Feedforward Neural Networks and Backpropagation

Changyou Chen

Department of Computer Science and Engineering
Universitpy at Buffalo, SUNY
changyou@buffalo.edu

February 21, 2019

Backpropagation

Importance of Backpropagation

1. Backprop is a technique for computing derivatives quickly by avoiding duplicated computations.
2. It is the key algorithm that makes training deep models computationally tractable.
3. For modern neural networks it can make training gradient descent 10 million times faster relative to naive implementation:
   - it is the difference between a model that takes a week to train instead of 200,000 years.
4. It is based on the computational graph.
5. Simply called backprop.

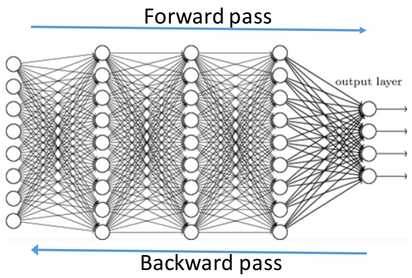
Anecdote

Geoffery Hinton and BP

- **1986**: Invented Backpropagation.
- **2017**: “Discard it, do it from the beginning!”
  - Capsule networks.

Basic Steps in Backpropagation

1. Two passes through the computational graph:
   - forward pass: compute the outputs of each layer.
   - backward pass: compute the gradients of parameters in each layer, based on results from the forward pass.
2. Backprop algorithm does this using a simple and inexpensive procedure.



Backprop vs. Learning

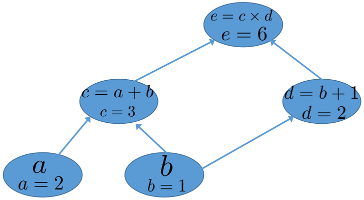
1. Backprop does not mean the whole learning algorithm.
2. Backprop only refers to the method of computing the gradient.
    - Another algorithm, such as SGD, is used to perform leaning using this gradient:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \nabla_{\mathbf{w}} J(\mathbf{w})$$

3. Often misunderstood to being specific to multilayer neural networks.
    - It can be used to compute derivatives for any function.
    - It also can be used to compute Jacobian of a function with multiple outputs.
    - We restrict to case where *f* has a single output.

Recap: Computational Graph

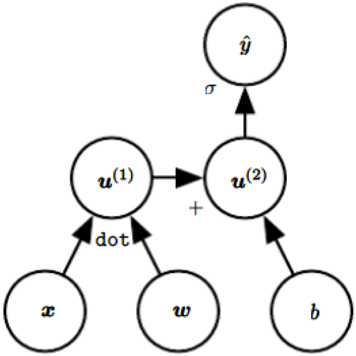
- Each node is either
  - a variable: scalar, vector, matrix, tensor, or other type
  - or an operation
    - ★ simple function of one or more variables
    - ★ functions more complex than operations are obtained by composing operations
  - if variable y is computed by applying operation to variable x then draw directed edge from x to y.



Example: Graph of Logistic Regression

$$\hat{y} = \sigma\left(\mathbf{w}^T \mathbf{x} + b\right)$$

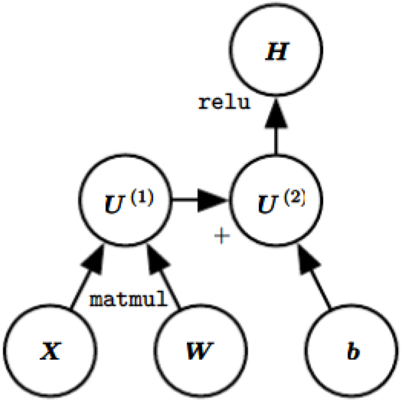
- Variables in graph $\mathbf{u}^{(1)}$ and $\mathbf{u}^{(2)}$ are not in the original expression, but are needed in the graph.



Example: Graph of ReLU

$$\mathbf{H} = \max\left\{0, \mathbf{W}^T\mathbf{X} + \mathbf{b}\right\}$$

- Variables in graph $\mathbf{U}^{(1)}$ and $\mathbf{U}^{(2)}$ are not in the original expression, but are needed in the graph.



Recap: Calculus' Chain Rule for Scalars

- Let x be a real number; f and g be functions mapping from a real number to a real number.
- Let $y = g(x)$, $z = f(g(x)) = f(y)$, then

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$$

Generalizing Chain Rule to Vectors

- Let $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$; $f : \mathbb{R}^m \to \mathbb{R}^n$, and $g : \mathbb{R}^n \to \mathbb{R}$.
- Let $\mathbf{y} = g(\mathbf{x})$, $z = f(g(\mathbf{x})) = f(\mathbf{y})$, then

$$\frac{\mathrm{d}z}{\mathrm{d}x_i} = \sum_j \frac{\mathrm{d}z}{\mathrm{d}y_j} \cdot \frac{\mathrm{d}y_j}{\mathrm{d}x_i}$$

  - Like multiple paths when each node represents one element of a vector.
- In vector notation, this is

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

  where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of g.
- Backprop algorithm consists of performing “Jacobian-gradient” product for each step of graph.

Generalizing Chain Rule to Tensors

1. Backpropagation is usually applied to tensors with arbitrary dimensionality.
2. This is almost the same as with vectors:
    - we could flatten each tensor into a vector, compute a vector-valued gradient and reshape it back to a tensor.
3. In this view backpropagation is still multiplying Jacobians by gradients:

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

Chain Rule for Tensors

1. To denote gradient of value z w.r.t. a tensor $\mathbf{X}$, we write as if $\mathbf{X}$ were a vector:
   - an element in a k-order tensor is indexed by a tuple $(i_1, \cdots, i_k)$, we abstract this away by a single (vector) variable $i \triangleq (i_1, \cdots, i_k)$ to represent the tuple.
2. For all possible tuples i, $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial X_i}$:
   - exactly same as how for all possible indices i in a vector, $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial X_i}$.
3. Chain rule for tensors: let $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$:

$$\nabla_{\mathbf{X}} z = \sum_i \underbrace{(\nabla_{\mathbf{X}} Y_i)}_{\text{tensor}} \underbrace{\frac{\partial z}{\partial Y_i}}_{\text{scalar}}$$

Backprop is Recursive Chain Rule

1. Backprop is obtained by recursively applying the chain rule.
2. Using the chain rule, it is straightforward to write expression for gradient of a scalar w.r.t. any node in graph producing that scalar.
3. However, evaluating that expression on a computer has some extra considerations:
    - *e.g.*, many subexpressions may be repeated several times within overall expression.
    - should we store subexpressions or recompute them?

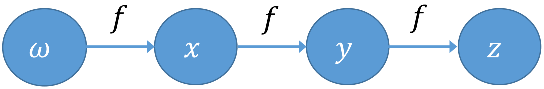
Example of repeated subexpressions

1. Let ω be the input to the graph; Use same function $f : \mathbb{R} \to \mathbb{R}$ at every step: $x = f(\omega), \quad y = f(x), \quad z = f(y)$.

$$\frac{\partial z}{\partial \omega} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \cdot \frac{\partial y}{\partial \omega} = f'(y)f'(x)f'(\omega) \qquad (1)$$

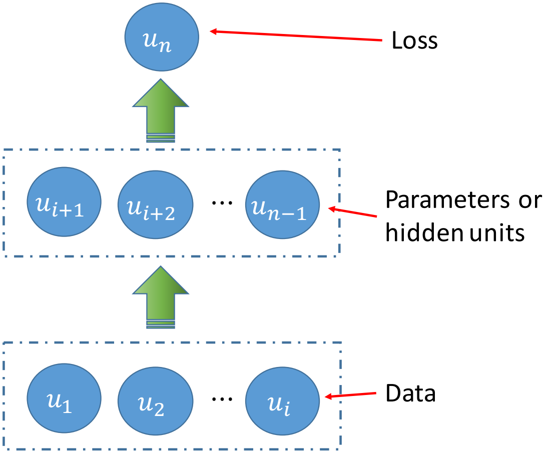
$$= f'(f(f(\omega)))f'(f(\omega))f'(\omega) \qquad (2)$$

2. We can use (1) to compute the derivative, where $f(\omega)$ is computed once and stored it in x.
   - This is the approach taken by backprop.
3. We can also use (2) to compute the derivative, where $f(\omega)$ is recomputed each time it is needed.
   - For low memory, (1) preferable: reduced runtime.
   - (2) is also valid chain rule, useful for limited memory.
4. For complicated graphs as in deep neural networks, (2) exponentially wastes computations for repeated subexpressions.



Basic Backprop Algorithm

- Assume each u_i is a scalar.
- Each $\{u_j\}$ is connected by a subset of other $\{u_k\}_{k \leq j}$, dented as $Pa(u_j)$.
- For Each $\{u_j\}_{j>i}$, $u_j = f^{(j)}(Pa(u_j))$.
- The basic backprop consists of two procedures: the forward pass and the backward pass.

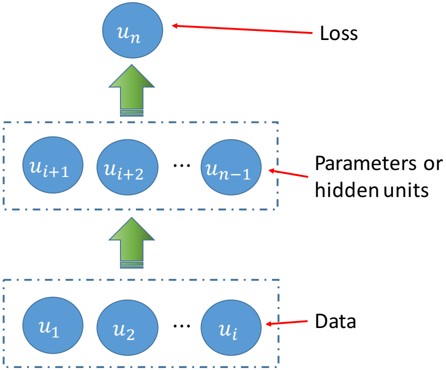


Forward Propagation Computation

```
Input: Data $\{x_i\}$
Output: Loss $\{u_n\}$

for $j = 1, \ldots, i$ do
    $u_i = x_i$
end
for $j = i+1, \ldots, n$ do
    $u_j = f^{(j)}(Pa(u_j))$
end
```

Algorithm 1: Forward Propagation



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

Backward Propagation Computation

1. Proceeds exactly in reverse order of computation in $\mathcal{G}$.
2. Each node in $\mathcal{B}$ computes the derivative $\frac{\partial u_n}{\partial u_i}$.
3. Done by using the chain rule w.r.t. the scalar output u_n:

$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j\in Pa(u_i)} \underbrace{\frac{\partial u_n}{\partial u_i}}_{(1)} \cdot \underbrace{\frac{\partial u_i}{\partial u_j}}_{(2)},$$

where all derivatives in the summation can be computed:

- (1) has been computed in previous step.
- (2) is easily computed given the mapping from u_j to u_i.
- acting like dynamic programming.

Backpropagation Algorithm

```
Input: Data $\{x_i\}$
Output: All derivatives $\{\frac{\partial u_n}{\partial u_j}\}$

Run forward propagation to obtain network activations (outputs)
Initialize grad-table, a data structure that will store derivatives that have
  been computed, The entry grad-table[$u_j$] will store the computed
  value of $\frac{\partial u_n}{\partial u_j}$
grad-table[$u_j$] $\leftarrow 1$
for $j = n-1, \ldots, i+1$ do
    grad-table[$u_j$] $= \sum_{i:j \in Pa(u_i)}$ grad-table[$u_i$]$\frac{\partial u_i}{\partial u_j}$        (*)
end
```

Algorithm 2: Backpropagation Algorithm

Step (*) computes $\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in Pa(u_i)} \frac{\partial u_n}{\partial u_i} \cdot \frac{\partial u_i}{\partial u_j}$.

Computational Complexity & Generalization

1. Computational cost is proportional to the number of edges in the graph (same as for forward pass):
   - each edge is visited a limited number of times.
   - complexity proportional to #edges in the computational graph.
2. Backpropagation thus avoids exponential explosion in repeated subexpressions
   - by simplifications on the computational graph.
3. It can be generalized to the case of intermediate tensor-output by replacing the formula $\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in Pa(u_i)} \frac{\partial u_n}{\partial u_i} \cdot \frac{\partial u_i}{\partial u_j}$ in the BP algorithm by the tensor-version:

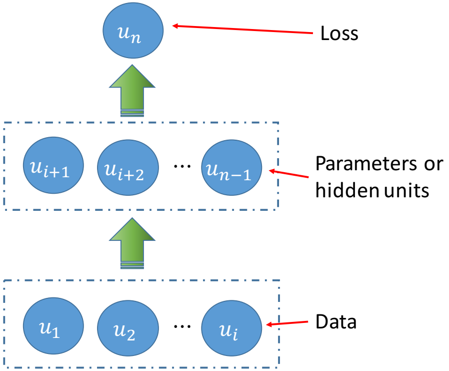
$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in Pa(u_i)} \sum_{\mathbf{k}} \left(\nabla_{u_j} u_{i\,\mathbf{k}}\right) \frac{\partial u_n}{\partial u_{i\,\mathbf{k}}} ,$$

where **k** is the tuple index of a tensor.

Recap: Backward Propagation Computation

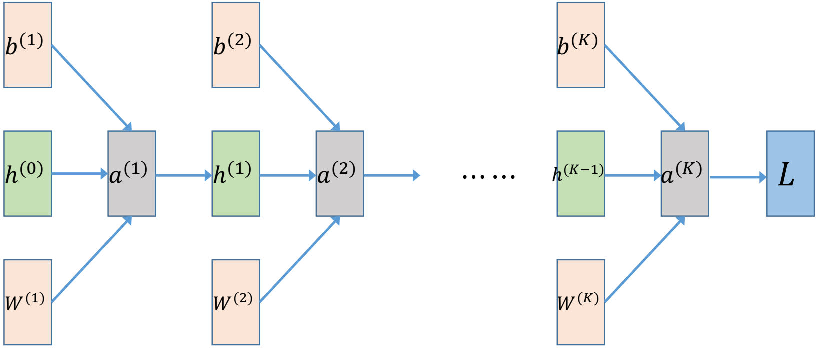
$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in Pa(u_i)} \underbrace{\frac{\partial u_n}{\partial u_i}}_{(1)} \cdot \underbrace{\frac{\partial u_i}{\partial u_j}}_{(2)},$$

- (1) has been computed in previous step.
- (2) is easily computed given the mapping from u_j to u_i.
- When u_i's are vectors, (2) becomes a matrix (Jacobian matrix), and (1) becomes a vector: $\rightarrow$ the result is a vector.



Backprop in Fully Connected MLP

- Maps parameters to supervised loss $L(\hat{y}, y)$ associated with a single training example $(\mathbf{x}, y)$ with $\hat{y}$ the output when $\mathbf{x}$ is the input.



Forward Prop for MLP (without regularization)

Input: Network depth l; Weight matrices $\mathbf{W}^{(i)}, i \in \{1, \ldots, l\}$; bias parameters $\mathbf{b}^{(i)}, i \in 1, \ldots, l$; Input data $\mathbf{x}$ and target output y;
Input: Activation function $f^{(k)}, k \in \{1, \ldots, l\}$
Output: Loss function L

```
$\hbar^{(0)} = \mathbf{x}$
for $k = 1, \ldots, l$ do
    $\mathbf{a}^{(k)} = \mathbf{W}^{(k)^T} h^{(k-1)} + \mathbf{b}^{(k)}$
    $\mathbf{h}^{(k)} = f^{(k)}(\mathbf{a}^{(k)})$
end
$\hat{y} = \mathbf{h}^{(l)}$
Return $L(\bar{y}, y)$
```

Algorithm 3: Forward prop through typical deep NN

- $L(\bar{y}, y)$ is the loss function w.r.t. the network output $\bar{y}$ and the target y, *e.g.*, the cross entropy.

Backward Prop for MLP

```
Forward prop through typical deep NN
$\mathbf{g} \leftarrow \nabla_{\bar{y}} J = \nabla_{\bar{y}} L(\bar{y}, y)$ (gradient of hidden layers, *e.g.*, derivative of softmax on the top-layer)
for $k = l, l-1, \ldots, 1$ do
    Caculate the gradient of the pre-nonlinearity activation $\mathbf{a}^{(k)}$ (element-wise multiplication if $f$ is elementwise)
        //g stores either $\nabla_{\mathbf{a}^{(k)}} J$ or $\nabla_{\mathbf{h}^{(k)}} J$
        $\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \nabla_{f(\mathbf{a}^{(k)})} J \odot \frac{f(\mathbf{a}^{(k)})}{\mathbf{a}^{(k)}} = \mathbf{g} \odot f'(\mathbf{a}^{(k)})$
    Compute the gradients on weights and biases (including the regularization term: $\mathbf{a}^{(k)} = \mathbf{W}^{(k)} \mathbf{h}^{(k-1)} + b^{(k)}$)
        $\nabla_{b^{(k)}} J = \nabla_{\mathbf{a}^{(k)}} J \odot \frac{\partial \mathbf{a}^{(k)}}{\partial b^{(k)}} = \mathbf{g}$
        $\nabla_{\mathbf{W}^{(k)}} J = \nabla_{\mathbf{a}^{(k)}} J \odot \frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{W}^{(k)}} = \mathbf{g} \, {\mathbf{h}^{k-1}}^T$
    Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
    $\mathbf{g} = \nabla_{\mathbf{h}^{(k-1)}} J = \frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{h}^{(k-1)}} \cdot \nabla_{\mathbf{a}^{(k)}} J = {\mathbf{W}^{(k)}}^T \mathbf{g}$
end
```

Implementation: Symbol-to-Symbol Derivatives

1. Algebraic expressions and computational graphs operate on symbolic representations.
2. When we train a neural network we must assign specific values for these symbols, *e.g.*, $[2.5, 3.75, -1.8]^T$.

Two Approaches to BP

1. Symbol-to-number differentiation:
    - Input: computational graph and numerical values of inputs to graph.
    - Output: a set of numerical values describing gradient at the input values.
    - Used by libraries: PyTorch/Torch and Caffe.
2. Symbol-to-symbol differentiation:
    - Input: computational graph.
    - Output: Add additional nodes to the graph.
    - Used by libraries: Theano and Tensorflow.
    - BP 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.

Example: Symbol-to-symbol Derivatives

$$\begin{tikzcd}
z & & z & & \\
y \arrow[u, "f"] & & y \arrow[u, "f"] \arrow[r, "f'"] & \frac{dz}{dy} \arrow[dr] & \\
x \arrow[u, "f"] & \arrow[r, Rightarrow, "\text{BP}"] & x \arrow[u, "f"] \arrow[r, "f'"] & \frac{dy}{dx} \arrow[r, "\times"] & \frac{dz}{dx} \arrow[d, "\times"'] \\
\omega \arrow[u, "f"] & & \omega \arrow[u, "f"] \arrow[r, "f'"] & \frac{dx}{d\omega} \arrow[r] & \frac{dz}{d\omega}
\end{tikzcd}$$

Advantages of Approach

1. Derivatives are described in the same language as the original expression.
2. Because the derivatives are just another computational graph, it is possible to run back-propagation again:
   - differentiating the derivatives.
   - yields higher-order derivatives.

How is BP Implemented

Constructing the Graph

1. Each node in graph $\mathcal{G}$ corresponds to a variable.
2. Each variable is described by a tensor $\mathbf{V}$, subsuming scalars, vectors and matrices.
3. For each node, several routines are implemented associated with the node.

Subroutines Associated with V

- get_operation (**V**):
  - returns the operation that computes **V** represented by the edges coming into **V**.
  - *e.g.*, suppose we have a variable that is computed by matrix multiplication **C** $=$ **A B**, then get_operation (**V**) returns a pointer to an instance of the corresponding C++ class implemented the multiplication.
- get_consumers (**V**, $\mathcal{G}$):
  - returns list of variables that are children of **V** in the computational graph $\mathcal{G}$.
- get_inputs (**V**, $\mathcal{G}$):
  - returns list of variables that are parents of **V** in the computational graph $\mathcal{G}$.

bprop Operation

1. Each operation (node) *op* is also associated with a *bprop* operation.
2. *op.bprop* operation can compute a Jacobian-vector product as described by: $\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$ for input **x** and output **y**.
3. 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 (local operation).

Inputs, Outputs of bprop

1. Backprogation algorithm itself does not need to know any differentiation rules:
   - op.bprop implements it.
   - it only needs to call each operation's op.bprop rules with the right arguments.
2. Formally op.bprop(inputs, $\mathbf{X}$, $\mathbf{D}$) must return:

$$\nabla_{\mathbf{X}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{X}}\right)^T \nabla_{\mathbf{y}} z = \sum_i \left(\nabla_{\mathbf{X}} y_i\right) \mathbf{D}_i$$

   - "inputs" is a list of inputs that are supplied to the operation, $\mathbf{y}$ is a math function that the operation implements.
   - $\mathbf{X}$ is the input whose gradient we wish to compute.
   - $\mathbf{D}$ is the gradient on the output of the operation, *e.g.*, gradient of last layer.
   - Just an implementation of the chain rule: $\nabla_{\mathbf{X}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{X}}\right)^T \underbrace{\nabla_{\mathbf{y}} z}_{\mathbf{D}_i}$.

Software Implementations

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

Formal Backpropagation Algorithm: Outermost Skeleton

```
Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: $z$, the variable to be differentiated
  Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of $z$ and descendents
  of nodes in $\mathbb{T}$.
  Initialize grad_table, a data structure associating tensors to their gradients
  grad_table$[z] \leftarrow 1$
  for $\mathbf{V}$ in $\mathbb{T}$ do
    build_grad$(\mathbf{V}, \mathcal{G}, \mathcal{G}',$ grad_table$)$
  end for
  Return grad_table restricted to $\mathbb{T}$
```

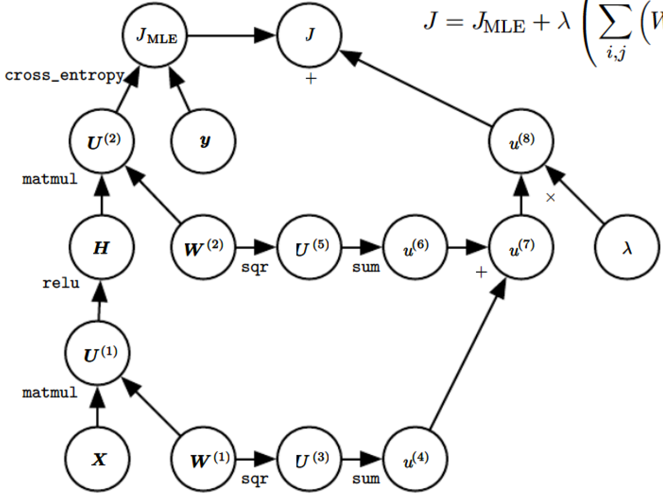
Inner Loop: build-grad ($\mathbf{V}, \mathcal{G}, \mathcal{G}'$, grad-table)

```
Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and grad_table.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: grad_table, a data structure mapping nodes to their gradients
  if $\mathbf{V}$ is in grad_table then
    Return grad_table[$\mathbf{V}$]
  end if
  $i \leftarrow 1$
  for $\mathbf{C}$ in get_consumers($\mathbf{V}, \mathcal{G}'$) do
    op $\leftarrow$ get_operation($\mathbf{C}$)
    $\mathbf{D} \leftarrow$ build_grad($\mathbf{C}, \mathcal{G}, \mathcal{G}'$, grad_table)
    $\mathbf{G}^{(i)} \leftarrow$ op.bprop(get_inputs($\mathbf{C}, \mathcal{G}'$), $\mathbf{V}, \mathbf{D}$)
    $i \leftarrow i + 1$
  end for
  $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
  grad_table[$\mathbf{V}$] $= \mathbf{G}$
  Insert $\mathbf{G}$ and the operations creating it into $\mathcal{G}$
  Return $\mathbf{G}$
```

Example: FNN 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

1. It would be large and tedious for this example.
2. 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.

After gradient computation, it is the responsibility of SGD or other optimization algorithm to use gradients to update parameters.