward
  - until it produces scalar cost $J(\boldsymbol{\theta})$ over N training samples $(\boldsymbol{x}_n, \boldsymbol{y}_n)$

Equations for Forward Propagation



Producing an output:

First layer given by $\boldsymbol{h}^{(1)}=g^{(1)}(\boldsymbol{W}^{(1)\mathrm{T}}\boldsymbol{x}+\boldsymbol{b}^{(1)})$
Second layer is $\boldsymbol{h}^{(2)}=g^{(2)}(\boldsymbol{W}^{(2)\mathrm{T}}\boldsymbol{h}^{(2)}+\boldsymbol{b}^{(2)})$,
.....
Final output is $\hat{\boldsymbol{y}}=g^{(d)}(\boldsymbol{W}^{(d)\mathrm{T}}\boldsymbol{h}^{(d)}+\boldsymbol{b}^{(d)})$

During training, cost over n exemplars:

$$J(\theta)=J_{MLE}+\lambda\left(\sum_{i,j}\left(W_{i,j}^{(1)}\right)^2+\sum_{i,j}\left(W_{i,j}^{(2)}\right)^2+....\right)$$

$$J_{MLE}=\frac{1}{N}\sum_{n=1}^{N}||\,\hat{\boldsymbol{y}}-\boldsymbol{y}_n\,||^2$$

Back-Propagation Algorithm

- Often simply called *backprop*
  - Allows information from the cost to flow back through network to compute gradient
- Computing analytical expression for the gradient is straightforward
  - But numerically evaluating the gradient is computationally expensive
- The backpropagation algorithm does this using a simple and inexpensive procedure

Analytical Expression for Gradient

- Sum-of-squares criterion over n samples

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\left\{ \boldsymbol{w}^T\boldsymbol{\phi}(\mathrm{x}_n) - t_n \right\}^2$$

  - Expression for gradient

$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) = \sum_{n=1}^{N}\left\{\boldsymbol{w}^T\boldsymbol{\phi}(x_n) - t_n\right\}\boldsymbol{\phi}(x_n)$$

- Another way of saying the same with cost $J(\theta)$:

$$J_n(\boldsymbol{\theta}) = || \boldsymbol{\theta}^T\boldsymbol{x}_n - \boldsymbol{y}_n ||^2$$

$$\nabla_{\boldsymbol{\theta}} J_n(\boldsymbol{\theta}) = \boldsymbol{\theta}^T\left(\boldsymbol{\theta}\ \boldsymbol{x}_n - \boldsymbol{y}_n\right) = \boldsymbol{\theta}^T\boldsymbol{\theta}\boldsymbol{x}_n - \boldsymbol{\theta}^T\boldsymbol{y}_n$$

Backpropagation is not Learning

- Backpropagation often misunderstood as the whole learning algorithm for multilayer networks
  - It only refers to method of computing gradient
- Another algorithm, e.g., SGD, is used to perform learning using this gradient
  - Learning is updating weights using gradient:

$$\boxed{\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla J_n(\theta)}$$

- Backpropagation is also misunderstood to being specific to multilayer neural networks
  - It can be used to compute derivatives for any function (or report that the derivative is undefined)

Importance of Backpropagation

- Backprop is a technique for computing derivatives quickly
  - It is the key algorithm that makes training deep models computationally tractable
  - For modern neural networks it can make training gradient descent 10 million times faster relative to naiive implementation
    - It is the difference between a model that takes a week to train instead of $200{,}000$ years

Computing gradient for arbitrary function

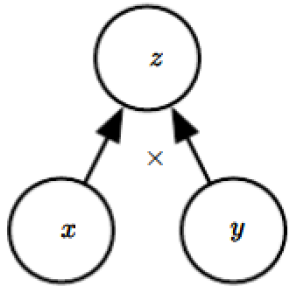
- Arbitrary function $f(\boldsymbol{x}, \boldsymbol{y})$
  - $\boldsymbol{x}$: variables for which derivatives $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ are desired
  - $\boldsymbol{y}$ is an additional set of variables that are inputs to the function but whose derivatives are not required
- Gradient required is of cost wrt parameters, $\nabla_{\theta} J(\theta)$
- Backprop is also useful for other ML tasks
  - Those that need derivatives, as part of learning process or to analyze a learned model
  - To compute Jacobian of a function f with multiple outputs
- We restrict to case where f has a single output

Computational Graphs

Computational Graphs

- To describe backpropagation use precise computational graph language
  - Each node is either
    - A variable
      - Scalar, vector, matrix, tensor, or other type
    - Or an Operation
      - Simple function of one or more variables
      - Functions more complex than operations are obtained by composing operations
  - If variable y is computed by applying operation to variable x then draw directed edge from x to y

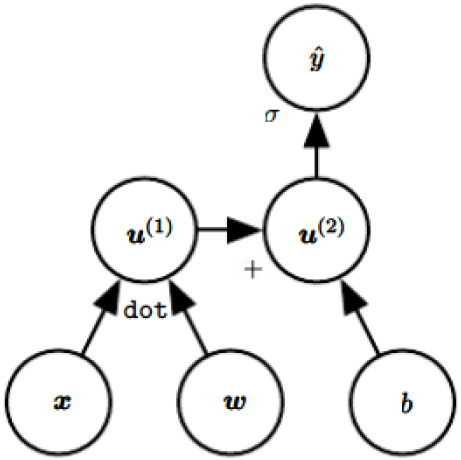
Ex: Computational Graph of xy



(a)

(a) Compute $z = xy$

Ex: Graph of Logistic Regression

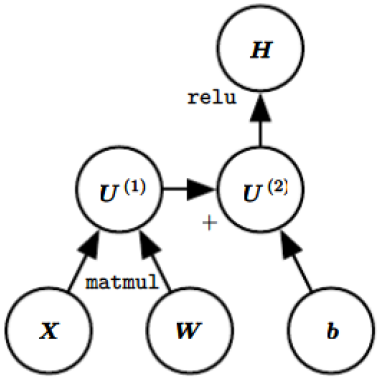


(b)

(b) Logistic Regression Prediction $\hat{y} = \sigma(\mathrm{x}^T \boldsymbol{w} + b)$

– Variables in graph $u^{(1)}$ and $u^{(2)}$ are not in original expression, but are needed in graph

Ex: Graph for ReLU



(c)

(c) Compute expression $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$

– Computes a design matrix of Rectified linear unit activations $\boldsymbol{H}$ given design matrix of minibatch of inputs $\boldsymbol{X}$

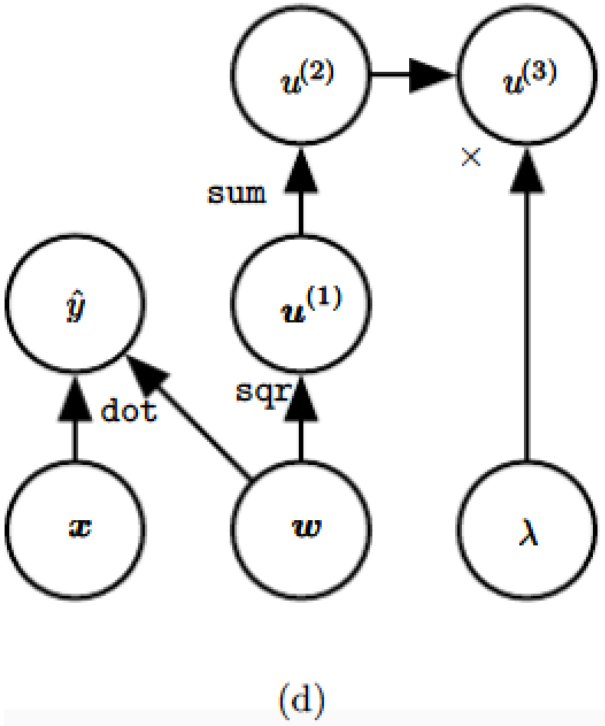
Ex: Two operations on input

(d) Perform more than one operation to a variable

Weights $\boldsymbol{w}$ are used in two operations:

- to make prediction $\hat{y}$ and
- the weight decay penalty

$$\lambda \sum_i w_i^2$$



(d)

Chain Rule of Calculus

Calculus' Chain Rule for Scalars

- Formula for computing derivatives of functions formed by composing other functions whose derivatives are known
  - Backpropagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient
    - Let x be a real number
    - Let f and g be functions mapping from a real number to a real number
    - If $y=g(x)$ and $z=f\left(g(x)\right)=f\left(y\right)$
- Then the chain rule states that $$\boxed{\frac{dz}{dx}=\frac{dz}{dy}\cdot\frac{dy}{dx}}$$

Generalizing Chain Rule to Vectors

- Suppose $\boldsymbol{x} \in R^m, \boldsymbol{y} \in R^n$

  g maps from R^m to R^n and

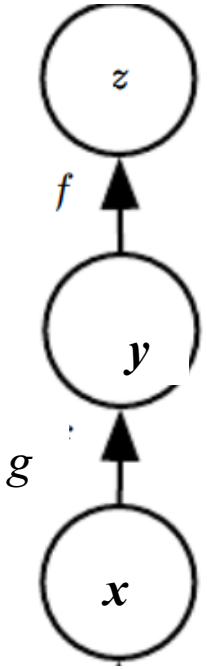
  f from R^n to R

- If $\boldsymbol{y}=g(\boldsymbol{x})$ and $z=f(\boldsymbol{y})$ then $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x}$

- In vector notation this is $\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$

  - where $\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)$ is the n x m Jacobian matrix of g

- Thus gradient of z wrt $\boldsymbol{x}$ is product of:
- Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ and gradient vector $\nabla_{\boldsymbol{y}} z$
- Backprop algorithm consists of performing *Jacobian-gradient* product for each step of graph

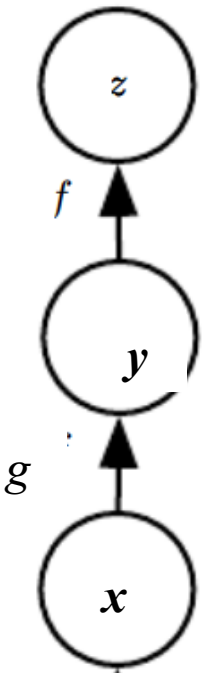


Generalizing Chain Rule to Tensors

- Backpropagation is usually applied to tensors with arbitrary dimensionality
- This is exactly the same as with vectors
  - Only difference is how numbers are arranged in a grid to form a tensor
    - We could flatten each tensor into a vector, compute a vector-valued gradient and reshape it back to a tensor
- In this view backpropagation is still multiplying Jacobians by gradients

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{T} \nabla_{\boldsymbol{y}} z$$

Chain Rule for tensors



- To denote gradient of value z wrt a tensor X we write $\boxed{\nabla_{\mathrm{X}} z}$ as if X were a vector
- For 3-D tensor, X has three coordinates
  - We can abstract this away by using a single variable i to represent complete tuple of indic
- For all possible tuples i $\boxed{\left(\nabla_X z\right)_i}$ gives $\frac{\partial z}{\partial X_i}$
  - Exactly same as how for all possible indices i into a vector, $\boxed{\left(\nabla_X z\right)_i}$ gives $\frac{\partial z}{\partial X_i}$
- Chain rule for tensors
  - If $\mathrm{Y}=g\left(\mathrm{X}\right)$ and $z=f\left(\mathrm{Y}\right)$ then $\boxed{\nabla_X(z)=\sum_j \left(\nabla_X Y_j\right)\frac{\partial z}{\partial Y_j}}$

Recursively applying the chain rule to obtain backprop

Backprop is Recursive Chain Rule

- Backprop is obtained by recursively applying the chain rule
- Using chain rule it is straightforward to write expression for gradient of a scalar wrt any node in graph for producing that scalar
- However, evaluating that expression on a computer has some extra considerations
  - E.g., many subexpressions may be repeated several times within overall expression
    - Whether to store subexpressions or recompute them

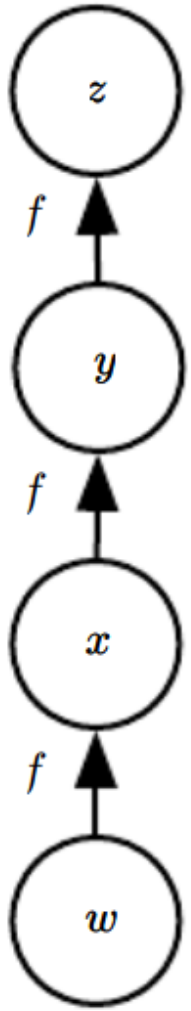
Example of repeated subexpressions

- Let w be the input to the graph
  - We use same function f: $R \rightarrow R$ at every step:

$x=f(w),\ y=f(x),\ z=f(y)$

- To compute $\frac{\partial z}{\partial w}$ apply

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$
$$= f'(y)f'(x)f'(w) \quad (1)$$
$$= f'(f(f(w)))f'(f(w))f'(w) \quad (2)$$



Eq. (1): compute $f(w)$ once and store it in x

- This the approach taken by backprop

Eq. (2): expression $f(w)$ appears more than once

$f(w)$ is recomputed each time it is needed

For low memory, (1) preferable: reduced runtime

(2) is also valid chain rule, useful for limited memory

For complicated graphs,

exponentially wasted computations making naiive implementation of chain rule infeasible

Simplified Backprop Algorithm

- Version that directly computes actual gradient
  - In the order it will actually be done according to recursive application of chain rule
    - Algorithm *Simplified Backprop* along with associated
    - *Forward Propagation*
- Could either directly perform these operations
  - or view algorithm as symbolic specification of computational graph for computing the back-prop
- This formulation does not make specific
  - Manipulation and construction of symbolic graph that performs gradient computation

Computing a single scalar

- Consider computational graph of how to compute scalar $u^{(n)}$
  - say loss on a training example
- We want gradient of this scalar $u^{(n)}$ wrt n_i input nodes $u^{(1)},..u^{(ni)}$
  - i.e., we wish to compute $\boxed{\frac{\partial u^{(n)}}{\partial u_i}}$ for all $i = 1,..,n_i$
- In application of backprop to computing gradients for gradient descent over parameters
  - $u^{(n)}$ will be cost associated with an example or a minibatch, while
  - $u^{(1)},..u^{(ni)}$ correspond to model parameters

Nodes of Computational Graph

- Assume that nodes of the graph have been ordered such that
  - We can compute their output one after another
  - Starting at $u^{(n_i+1)}$ and going to $u^{(n)}$
- As defined in Algorithm shown next
  - Each node $u^{(i)}$ is associated with operation $f^{(i)}$ and is computed by evaluating the function

$$u^{(i)} = f\left(A^{(i)}\right)$$

where $A^{(i)} = \mathrm{Pa}(u^{(i)})$ is set of nodes that are parents of $u^{(i)}$

- Algorithm specifies a computational graph $\mathcal{G}$
  - Computation in reverse order gives back-propagation computational graph $\mathcal{B}$

Forward Propagation Algorithm

Algorithm 1: Performs computations mapping n_i inputs $u^{(1)},..u^{(ni)}$ to an output $u^{(n)}$. This defines computational graph $\mathcal{G}$ where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to a set of arguments $A^{(i)}$ that comprises values of previous nodes $u^{(j)}$, $j < i$ with j ε $\mathrm{Pa}(u^{(i)})$. Input to $\mathcal{G}$ is $\boldsymbol{x}$ set to the first n_i nodes $u^{(1)},..u^{(ni)}$. Output of $\mathcal{G}$ is read off the last (output) node for $u^{(n)}$.

- **for** $i = 1,..,n_i$ **do**
  $u^{(i)} \leftarrow x_i$
- **end for**
- **for** $i = n_i+1,..,\ n$ **do**
  $A^{(i)} \leftarrow \{u^{(j)} |\ j\ \varepsilon\ \mathrm{Pa}(u^{(i)})\ \}$
  $u^{(i)} \leftarrow f^{(i)}(A^{(i)})$
- **end for**
- **return** $u^{(n)}$

Computation in $\mathcal{B}$

- Proceeds exactly in reverse order of computation in $\mathcal{G}$
- Each node in $\mathcal{B}$ computes the derivative $\boxed{\frac{\partial u^{(n)}}{\partial u_i}}$ associated with the forward graph node $u^{(i)}$
- This is done using the chain rule wrt the scalar output $u^{(n)}$

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j\hat{I}Pa\left(u^{(i)}\right)} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

Preamble to Simplified Backprop

- Objective is to compute derivatives of $u^{(n)}$ with respect to variables in the graph
  - Here all variables are scalars and we wish to compute derivatives wrt
  - We wish to compute the derivatives wrt $u^{(1)},..u^{(ni)}$
- Algorithm computes the derivatives of all nodes in the graph

Simplified Backprop Algorithm

- **Algorithm 2:** For computing derivatives of $u^{(n)}$ wrt variables in G. All variables are scalars and we wish to compute derivatives wrt $u^{(1)},..u^{(ni)}$. We compute derivatives of all nodes in G.

- Run forward propagation to obtain network activations
- Initialize **grad-table**, a data structure that will store derivatives that have been computed, The entry **grad-table** $[u^{(i)}]$ will store the computed value of $\boxed{\frac{\partial u^{(n)}}{\partial u_i}}$

1. **grad-table** $[u^{(n)}] \leftarrow 1$
2. **for** $j=n-1$ down to 1 do
   grad-table $[u^{(j)}] \leftarrow \boxed{\sum_{i:j\in Pa\left(u^{(i)}\right)} \textbf{grad-table}\left[u^{(i)}\right]\frac{\partial u^{(i)}}{\partial u^{(j)}}}$
3. **endfor**
4. **return** $\{[u^{(i)}] \mid i=1,.., n_i\}$

Step 2 computes

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j\in Pa\left(u^{(i)}\right)} \frac{\partial u^{(n)}}{\partial u^{(i)}}\frac{\partial u^{(i)}}{\partial u^{(j)}}$$

Computational Complexity

- Computational cost is proportional to no. of edges in graph (same as for forward prop)
  - Each $\boxed{\frac{\partial u^{(i)}}{\partial u^{(j)}}}$ is a function of the parents of $u^{(j)}$ and $u^{(i)}$ thus linking nodes of the forward graph to those added for $\mathcal{B}$
- Backpropagation thus avoids exponential explosion in repeated subexpressions
  - By simplifications on the computational graph

Generalization to Tensors

- Backprop is designed to reduce the no. of common sub-expressions without regard to memory
- It performs on the order of one Jacobian product per node in the graph

Backprop in fully connected MLP

Backprop in fully connected MLP

- Consider specific graph associated with fully-connected multilayer perceptron
- Algorithm discussed next shows forward propagation
  - Maps parameters to supervised loss $L(\hat{y}, y)$ associated with a single training example $(\boldsymbol{x}, y)$ with $\hat{y}$ the output when x is the input

Forward Prop: deep nn & cost computation

- **Algorithm 3:** The loss $L(\hat{\boldsymbol{y}},\boldsymbol{y})$ depends on output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$. To obtain total cost J the loss may be added to a regularizer $\Omega(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ contains all the parameters (weights and biases). *Algorithm 4* computes gradients of J wrt parameters W and $\boldsymbol{b}$. This demonstration uses only single input example $\boldsymbol{x}$.

- **Require**: Net depth l; Weight matrices $W^{(i)}$, $i \in \{1,..,l\}$; bias parameters $\boldsymbol{b}^{(i)}$, $i \in \{1,..,l\}$; input $\boldsymbol{x}$; target output $\boldsymbol{y}$

  1. $\boldsymbol{h}^{(0)} = \boldsymbol{x}$

  2. for $k = 1$ to l **do**

  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + W^{(k)}\boldsymbol{h}^{(k-1)}$

  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$

  3. end for

  4. $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$

  $J = L(\hat{\boldsymbol{y}},\boldsymbol{y}) + \lambda\,\Omega(\boldsymbol{\theta})$

Backward compute: deep NN of Algorithm 3

Algorithm 4: uses in addition to input $\boldsymbol{x}$ a target $\boldsymbol{y}$. Yields gradients on activations $\boldsymbol{a}^{(k)}$ for each layer starting from output layer to first hidden layer. From these gradients one can obtain gradient on parameters of each layer, Gradients can be used as part of SGD.

After forward computation compute gradient on output layer

$\boldsymbol{g} \leftarrow \boxed{\nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)}$

for $k = l$, l -1,..1 **do**

Convert gradient on layer's output into a gradient into the pre-nonlinearity activation (elementwise multiply if f is elementwise)

$\boldsymbol{g} \leftarrow \boxed{\nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'\left(\boldsymbol{a}^{(k)}\right)}$

Compute gradients on weights biases (incl. regularizn term)

$$\boxed{\nabla_{b^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta), \qquad \nabla_{W^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)}$$

Propagate the gradients wrt the next lower-level hidden layer's activations: $\boldsymbol{g} \leftarrow \boxed{\nabla_{h^{(k-1)}} J = W^{(k)T} \boldsymbol{g}}$

end for

Symbol-to-Symbol Derivatives

- Both algebraic expressions and computational graphs operate on symbols, or variables that do not have specific values
- They are called symbolic representations
- When we actually use or train a neural network, we must assign specific values for these symbols
- We replace a symbolic input to the network with a specific numeric value
  - E.g., $[2.5, 3.75, -1.8]^T$

Two approaches to backpropagation

1. Symbol-to-number differentiation
   - Take a computational graph and a set of numerical values for inputs to the graph
   - Return a set of numerical values describing gradient at those input values
   - Used by libraries: Torch and Caffe
2. Symbol-to-symbol differentiation
   - Take a computational graph
   - Add additional nodes to the graph that provide a symbolic description of desired derivatives
   - Used by libraries: Theano and Tensorflow

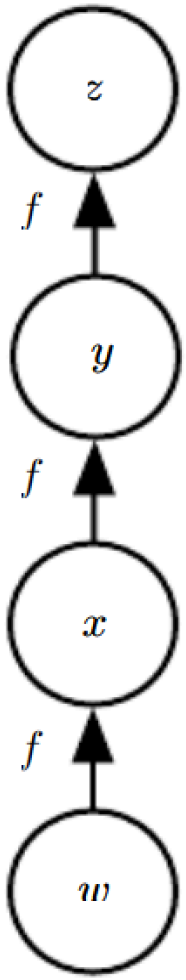
Symbol-to-symbol Derivatives

- To compute derivative using this approach, backpropagation does not need to ever access any actual numerical values
  - Instead it adds nodes to a computational graph describing how to compute the derivatives for any specific numerical values
  - A generic graph evaluation engine can later compute derivatives for any specific numerical values

Ex: Symbol-to-symbol Derivatives

- Begin with graph representing

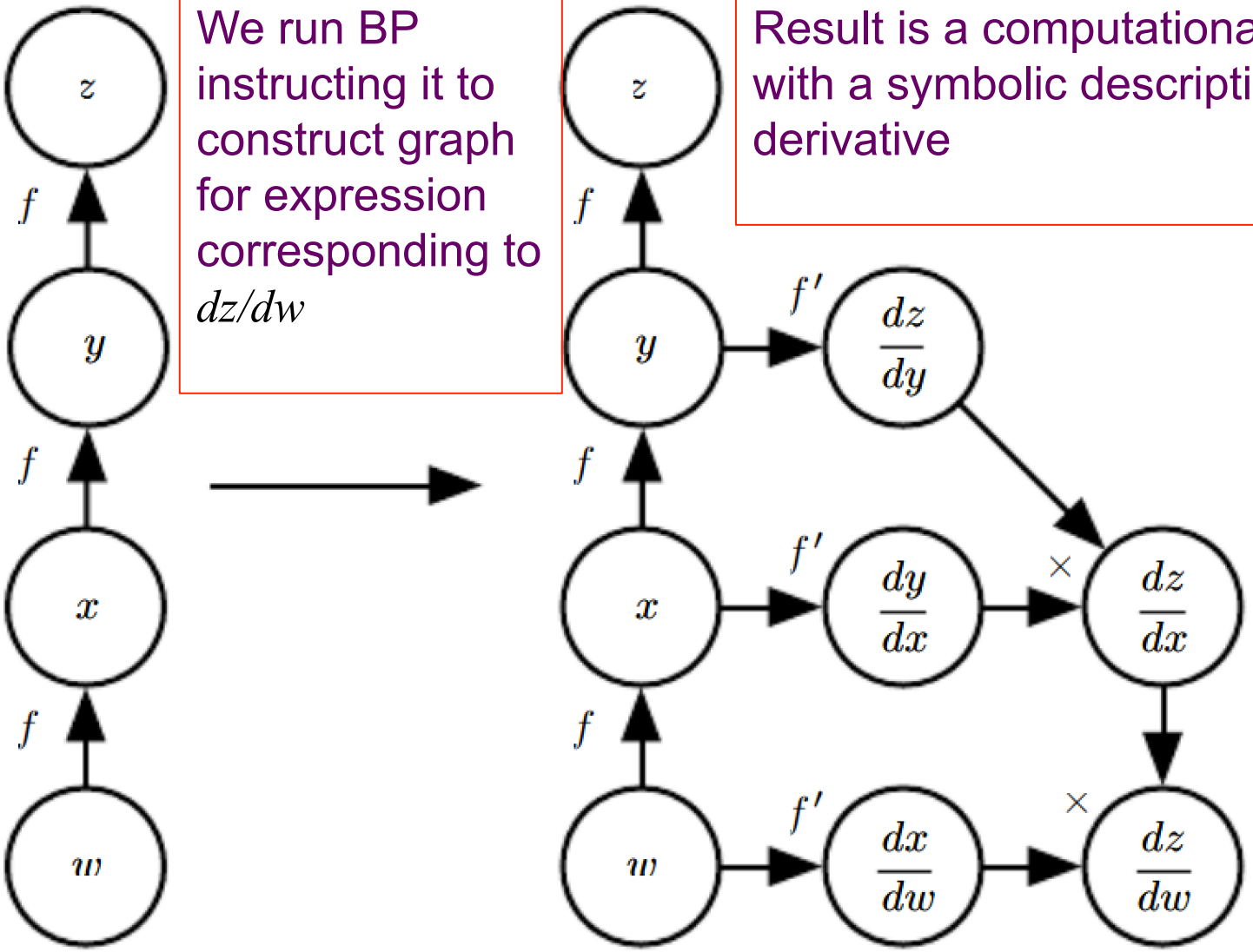
$$z = f(f(f(w)))$$



Symbol-to-Symbol Derivative Computation

We run BP instructing it to construct graph for expression corresponding to dz/dw

Result is a computational graph with a symbolic description of the derivative

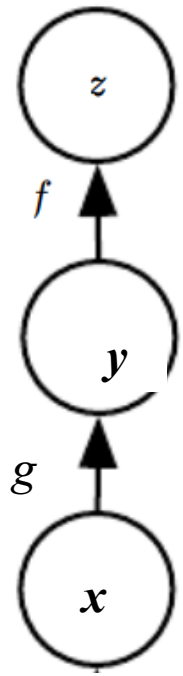


Advantages of Approach

- Derivatives are described in the same language as the original expression
- Because the derivatives are just another computational graph, it is possible to run back-propagation again
  - Differentiating the derivatives
  - Yields higher-order derivatives

General Backpropagation

- To compute gradient of scalar z wrt one of its ancestors $\boldsymbol{x}$ in the graph
  - Begin by observing that gradient wrt z is $\frac{dz}{dz} = 1$
  - Then compute gradient wrt each parent of z by multiplying current gradient by Jacobian of: operation that produced z
  - We continue multiplying by Jacobians traveling backwards until we reach $\boldsymbol{x}$
  - For any node that can be reached by going backwards from z through two or more paths sum the gradients from different paths at that node



$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{T} \nabla_{\boldsymbol{y}} z$$

Formal Notation for backprop

- Each node in the graph $\mathcal{G}$ corresponds to a variable
- Each variable is described by a tensor **V**
  - Tensors have any no. of dimensions
  - They subsume scalars, vectors and matrices

Each variable V is associated with the following subroutines:

- $\mathrm{get\_operation}\ (\mathrm{V})$
  - Returns the operation that computes V represented by the edges coming into V in G
  - Suppose we have a variable that is computed by matrix multiplication $C=AB$
    - Then $\mathrm{get\_operation}\ (\mathrm{V})$ returns a pointer to an instance of the corresponding C++ class

Other Subroutines of V

- $\mathrm{get\_consumers}\ (\mathrm{V}, \mathrm{G})$
  - Returns list of variables that are children of V in the computational graph $\mathcal{G}$
- $\mathrm{get\_inputs}\ (\mathrm{V}, \mathrm{G})$
  - Returns list of variables that are parents of V in the computational graph $\mathcal{G}$

bprop operation

- Each operation op is associated with a bprop operation
- bprop operation can compute a Jacobian vector product as described by

$$\nabla_x z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_y z$$

- This is how the backpropagation algorithm can achieve great generality
  - Each operation is responsible for knowing how to backpropagate through the edges in the graph that it participates in

Example of bprop

- Suppose we have
  - a variable computed by matrix multiplication $C=AB$
  - the gradient of a scalar z wrt C is given by G
- The matrix multiplication operation is responsible for two back propagation rules
  - One for each of its input arguments
    - If we call bprop to request the gradient wrt A given that the gradient on the output is G
      - Then bprop method of matrix multiplication must state that gradient wrt A is given by GB^T
    - If we call bprop to request the gradient wrt B
      - Then matrix operation is responsible for implementing the bprop and specifying that the desired gradient is A^TG

Inputs, outputs of bprop

- Backprogation algorithm itself does not need to know any differentiation rules
  - It only needs to call each operation's bprop rules with the right arguments
- Formally op.bprop (inputs X, G) must return

$$\sum_i \left(\nabla_{\boldsymbol{x}} \text{op.f}(\text{inputs})_i\right) G_i$$

  - which is just an implementation of the chain rule

$$\nabla_x z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_y z$$

  - inputs is a list of inputs that are supplied to the operation, op.f is a math function that the operation implements,
  - X is the input whose gradient we wish to compute,
  - G is the gradient on the output of the operation

Computing derivative of x^2

- Example: The mul operator is passed to two copies of x to compute x^2
- The $\mathrm{ob.prop}$ still returns x as derivative wrt to both inputs
- Backpropagation will add both arguments together to obtain $2x$

Software Implementations

- Usually provide both:
  1. Operations
  2. Their bprop methods
- Users of software libraries are able to backpropagate through graphs built using common operations like
  - Matrix multiplication, exponents, logarithms, etc
- To add a new operation to existing library must derive ob.prop method manually

Formal Backpropagation Algorithm

- **Algorithm 5:** *Outermost skeleton of backprop*
- This portion does simple setup and cleanup work, Most of the important work happens in the **build_grad** subroutine of Algorithm 6

- **Require**: **T**, Target set of variables whose gradients must be computed
- **Require**: G, the computational graph

1. Let G' be G pruned to contain only nodes that are ancestors of z and descendants of nodes in **T**
2. **for** V in **T** ***do***

   build-grad (V,G , G', **grad-table**)

   endfor
4. Return grad-table restricted to **T**

Inner Loop: build-grad

- **Algorithm 6:** Innerloop subroutine **build-grad**(V,G,G',**grad-table**) of the back-propagation algorithm, called by Algorithm 5

- **Require**: V, Target set of variables whose gradients to be computed; G, the graph to modify; G', the restriction of G to modify; **grad-table**,a data structure mapping nodes to their gradients

  if V is in **grad-table, then** return **grad-table** [V] **endif** $i \leftarrow 1$

  for C in **get-customers(**V,G'**) do**

  op$\leftarrow$get-operation(C)

  D$\leftarrow$**build-grad**(C,G,G',**grad-table**)

  G(i)$\leftarrow$ob.bprop(get-inputs(C,G'),V,D)

  $i \leftarrow i+1$

  endfor

  $G \leftarrow \Sigma_i G^{(i)}$

  grad-table[V] = G

  Insert G and the operations creating it into **G**

  Return G

Ex: backprop for MLP training

- As an example, walk through back-propagation ealgorithm as it is used to train a multilayer perceptron
- We use Minibatch stochastic gradient descent
- Backpropagation algorithm is used to compute the gradient of the cost on a single minibatch
- We use a minibatch of examples from the training set formatted as a design matrix X, and a vector of associated class labels $\boldsymbol{y}$

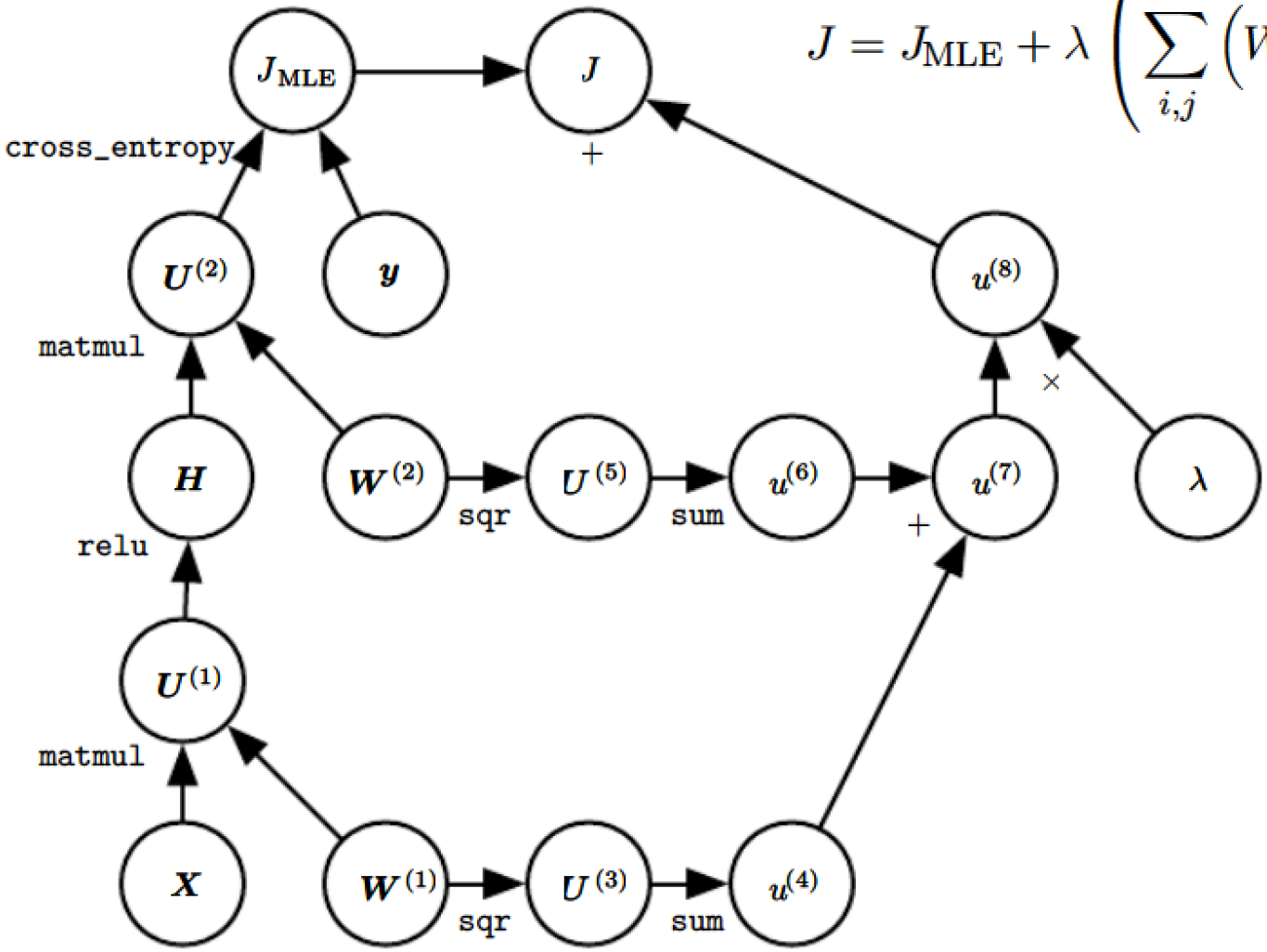
Ex: details of MLP training

- Network computes a layer hidden features $\boldsymbol{H}=\max\{0,\boldsymbol{X}\boldsymbol{W}^{(1)}\}$
  - No biases in model
- Graph language has relu to compute $\max\{0,\boldsymbol{Z}\}$
- Prediction: log-probs(unnorm) over classes: $\boldsymbol{H}\boldsymbol{W}^{(2)}$
- Graph language includes $\mathrm{cross\text{-}entropy}$ operation
  - computes cross-entropy between targets $\boldsymbol{y}$ and probability distribution defined by log probs
  - Resulting cross-entropy defines cost JMLE
  - We include a regularization term

Forward propagation graph

Total cost:

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W^{(1)}_{i,j} \right)^2 + \sum_{i,j} \left(W^{(2)}_{i,j} \right)^2 \right)$$

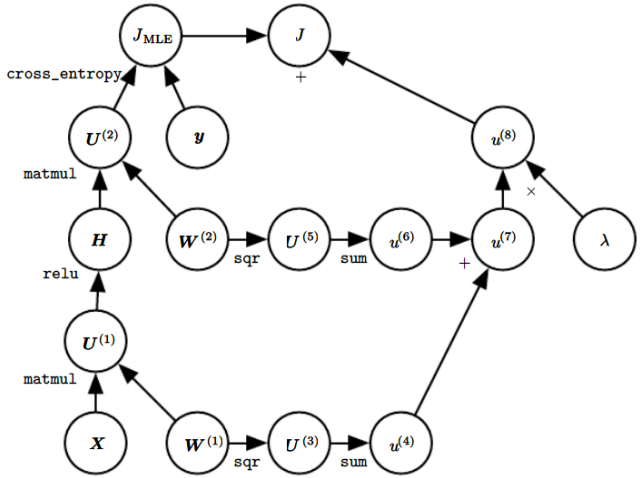


Computational Graph of Gradient

- It would be large and tedious for this example
- One benefit of back-propagation algorithm is that it can automatically generate gradients that would be straightforward but tedious manually for a software engineer to derive

Tracing behavior of Backprop

- Looking at forward prop graph
- To train we wish to compute both $\nabla_{W^{(1)}} J$ and $\nabla_{W^{(2)}} J$
- There are two different paths leading backward from J to the weights:
  - one through weight decay cost
    - It will always contribute $2\lambda W^{(i)}$ to the gradient on $W^{(i)}$
  - other through cross-entropy cost
    - It is more complicated



Cross-entropy cost

- Let G be gradient on unnormalized log probabilities $U^{(2)}$ given by cross-entropy op.
- Backprop needs to explore two branches:
  - On shorter branch adds $H^T G$ to the gradient on $W^{(2)}$
    - Using the backpropagation rule for the second argument to the matrix multiplication operation
  - Other branch: longer descending along network
    - First backprop computes $\nabla_H J = G W^{(2)T}$
    - Next relu operation uses backpropagation rule to zero out components of gradient corresponding to entries of $U^{(1)}$ that were less than 0. Let result be called G'
    - Use backpropagation rule for the second argument of matmul to add $X^T G'$ to the gradient on $W^{(1)}$

After Gradient Computation

- It is the responsibility of SGD or other optimization algorithm to use gradients to update parameters