Advanced Deep Learning

Deep Feedforward Networks - 2

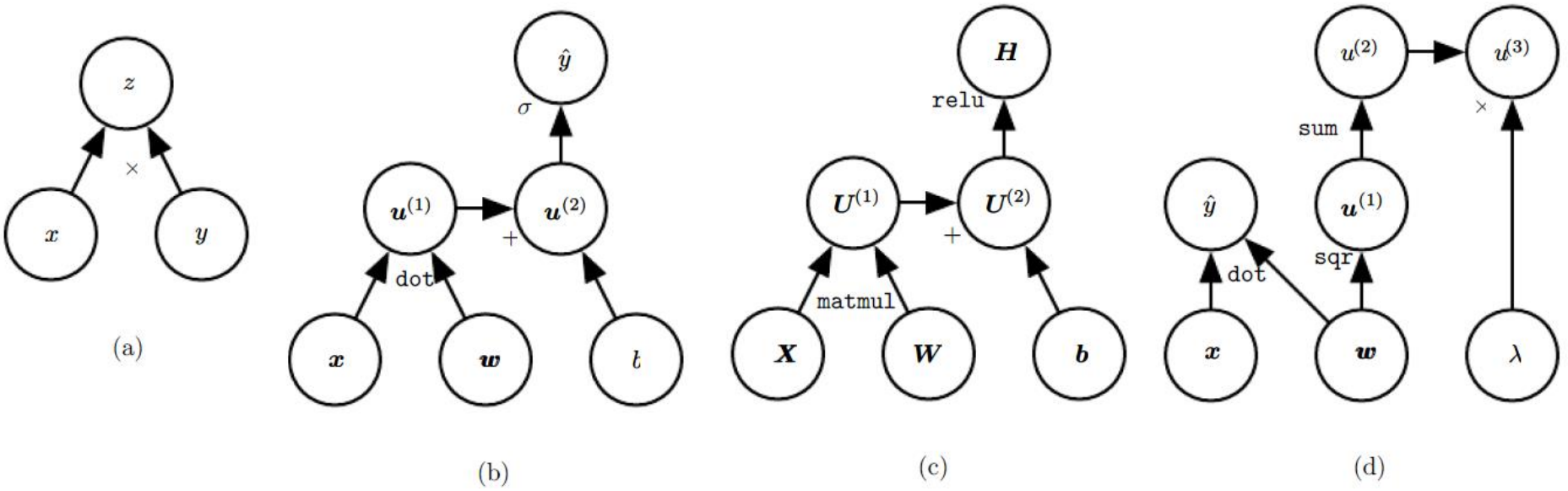
U Kang
Seoul National University

In This Lecture

- Back propagation
  - Motivation
  - Main idea
  - Procedure

Computational Graphs

- Formalizes computation
- Each node indicates a variable
- Each edge indicates an operation



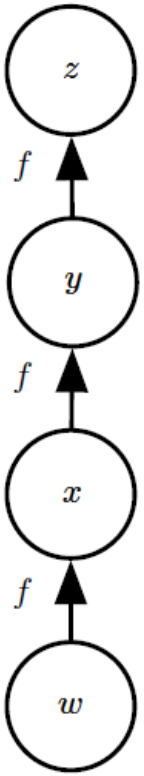
Chain Rule of Calculus

- Let x be a real number, and f and g be functions from R to R. Suppose y = g(x), and z = f(g(x)) = f(y). Then the chain rule states that $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- Suppose $x \in R^m, y \in R^n, g: R^m \to R^n, f: R^n \to R$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$.

  In vector notation: $\nabla_{\boldsymbol{x}} z = (\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}})^T \nabla_{\boldsymbol{y}} z$, where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the n x m Jacobian matrix of g
  - E.g., suppose $z = d^T \boldsymbol{y}$, and $\mathbf{y} = \nabla_{\boldsymbol{x}} g(\boldsymbol{x})$. Then $\nabla_{\boldsymbol{x}} z = (\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}})^T \nabla_{\boldsymbol{y}} z = H^T d$

Repeated Subexpressions

- Computing the same subexpression many times would be wasteful



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

Backpropagation - Overview

- Backpropagation is an algorithm to compute partial derivatives efficiently in neural networks

- The main idea is dynamic programming
  - Many computations share common computations
  - Store the common computations in memory, and read them from memory when needed, without re-computing them from scratch

Backpropagation - Overview

- Assume a computational graph to compute a single scalar $u^{(n)}$
  - E.g., this can be the loss
- Our goal is to compute a partial derivatives $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in \{1,2,\dots,n_i\}$ where $u^{(1)}$ to $u^{(n_i)}$ are the parameters of model
- We assume that nodes are ordered such that we compute their outputs one after the other, starting at $u^{(n_i+1)}$ and going up to $u^{(n)}$
- Each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by $u^{(i)} = f(A^{(i)})$ where $A^{(i)}$ is the set of all parents of $u^{(i)}$

Forward Propagation

- This procedure performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to output $u^{(n)}$

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

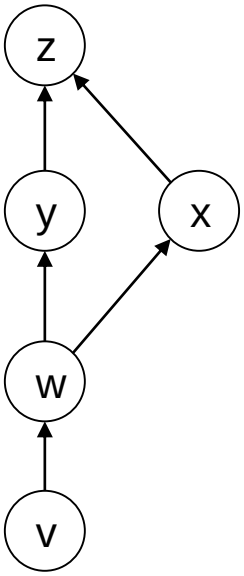
Back-Propagation

- This procedure computes the partial derivatives of $u^{(n)}$ with respect to the variables $u^{(1)}, \ldots, u^{(n_i)}$

Run forward propagation to obtain the activations of the network
Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.
`grad_table`$[\partial u^{(n)}] \leftarrow 1$
for $j = n - 1$ down to 1 **do**
  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:
  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

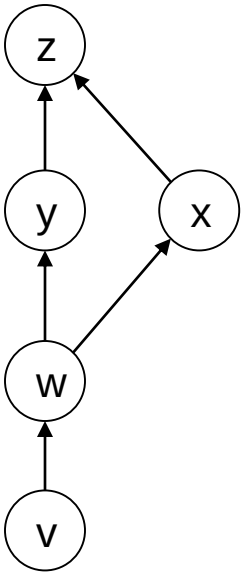
Back-Propagation Example

- Back-propagation procedure (re-use blue-colored computation)
  - Compute $\frac{\partial z}{\partial y}$ and $\frac{\partial z}{\partial x}$
  - $\frac{\partial z}{\partial w} \leftarrow \frac{\partial y}{\partial w}\frac{\partial z}{\partial y} + \frac{\partial x}{\partial w}\frac{\partial z}{\partial x}$
  - $\frac{\partial z}{\partial v} \leftarrow \frac{\partial w}{\partial v}\frac{\partial z}{\partial w}$



Cost of Back-Propagation

- The amount of computation scales *linearly with the number of edges* in the computational graph
  - Computation for each edge: computing a partial derivative, one multiplication, and one addition



Compute $\frac{\partial z}{\partial y}$ and $\frac{\partial z}{\partial x}$

$$\frac{\partial z}{\partial w} \leftarrow \frac{\partial y}{\partial w}\frac{\partial z}{\partial y} + \frac{\partial x}{\partial w}\frac{\partial z}{\partial x}$$

$$\frac{\partial z}{\partial v} \leftarrow \frac{\partial w}{\partial v}\frac{\partial z}{\partial w}$$

Back-Propagation in Fully Connected MLP

- Forward propagation

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Back-Propagation in Fully Connected MLP

- Backward computation

After the forward computation, compute the gradient on the output layer:

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

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

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

⊙: element-wise product

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

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

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

$\frac{\partial J}{\partial W_{11}} = h_1 g_1, \frac{\partial J}{\partial W_{12}} = h_2 g_1$

$\frac{\partial J}{\partial W_{21}} = h_1 g_2, \ldots$

Stochastic Gradient Descent (SGD)

- A recurring problem in ML: large training sets are necessary for good generalization, but large training sets are more computationally expensive

- Cost function using cross-entropy:
  - $J(\theta) = E_{x,y\sim\hat{p}_{data}} L(x, y, \theta) = \frac{1}{m}\sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta)$

    where L is the per-example loss $L(x, y, \theta) = -\log p(y|x; \theta)$
  - Gradient descent requires computing $\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$
  - The computational cost of the gradient descent is O(m) which can take long for large m

- Insight of SGD: gradient is an expectation which can be approximately estimated using a small set of samples

Stochastic Gradient Descent (SGD)

- SGD
  - We sample a minibatch of examples $B = \{x^{(1)}, \ldots, x^{(m')}\}$ drawn uniformly from the training set
  - The minibatch size m' is typically a small number (1 to few hundred)
  - m' is usually fixed as the training set size m grows
  - The estimate of the gradient is $g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$
  - Then the gradient descent is given by $\theta \leftarrow \theta - \epsilon g$

- SGD works well in practice
  - I.e., it often finds a very low value of the cost function quickly enough to be useful

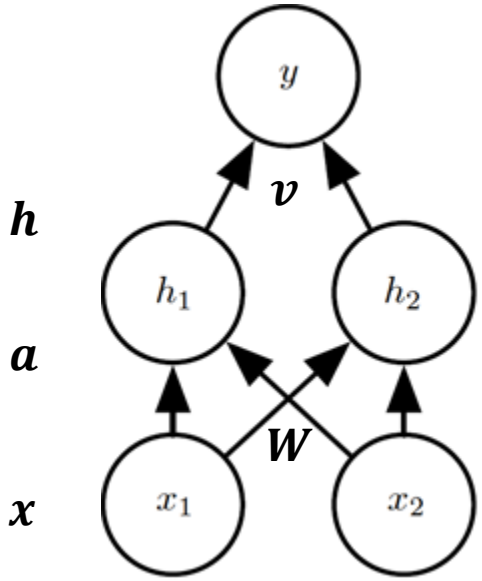
Minibatch Processing

- SGD (recap)
  - We sample a minibatch of examples $B = \{x^{(1)}, \ldots, x^{(m')}\}$ drawn uniformly from the training set
  - The estimate of the gradient is $g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$
  - Then the gradient descent is given by $\theta \leftarrow \theta - \epsilon g$

- SGD using back propagation
  - For each instance $(x^{(i)}, y^{(i)})$, where i=1~m', we compute the gradient $\nabla_\theta J^{(i)}$ using back propagation where $J^{(i)} = L(x^{(i)}, y^{(i)}, \theta)$ is the i-th loss
  - The final gradient is $g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta J^{(i)}$
  - Update $\theta \leftarrow \theta - \epsilon g$

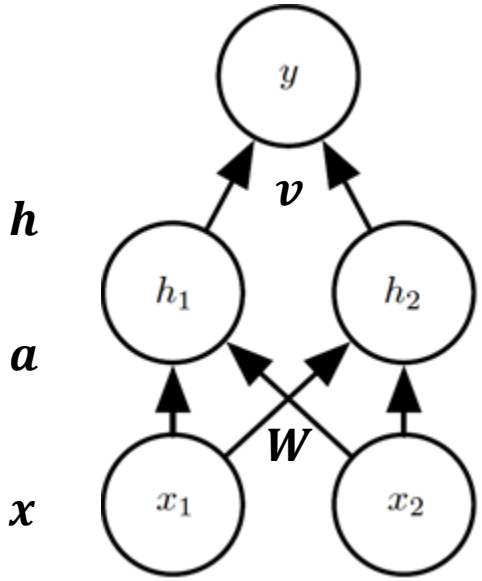
Example

- A feedforward neural network with one hidden layer
- Forward propagation
  - $\boldsymbol{a} \leftarrow \boldsymbol{W}\boldsymbol{x}$
  - $\boldsymbol{h} \leftarrow \boldsymbol{\sigma}(\boldsymbol{a})$ (elementwise)
  - $\hat{\boldsymbol{y}} \leftarrow \boldsymbol{v}^T \boldsymbol{h}$
  - $\boldsymbol{J} \leftarrow L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\boldsymbol{W}, \boldsymbol{v}) = (\hat{\boldsymbol{y}} - \boldsymbol{y})^2 + \lambda(|\boldsymbol{W}|_F^2 + |\boldsymbol{v}|_2^2)$



Example

- Back propagation
  - $g \leftarrow \nabla_{\hat{y}} J = 2(\hat{y} - y)$
  - $\nabla_{\boldsymbol{v}} J \leftarrow \nabla_{\boldsymbol{v}} [(\hat{y} - y)^2 + \lambda(|\boldsymbol{W}|_F^2 + |\boldsymbol{v}|_2^2)] = g\boldsymbol{h} + 2\lambda\boldsymbol{v}$
  - $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}} J = \nabla_{\boldsymbol{h}} [(\hat{y} - y)^2 + \lambda(|\boldsymbol{W}|_F^2 + |\boldsymbol{v}|_2^2)] = g\boldsymbol{v}$
  - $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}} J = \boldsymbol{g} \odot \sigma'(\boldsymbol{a})$ (elementwise)
  - $\nabla_{\boldsymbol{W}} J \leftarrow \nabla_{\boldsymbol{W}} [(\hat{y} - y)^2 + \lambda(|\boldsymbol{W}|_F^2 + |\boldsymbol{v}|_2^2)] = \boldsymbol{g}\boldsymbol{x}^T + 2\lambda\boldsymbol{W}$



$\hat{\boldsymbol{y}} \leftarrow \boldsymbol{v}^T \boldsymbol{h}$
$\boldsymbol{h} \leftarrow \boldsymbol{\sigma}(\boldsymbol{a})$
$\boldsymbol{a} \leftarrow \boldsymbol{W}\boldsymbol{x}$

What You Need to Know

- Backpropagation is an algorithm to compute partial derivatives efficiently in neural networks
  - Reuse partial derivatives
- Procedure of backpropagation
- Use of backpropagation in SGD

Questions?