General Back Propagation

Sargur N. Srihari
srihari@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

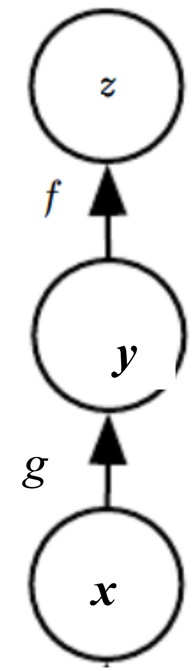
Topics in Backpropagation

- Forward and Backward Propagation

1. Computational Graphs
2. Chain Rule of Calculus
3. Recursively applying the chain rule to obtain backprop
4. Backpropagation computation in fully-connected MLP
5. Symbol-to-symbol derivatives
6. General backpropagation
7. Ex: backpropagation for MLP training
8. Complications
9. Differentiation outside the deep learning community
10. 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 $\mathbf{V}$
  - Tensors have any no. of dimensions
  - They subsume scalars, vectors and matrices

Each variable V is associated with the following subroutines:

- get_operation (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 get_operation (V) returns a pointer to an instance of the corresponding C++ class

Other Subroutines of V

- get_consumers (V, G)
  - Returns list of variables that are children of V in the computational graph $\mathcal{G}$
- get_inputs (V, 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

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

- 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

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

  - which is just an implementation of the chain rule

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

  - 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 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

7. Ex: general backprop for MLP

Ex: backprop for MLP training

- As an example, walk through back-propagation algorithm 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 $\boldsymbol{y}$

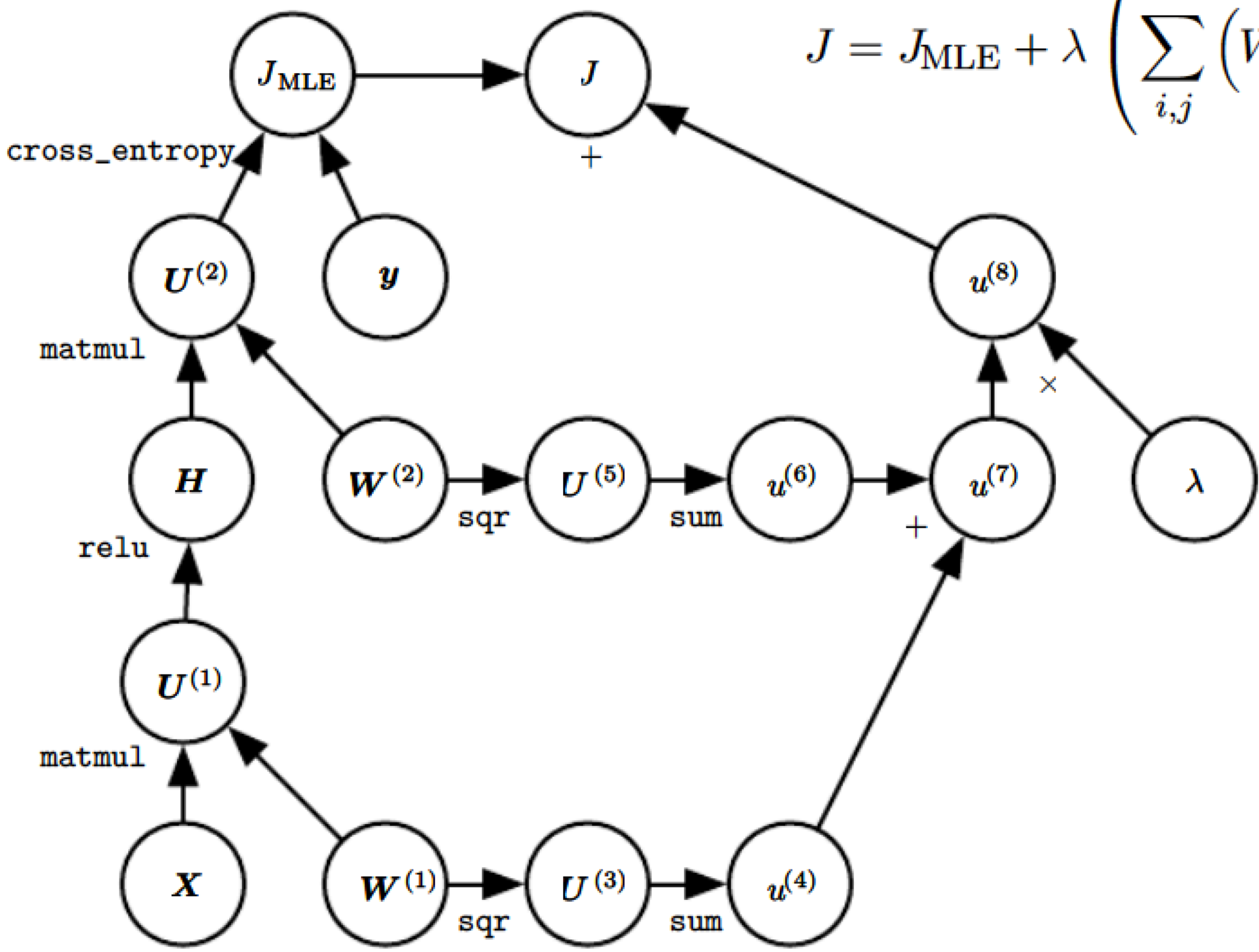
 # Ex: details of MLP training

- 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 cross-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:

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

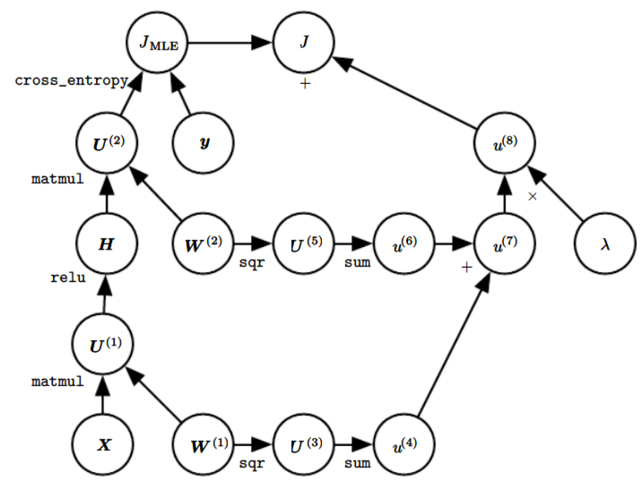


Computational Graph of Gradient

- 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

8. Complications

Complications

1. Returning more than a single tensor
2. Memory consumption
   – Backprop requires summing many tensors together
     • Instead of computing tensors separately add to a buffer
3. Need to handle various data types
   – 32-bit floating point, 64-bit floating point and integer
4. Determine whether gradient is undefined

• Various technicalities make real-world differentiation more complicated
  – But not unsurmountable