Chapter 6

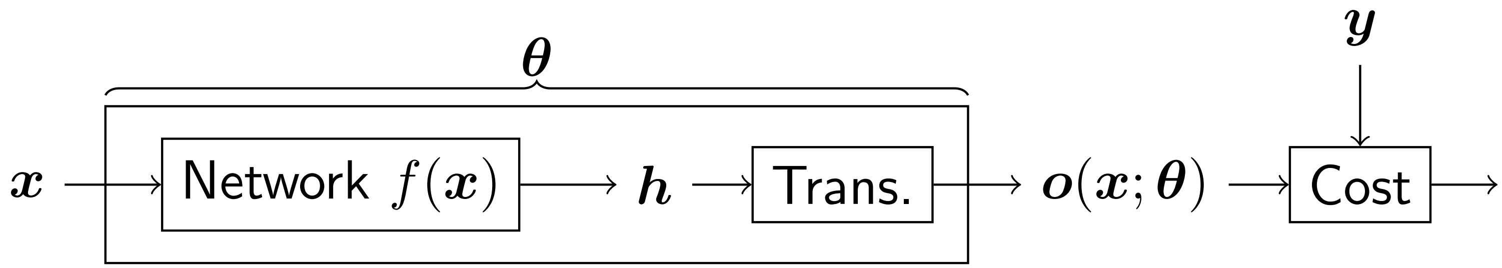
Deep Feedforward Networks

Learning XOR

- Study by yourself

Gradient-based Learning

- General setting



  - $\boldsymbol{x}$: Inputs
  - $f(\boldsymbol{x})$: Feedforward network
  - $\boldsymbol{h}$: Hidden units computed by $f(\boldsymbol{x})$
  - Trans.: Output layer transforming $\boldsymbol{h}$ to output $\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta})$
  - $\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta})$: Output units parameterized by model parameters $\boldsymbol{\theta}$
  - Cost: A function of ground-truth $\boldsymbol{y}$ and output $\boldsymbol{o}$ to be minimized w.r.t. model parameters $\boldsymbol{\theta}$

Cost Functions

- The maximum likelihood (ML) principle provides a guide for designing cost functions
- If we define a conditional distribution $p(y|\boldsymbol{x})$ as a distribution over y parameterized by the network outputs $\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta})$

$$p(y|\boldsymbol{x}) \triangleq p(y;\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta}))$$

- Then, the ML principle suggests we take the negative log-likelihood

$$-\log p(y;\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta}))$$

as the cost function to be minimized w.r.t. model parameters $\boldsymbol{\theta}$

Learning Conditional Statistics

- If we define

$$p(y|\boldsymbol{x}) \triangleq \mathcal{N}(y; o(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I}),$$

- Then, minimizing the negative log-likelihood yields

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} E_{\boldsymbol{x},y\sim p_{\mathsf{data}}} \|y - o(\boldsymbol{x};\boldsymbol{\theta})\|^2$$

- With sufficient capacity, the network will learn the conditional mean

$$o(\boldsymbol{x};\boldsymbol{\theta}^*) \approx E[y|\boldsymbol{x}],$$

which predicts the mean value of y for each $\boldsymbol{x}$

- By the same token, if we define

$$p(y|\boldsymbol{x}) \triangleq \mathsf{Laplace}(y; o(\boldsymbol{x};\boldsymbol{\theta}), \gamma) = \frac{1}{2\gamma} exp\left(-\frac{|y-o(\boldsymbol{x};\boldsymbol{\theta})|}{\gamma}\right),$$

- Then, minimizing the negative log-likelihood yields

$$\boldsymbol{\theta^*} = \arg\min_{\boldsymbol{\theta}} E_{\boldsymbol{x},y\sim p_{\mathsf{data}}} |y - o(\boldsymbol{x};\boldsymbol{\theta})|$$

- With sufficient capacity, the network will learn the conditional median

$$o(\boldsymbol{x};\boldsymbol{\theta}^*) \approx \mathsf{Median}[y|\boldsymbol{x}],$$

  which predicts the median value of y for each $\boldsymbol{x}$

- In other words, $o(\boldsymbol{x};\boldsymbol{\theta}^*)$ satisfies

$$\int_{-\infty}^{o(\boldsymbol{x};\boldsymbol{\theta}^*)} p(y|\boldsymbol{x})dy = \int_{o(\boldsymbol{x};\boldsymbol{\theta}^*)}^{\infty} p(y|\boldsymbol{x})dy$$

Learning Gaussian Output Distributions

- The n-dimensional Gaussian distribution $\mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is given by

$$\mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp(-\frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{y} - \boldsymbol{u}))$$

- A conditional Gaussian can be learned by treating $o(\boldsymbol{x}; \boldsymbol{\theta})$ as the means

$$p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; o(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{I})$$

- In this model, the outputs $o(\boldsymbol{x}; \boldsymbol{\theta})$ often take a linear form

$$o(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{W}^T \boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{b}$$

- As such, they are known as *linear output units*

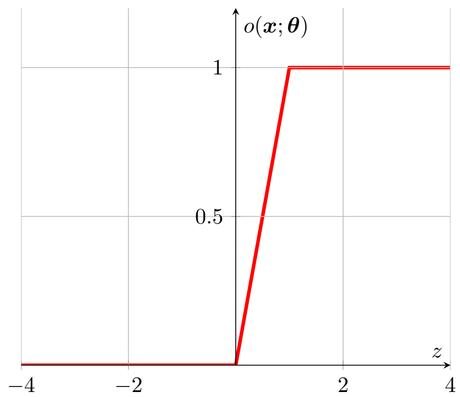
Learning Bernoulli Output Distributions

- A conditional Bernoulli can be learned by having $o(\boldsymbol{x};\boldsymbol{\theta})$ as the very distribution parameter

$$p(y|\boldsymbol{x}) = o(\boldsymbol{x};\boldsymbol{\theta})^y(1-o(\boldsymbol{x};\boldsymbol{\theta}))^{1-y},\ y \in \{0,1\}$$

- The $o(\boldsymbol{x};\boldsymbol{\theta})$ must be in $[0,1]$ in order to be a valid parameter, one trivial implementation being

$$o(\boldsymbol{x};\boldsymbol{\theta}) = \max\{0, \min\{1,z\}\},\ z = \boldsymbol{w}^T h(\boldsymbol{x}) + b$$

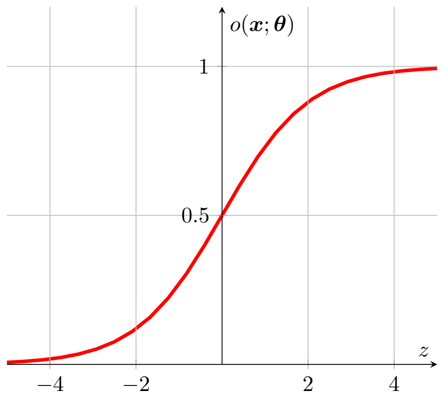


- This however may be problematic for the gradient of the log-likelihood w.r.t. $\boldsymbol{w}, b$ is zero when $z = \boldsymbol{w}^T h(\boldsymbol{x}) + b$ is outside the unit interval

$$-\nabla_{\boldsymbol{w},b} \log p(y|\boldsymbol{x}) = -\frac{\partial \log p(y|\boldsymbol{x})}{\partial o(\boldsymbol{x};\boldsymbol{\theta})} \underbrace{\frac{\partial o(\boldsymbol{x};\boldsymbol{\theta})}{\partial z}}_{=0} \nabla_{\boldsymbol{w},b} z$$

- The gradient-based learning may fail to learn $\boldsymbol{w}, b$ properly
- An alternative approach is to use sigmoid units combined with the negative log-likelihood function

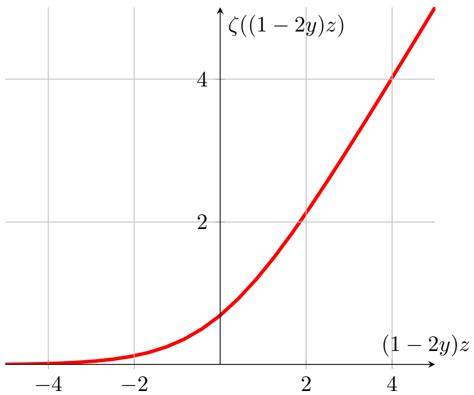
$$o(\boldsymbol{x};\boldsymbol{\theta}) = \sigma(z), \text{ with } z = \boldsymbol{w}^T h(\boldsymbol{x}) + b, \ \sigma(z) = \frac{e^z}{1 + e^z}$$



- By definition, the negative log-likelihood can be computed as

$$
\begin{aligned}
-\log p(y|\boldsymbol{x}) &= -\log o(\boldsymbol{x};\boldsymbol{\theta})^y(1-o(\boldsymbol{x};\boldsymbol{\theta}))^{1-y} \\
&= -\log \sigma((2y-1)z) \\
&= \zeta((1-2y)z)
\end{aligned}
$$

where $y \in \{0,1\}$ and $\zeta(\cdot)$ is the softplus function



- Ideally, we want $z = \boldsymbol{w}^T h(\boldsymbol{x}) + b$ to be very positive (respectively, negative) when the ground-true $y = 1$ (respectively, $y = 0$)
- This suggests that samples correctly classified have very negative

$(1-2y)z$; in other words, these samples will not contribute to the gradient computation

$$-\nabla_{\boldsymbol{w},b} \log p(y|\boldsymbol{x}) = -\frac{\partial \log p(y|\boldsymbol{x})}{\partial \zeta} \underbrace{\frac{\partial \zeta}{\partial (1-2y)z}}_{=0} \frac{\partial (1-2y)z}{\partial z} \nabla_{\boldsymbol{w},b} z$$

- On the other hand, samples incorrectly classified have very positive $(1-2y)z$, which leads to strong gradients
- When sigmoid units are combined with other cost functions, e.g. mean squared error, the gradient vanishing problem may occur
- *The choice of output units is tightly coupled with that of cost functions*

Multinoulli Random Variables

- A Multinoulli random variable x has n possible values with distribution

$$P(x; \boldsymbol{\alpha}) = \prod_{i=1}^{n} (\alpha_i)^{\mathbf{1}_{x=i}}, \; x = 1, 2, ..., n$$

where

$$\alpha_i \in [0, 1], \; \forall i$$

$$\sum_{i=1}^{n} \alpha_i = 1$$

$$\mathbf{1}_{x=i} = \begin{cases} 1, & \text{if } x = i \\ 0, & \text{if } x \neq i \end{cases}$$

- When using a one-hot vector (or the 1-of-K coding) $\boldsymbol{x}$, in which only one element equals to 1 with the others being 0, to represent the n

possible values, the Multinoulli distribution is given by

$$P(\boldsymbol{x}; \boldsymbol{\alpha}) = \prod_{i=1}^{n} (\alpha_i)^{x_i}, \ x_i \in [0, 1]$$

- Observe that Bernoulli is a special case of Multinoulli with $n = 2$

Learning Multinoulli Output Distributions

- A conditional Multinoulli can be learned by having its distribution parameters be modeled by softmax output units $o(\boldsymbol{x};\boldsymbol{\theta})$

$$p(\boldsymbol{y}|\boldsymbol{x}) = \prod_{i=1}^{n} (o(\boldsymbol{x};\boldsymbol{\theta})_i)^{y_i},$$

where $\boldsymbol{y}$ is a one-hot vector and

$$o(\boldsymbol{x};\boldsymbol{\theta})_i = \mathsf{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$$

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{b}$$

- When the difference between the maximal element of $\boldsymbol{z}$ and the others becomes large, softmax becomes a form of winner-take-all, namely, one output is nearly 1 and the others are nearly 0
- As such, it is viewed as a softened version of $\arg\max$ with a one-hot

representation

- Softmax overparameterizes the distribution: the n-th probability may be obtained by subtracting the first $n-1$ probabilities from 1
- One may require that one element of $\boldsymbol{z}$ be 0; this leads to sigmoid when $n = 2$

$$\sigma(z) = \frac{\exp(z)}{\exp(z) + \exp(0)} = \frac{1}{1 + \exp(-z)}$$

- As with sigmoid units, softmax units almost always come with the log-likelihood function

$$-\log p(\boldsymbol{y}|\boldsymbol{x}) = -\sum_{i=1}^{n} y_i \log(o(\boldsymbol{x};\boldsymbol{\theta})_i) = -\sum_{i=1}^{n} y_i (z_i - \log \sum_{j=1}^{n} \exp(z_j))$$

- Assuming $y_i = 1$ and $y_j = 0, j \neq i$, we have

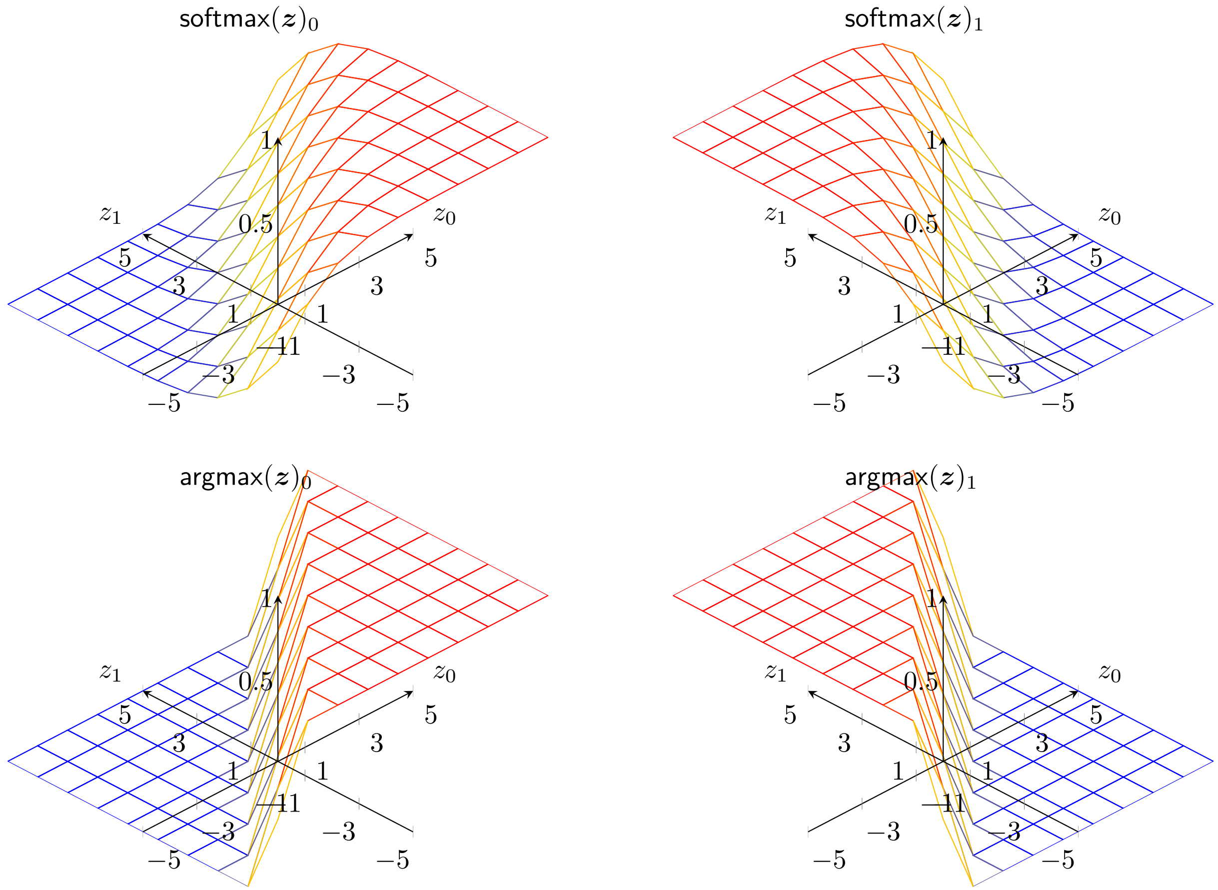
$$-\log p(\boldsymbol{y}|\boldsymbol{x}) = -(z_i - \log \sum_{j=1}^{n} \exp(z_j))$$

- This suggests that when the ground truth $y_i = 1$, minimizing the negative log-likelihood amounts to maximizing z_i and penalizes the most active incorrect prediction z_j if $z_j \gg z_k, k \neq j$

- On the other hand, samples correctly classified (i.e., $z_i \gg z_k, k \neq i$) will contribute little to the gradient computation; in this case, $(z_i - \log \sum_{j=1}^{n} \exp(z_j)) \approx 0$

- Observe that the log undoes the saturating effect of the softmax

Softmax vs. Argmax



Relative Frequencies

- With sufficient capacity, minimizing the negative log-likelihood will

$$-E_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{\text{data}}} \log p(\boldsymbol{y}|\boldsymbol{x}) = -E_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{\text{data}}} \left[\sum_{i=1}^{n} y_i \log o(\boldsymbol{x};\boldsymbol{\theta})_i\right]$$

$$= -\sum_{j=1}^{m}\sum_{i=1}^{n} y_i^{(j)} \log o(\boldsymbol{x}^{(j)};\boldsymbol{\theta})_i,$$

drive output units to approximate relative frequencies in training data

$$o(\boldsymbol{x};\boldsymbol{\theta})_i \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x},\, y_i^{(j)}=1}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}$$

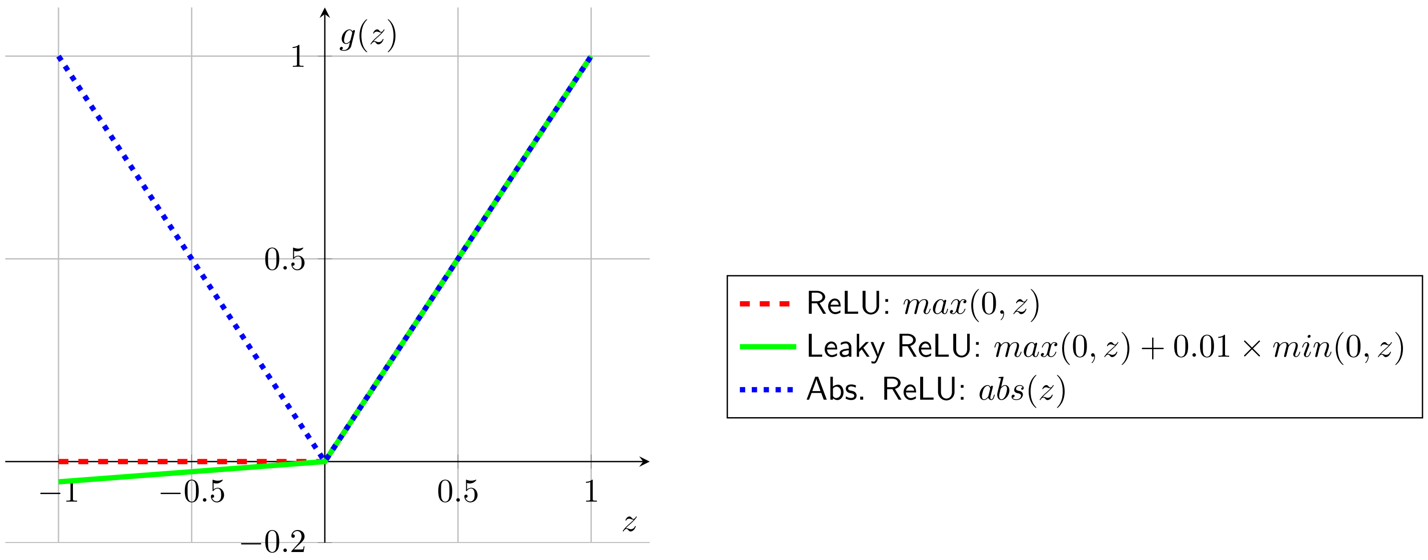
- To see this, solve the following constrained optimization problem:

$$\arg\min_{o(\boldsymbol{x};\boldsymbol{\theta})_i} \left[-\sum_{j:\boldsymbol{x}^{(j)}=\boldsymbol{x}}\sum_{i=1}^{n} y_i^{(j)} \log o(\boldsymbol{x};\boldsymbol{\theta})_i\right] \text{ s.t. } \sum_{i=1}^{n} o(\boldsymbol{x};\boldsymbol{\theta})_i = 1$$

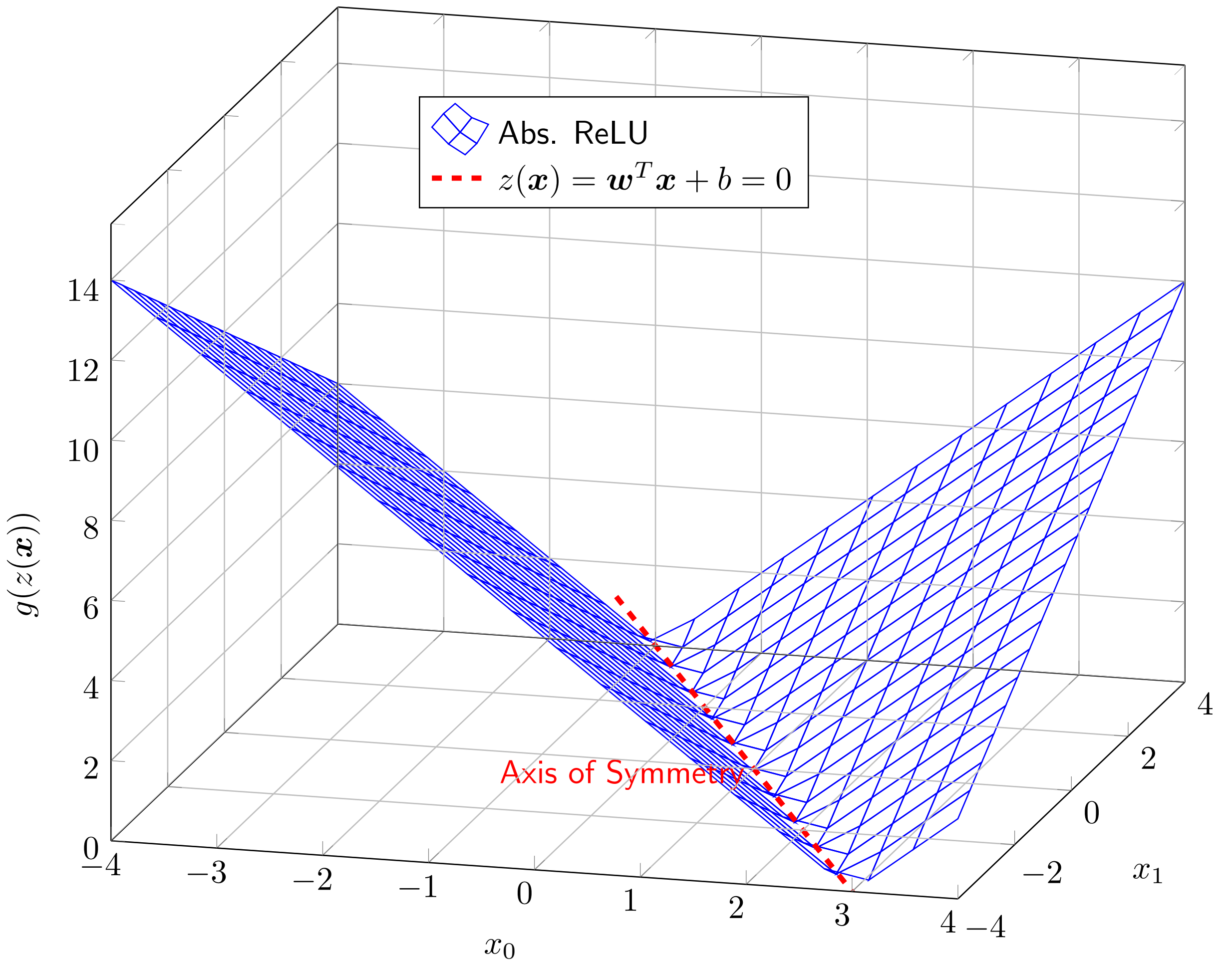
- Essentially, we are using the network to encode/approximate relative frequencies for different combinations of $\boldsymbol{x}, \boldsymbol{y}$
- How should we interpret the values of $o(\boldsymbol{x}; \boldsymbol{\theta})$ if we input an $\boldsymbol{x}$ that is not included in training data?

Hidden Units and Activation Functions

- Computation of hidden units $\boldsymbol{h}$
  1. Accepting inputs $\boldsymbol{x}$
  2. Applying affine transformation $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$
  3. Applying element-wise non-linear mapping $\boldsymbol{h} = g(\boldsymbol{z})$
- Typical $g(z)$, a.k.a. activation functions



Activation Functions in Input Space



Maxout Units

- Formulation

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}(i)} z_j$$

where

$$\mathbb{G}(i) = \{(i-1)k+1, (i-1)k+2, ..., (i-1)k+k\}$$

- Maxout can implement many typical activation functions
  - Given

$$z_1 = \boldsymbol{w}_1^T \boldsymbol{x} + b_1$$
$$z_2 = \boldsymbol{w}_2^T \boldsymbol{x} + b_2$$

  - ReLU

$$g(z_1) = \max(z_1, z_2)$$

$$\text{with } z_2 = 0 \text{ by } (\boldsymbol{w_2}, b2) = \mathbf{0}$$

– Leaky ReLU

$$g(z_1) = \max(z_1, z_2)$$
$$\text{with } z_2 = 0.01 \times z_1 \text{ by } (\boldsymbol{w_2}, b2) = 0.01 \times (\boldsymbol{w_1}, b1)$$

- Maxout can be seen as learning the activation function, in the sense that $(\boldsymbol{w_1}, b_1)$ and $(\boldsymbol{w_2}, b_2)$ can both be learned
- Maxout can approximate any piecewise linear, convex function in input space $\boldsymbol{x}$, when more pieces $\{z_i\}$ are input
- ReLU and all these variants exhibit linear behavior

Why Linearity?

- Toy problem: Training a network with only one hidden unit

$$\boldsymbol{x} \xrightarrow[\text{Affine}]{f(\boldsymbol{x})=\boldsymbol{w}^T\boldsymbol{x}+b} z \xrightarrow[\text{Activation}]{g(z)} h \xrightarrow[\text{Likelihood}]{c(h)} J$$

- Partial derivative of J w.r.t model parameter w_i is given by

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial h} \times \underbrace{\frac{\partial h}{\partial z}} \times \frac{\partial z}{\partial w_i} = c'(h) \times \underbrace{g'(z)} \times x_i$$

- Linearity in activation (i.e. $\partial h/\partial z$ is some non-zero constant) helps ensure a well-behaved gradient w.r.t. model parameters

Universal Approximation Theorem

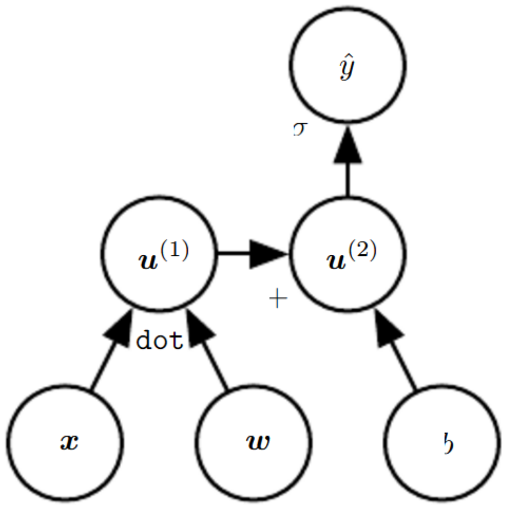
- (Hornik et al., 1989; Cybenko, 1989) A feedforward network with one linear output layer and at least one hidden layer with squashing activation function (e.g sigmoid) and enough hidden units can approximate any Borel measurable function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with any degree of accuracy
- Any continuous function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable
- (Leshno et al., 1993) The theorem holds true for networks with rectified linear activation functions

How Many Are Enough?

- (Montufar et al., 2014) Functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow network of one hidden layer (Check the paper for details)
- Empirically, deeper models can use fewer units to represent the desired function and generalize better

Computational Graphs

- To formalize computation as a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$
  - A node $v \in \mathcal{V}$ indicates a variable
  - A directed edge $e \in \mathcal{E}$ from x to y indicates that y is computed by applying an operation to x
- Example: Logistic regression prediction $\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b$



- Observe that some nodes represent model parameters $\boldsymbol{w}, b$

Chain Rule of Calculus

- To compute the derivative dz/dx of a function $z(x)$ formed by the composition of functions $z(x) = f(g(x))$

$$x \xrightarrow{g(x)} y \xrightarrow{f(y)} z$$

  - $x \in \mathbb{R}$ is a real number
  - $f, g : \mathbb{R} \rightarrow \mathbb{R}$ are real-valued functions of single variable

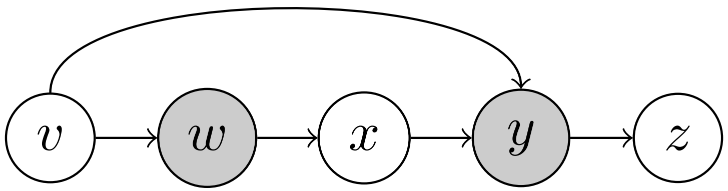
- The chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- As an extension, we have for the following graph

$$\frac{dz}{dv} = \frac{dz}{dy}\frac{dy}{dv} + \frac{dz}{dw}\frac{dw}{dv} = \sum_{n:v \in Pa(n)} \frac{dz}{dn}\frac{dn}{dv}$$

where $Pa(n)$ is the set of nodes that are parents of n



- To verify the result requires another chain rule from calculus

$$z = f(y_1, y_2),\ y_1 = g_1(x),\ y_2 = g_2(x)$$

$$\frac{dz}{dx} = \frac{dz}{dy_1}\frac{dy_1}{dx} + \frac{dz}{dy_2}\frac{dy_2}{dx}$$

Vector Case

- z is a scalar, and $\boldsymbol{x}, \boldsymbol{y}$ are vectors

$$\boldsymbol{x}_{m\times 1} \xrightarrow{g(\boldsymbol{x})} \boldsymbol{y}_{n\times 1} \xrightarrow{f(\boldsymbol{y})} z_{1\times 1}$$

- Applying the chain rule leads to

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}, \ \ i = 1, 2, ..., m$$

- In matrix notation,

$$\begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \frac{\partial y_2}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \\ \vdots \\ \frac{\partial z}{\partial y_m} \end{bmatrix}$$

- This is recognized as a **Jacobian-gradient product**

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

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ (abbrev. $J_{y,x}$) is known as **Jacobian matrix** with

$$(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}})_{i,j} = \frac{\partial y_i}{\partial x_j}$$

As an example, when $\boldsymbol{y} = g(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}$ (i.e. $y_i = \sum_j w_{i,j} x_j$),

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \boldsymbol{W}$$

Matrix Case

- z is a scalar, and $\boldsymbol{X}, \boldsymbol{Y}$ are matrices

$$\boldsymbol{X}_{m\times n} \xrightarrow{g(\boldsymbol{X})} \boldsymbol{Y}_{s\times k} \xrightarrow{f(\boldsymbol{Y})} z_{1\times 1}$$

- The chain rule suggests that

$$\frac{\partial z}{\partial x_{i,j}} = \sum_{s,k} \frac{\partial z}{\partial y_{s,k}} \frac{\partial y_{s,k}}{\partial x_{i,j}}, \ \forall i, j$$

- More generally, when $\boldsymbol{X}, \boldsymbol{Y}$ are tensors (high-dimensional arrays),

$$\nabla_{\boldsymbol{X}} z = \sum_j (\frac{\partial z}{\partial \mathrm{Y}_j}) \nabla_{\boldsymbol{X}} Y_j,$$

where $\boldsymbol{Y} = g(\boldsymbol{X}), z = f(\boldsymbol{Y})$ and $\boldsymbol{X}$ is treated as if it were a vector

- **Example 1:** Assume $\boldsymbol{Y} = g(\boldsymbol{X}) = \boldsymbol{W}\boldsymbol{X}$
  - Let $\nabla_{\boldsymbol{Y}} z$ denote a matrix with its element (i, j) given by $\partial z / \partial Y_{i,j}$
  - And $\nabla_{\boldsymbol{X}} z$ be a matrix with its element (i, j) given by $\partial z / \partial X_{i,j}$
  - Observe that each column $\boldsymbol{Y}_{:,j}$ of $\boldsymbol{Y}$ is a function of the corresponding column $\boldsymbol{X}_{:,j}$ in $\boldsymbol{X}$, i.e. $\boldsymbol{Y}_{:,j} = \boldsymbol{W}\boldsymbol{X}_{:,j}$
  - We apply the Jacobian-gradient product of vector form to obtain

$$\nabla_{\boldsymbol{X}_{:,j}} z = \boldsymbol{W}^T \nabla_{\boldsymbol{Y}_{:,j}} z \Rightarrow \underbrace{\nabla_{\boldsymbol{X}} z = \boldsymbol{W}^T \nabla_{\boldsymbol{Y}} z}$$

- **Example 2:** Assume $\boldsymbol{Y} = g(\boldsymbol{X}) = \boldsymbol{X}\boldsymbol{W}$
  - Observe that each row $\boldsymbol{Y}_{i,:}$ of $\boldsymbol{Y}$ is a function of the corresponding row $\boldsymbol{X}_{i,:}$ in $\boldsymbol{X}$, i.e., $\boldsymbol{Y}_{i,:} = \boldsymbol{X}_{i,:}\boldsymbol{W}$, or equivalently, $\boldsymbol{Y}_{i,:}^T = \boldsymbol{W}^T \boldsymbol{X}_{i,:}^T$
  - Applying the Jacobian-gradient product of vector form yields

$$\nabla_{\boldsymbol{X}_{i,:}^T} z = \boldsymbol{W} \nabla_{\boldsymbol{Y}_{i,:}^T} z \Rightarrow \underbrace{\nabla_{\boldsymbol{X}} z = (\nabla_{\boldsymbol{Y}} z)\boldsymbol{W}^T}$$

Back-Propagation

- Toy problem: To compute the derivative of J w.r.t. x

$$x \xrightarrow{f(x)} z \xrightarrow{g(z)} h \xrightarrow{c(h)} J$$

- From the chain rule, we have

$$\begin{aligned}\frac{\partial J}{\partial x} &= \frac{\partial J}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial x} \\ &= c'(h)g'(z)f'(x) \\ &= c'(g(f(x)))g'(f(x))f'(x)\end{aligned}$$

- There are two possible implementations
  - The one based on the last equality incurs redundant subcomputation (e.g. $f(x)$)
  - The other following the penultimate equality requires z, h be pre-computed through forward propagation

- Now, assuming z, h have been pre-computed, one way to compute the derivatives of J w.r.t. all variables x, z, h is to proceed in the order of

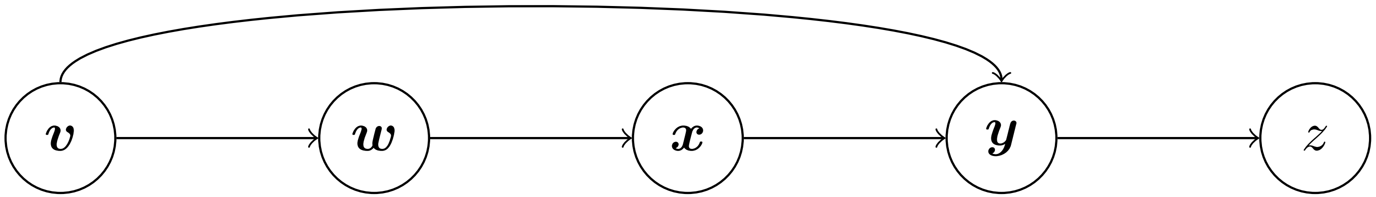
  1. $\partial J/\partial h = c'(h)$
  2. $\partial J/\partial z = (\partial J/\partial h)g'(z)$
  3. $\partial J/\partial x = (\partial J/\partial z)f'(x)$

  and, at each step, keep the result for subsequent use

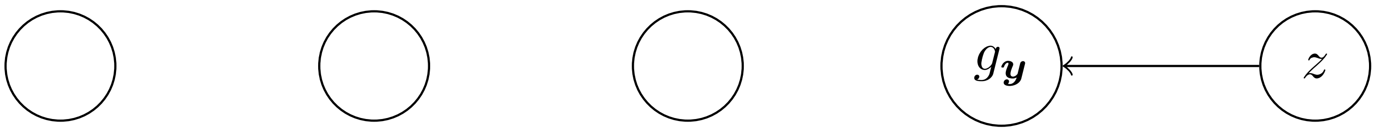
- This technique is known as the **back-propagation (backprop)** method

General Back-Propagation

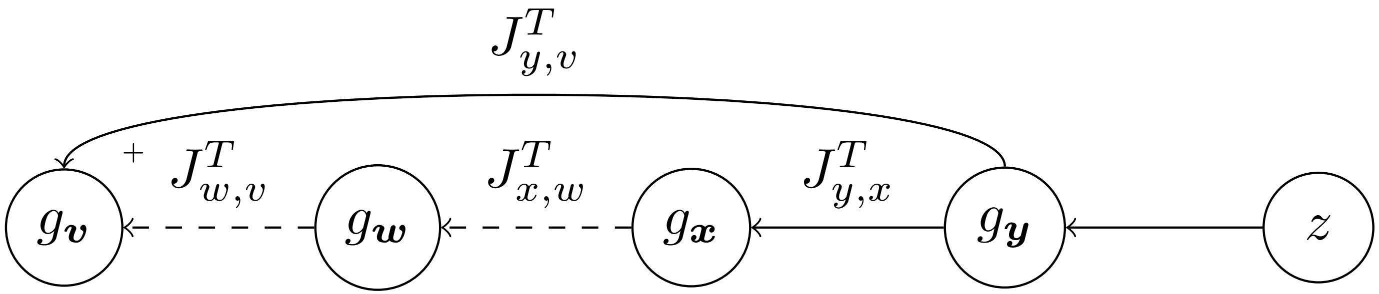
- To compute the gradient of z w.r.t. all its ancestors $\boldsymbol{y}, \boldsymbol{x}, \boldsymbol{w}, \boldsymbol{v}$



- Compute gradient w.r.t. every parent of z



- Travel backward, multiply current gradient by Jacobian recursively



- Sum gradients from different paths

- In symbols, we have

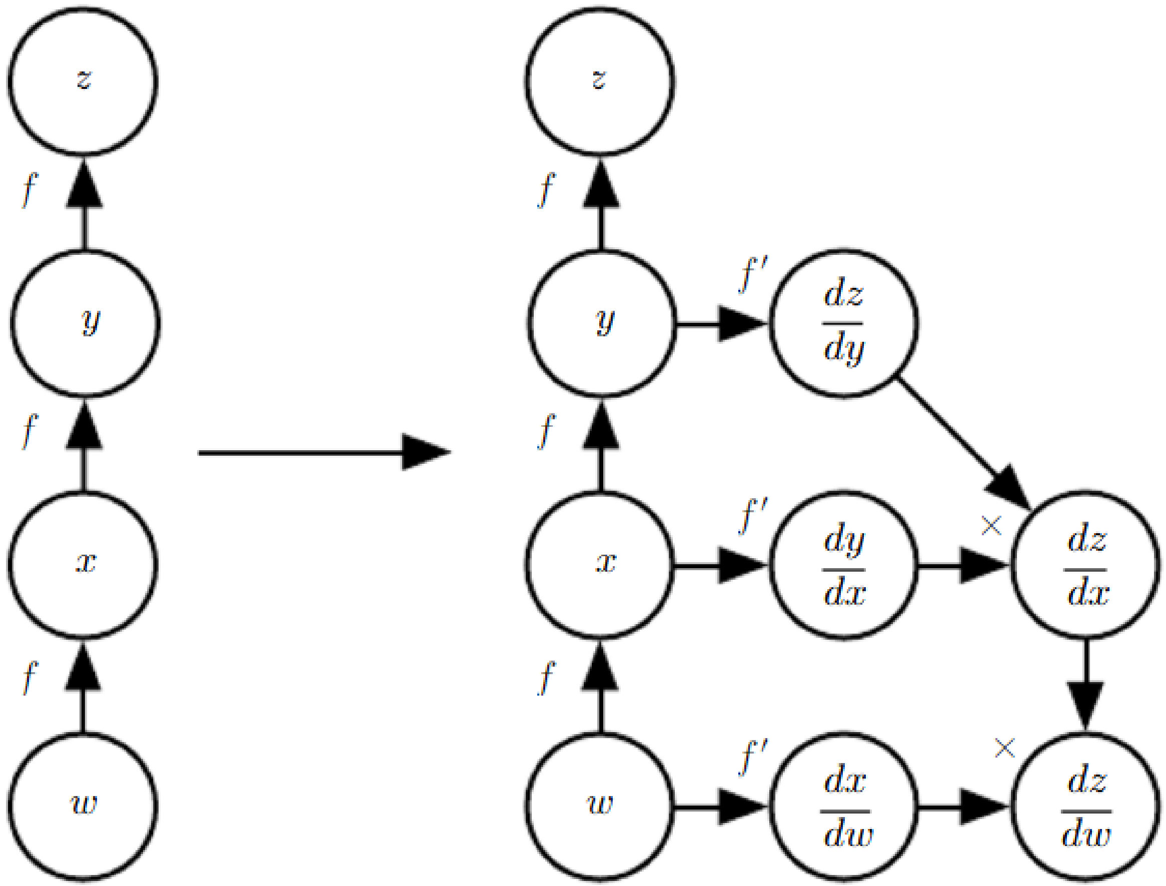
$$
\begin{aligned}
g_{\boldsymbol{y}} &= \nabla_{\boldsymbol{y}} z \\
g_{\boldsymbol{x}} &= \nabla_{\boldsymbol{x}} z = J_{y,x}^T g_{\boldsymbol{y}} \\
g_{\boldsymbol{w}} &= \nabla_{\boldsymbol{w}} z = J_{x,w}^T g_{\boldsymbol{x}} \\
g_{\boldsymbol{v}} &= \nabla_{\boldsymbol{v}} z = J_{w,v}^T g_{\boldsymbol{w}} + J_{y,v}^T g_{\boldsymbol{y}}
\end{aligned}
$$

where

$$
\begin{aligned}
J_{y,x} &= \partial \boldsymbol{y} / \partial \boldsymbol{x} \\
J_{x,w} &= \partial \boldsymbol{x} / \partial \boldsymbol{w} \\
J_{w,v} &= \partial \boldsymbol{w} / \partial \boldsymbol{v} \\
J_{y,v} &= \partial \boldsymbol{y} / \partial \boldsymbol{v}
\end{aligned}
$$

- The spirit of this procedure can extend to cases where $\boldsymbol{y}, \boldsymbol{x}, \boldsymbol{w}, \boldsymbol{v}$ are matrices, tensors, vectors, scalars, or their mixing combinations

Backprop in Computational Graphs



Backprop for MLP Training

- Input $\boldsymbol{X}$ in mini-batch form

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_0^T \\ \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_{m-1}^T \end{bmatrix}$$

- One layer of hidden features $\boldsymbol{H}$ with ReLU

$$\boldsymbol{U}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)},$$

$$\boldsymbol{H} = \max\{0, \boldsymbol{U}^{(1)}\}$$

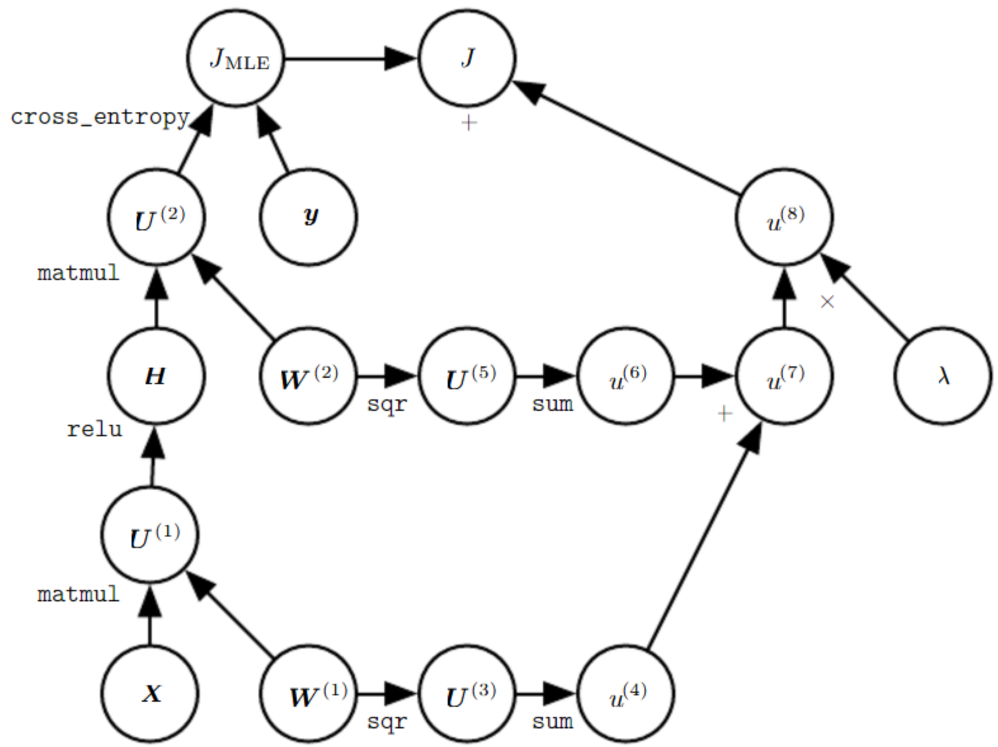
- One layer of outputs $\boldsymbol{U}^{(2)}$(before normalization)

$$\boldsymbol{U}^{(2)} = \boldsymbol{H}\boldsymbol{W}^{(2)}$$

- Objective: To minimize the cross-entropy with weight decay

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

- Computational graph



- Backprop: To compute $\nabla_{\boldsymbol{W}^{(1)}} J$ and $\nabla_{\boldsymbol{W}^{(2)}} J$
  - Two paths from J to the weights (only one path illustrated)
  - Assume $\nabla_{\boldsymbol{U}^{(2)}} J = \boldsymbol{G}$
  - Then $\nabla_{\boldsymbol{W}^{(2)}} J = \boldsymbol{H}^T \boldsymbol{G}$ (cf. **Example 1** in Matrix Case)
  - Similarly, $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)T}$ (cf. **Example 2** in Matrix Case)
  - Tracing back further, we have $\nabla_{\boldsymbol{U}^{(1)}} J = \boldsymbol{G}'$ by zeroing out elements in $\nabla_{\boldsymbol{H}} J$ corresponding to entries of $\boldsymbol{U}^{(1)}$ less than zero
  - Again, $\nabla_{\boldsymbol{W}^{(1)}} J = \boldsymbol{X}^T \boldsymbol{G}'$ (cf. **Example 1** in Matrix Case)

Review

- Maximum likelihood as cost functions
- Output units (linear/sigmoid/softmax) as model parameters
- Cost functions and output units: gradient vanishing issues
- Hidden units and activation functions
- Universal approximation theorem: depth vs. width
- Chain rules of calculus
- Back propagation