Deep Neural Networks

ACMS 80770: Deep Learning with Graphs

Instructor: Navid Shervani-Tabar

Department of Applied and Comp Math and Stats

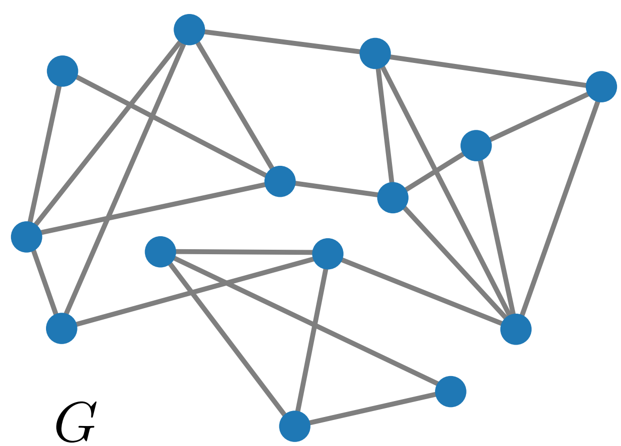
UNIVERSITY OF NOTRE DAME

Shallow Embedding

- The node embedding methods discussed so far learn based on a shallow embedding.
- Shallow embedding methods have a few shortcomings:
  - They do not **share parameters** within nodes in the encoder.
  - This is computationally more expensive and statistically less efficient.

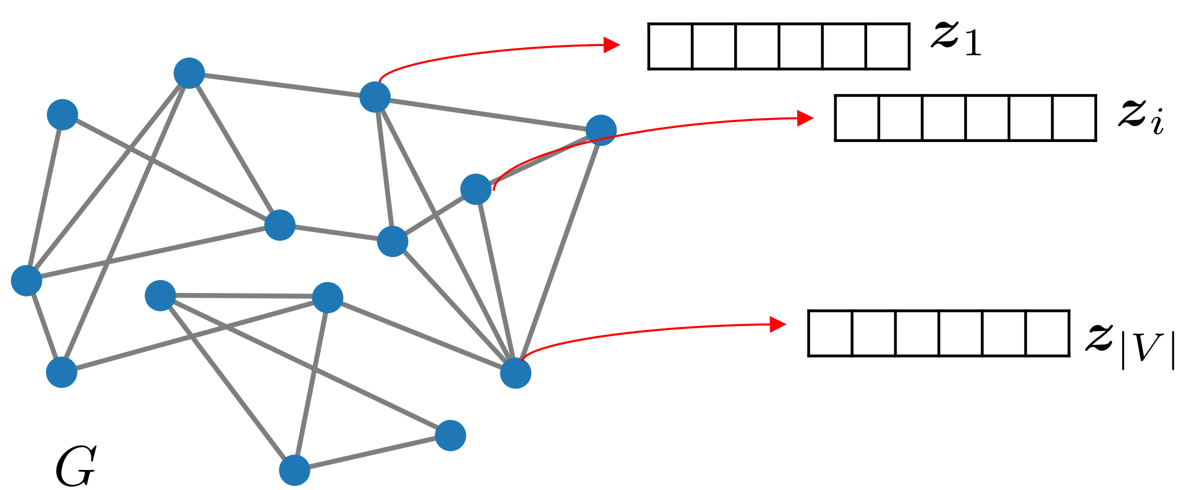
Shallow Embedding

- The node embedding methods discussed so far learn based on a shallow embedding.
- Shallow embedding methods have a few shortcomings:
  - They do not **share parameters** within nodes in the encoder.
    - This is computationally more expensive and statistically less efficient.
  - They are **transductive**.



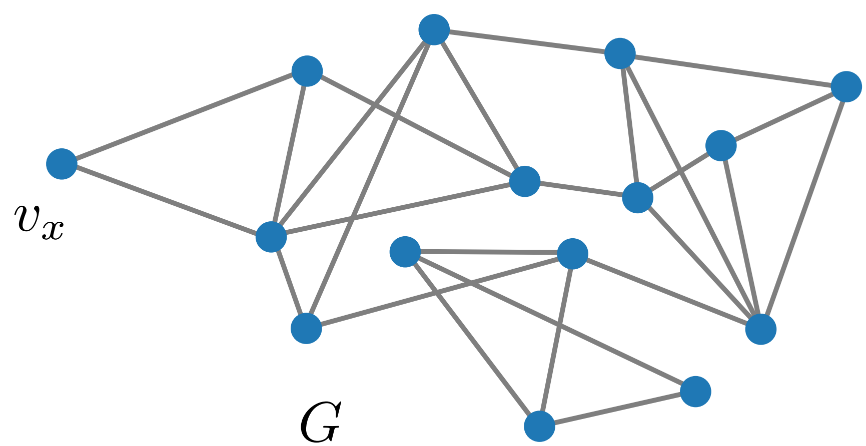
Shallow Embedding

- The node embedding methods discussed so far learn based on a shallow embedding.
- Shallow embedding methods have a few shortcomings:
  - They do not **share parameters** within nodes in the encoder.
  - This is computationally more expensive and statistically less efficient.
  - They are **transductive**.



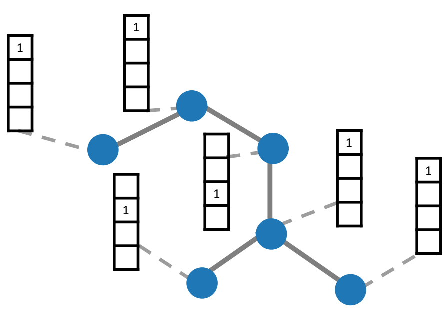
Shallow Embedding

- The node embedding methods discussed so far learn based on a shallow embedding.
- Shallow embedding methods have a few shortcomings:
  - They do not **share parameters** within nodes in the encoder.
  - This is computationally more expensive and statistically less efficient.
  - They are **transductive**.



Shallow Embedding

- The node embedding methods discussed so far learn based on a shallow embedding.
- Shallow embedding methods have a few shortcomings:
  - They do not **share parameters** within nodes in the encoder.
    - This is computationally more expensive and statistically less efficient.
  - They are **transductive**.
    - Can't learn embedding on nodes not seen during the training.
  - They don't leverage **node attributes**.



Shallow Embedding

- The node embedding methods discussed so far learn based on a shallow embedding.
- Shallow embedding methods have a few shortcomings:
  - They do not **share parameters** within nodes in the encoder.
    - This is computationally more expensive and statistically less efficient.
  - They are **transductive**.
    - Can't learn embedding on nodes not seen during the training.
  - They don't leverage **node attributes**.

- More sophisticated encoders based on **deep learning** models alleviate these limitations

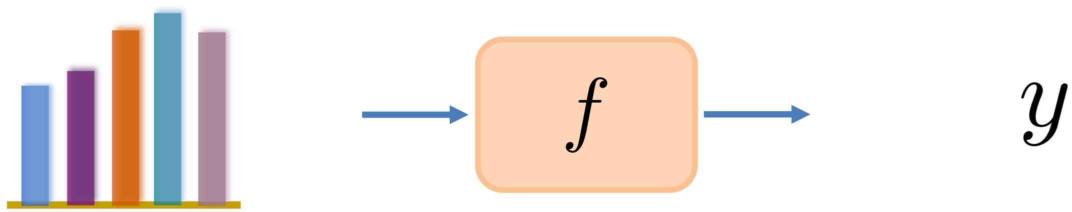
Supervised Learning

- Solving supervised ML problems involves approximating a **mapping** $f: x \rightarrow y$ between the inputs x and outputs y.

Supervised Learning

- Solving supervised ML problems involves approximating a **mapping** $f: x \rightarrow y$ between the inputs x and outputs y.

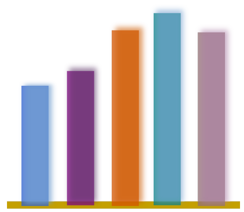
- Regression

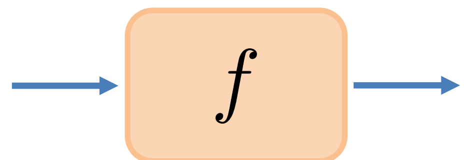


Supervised Learning

- Solving supervised ML problems involves approximating a **mapping** $f: x \rightarrow y$ between the inputs x and outputs y.

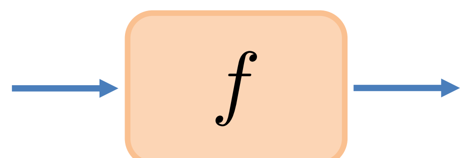
- Regression

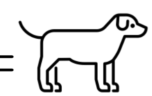




y

- Classification



$P(X =$  $)$

$P(X =$  $)$

Supervised Learning

- **Linear** models map data $\boldsymbol{x}$ to the output $\boldsymbol{y}$ using a function of the form

$$\mathbf{y} = f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{W}\mathbf{x} + b$$

  where $\boldsymbol{\theta} = \{\boldsymbol{W}, b\}$.

- The linearity assumption in these models is **restricting**.

Supervised Learning

- **Linear** models map data $\boldsymbol{x}$ to the output $\boldsymbol{y}$ using a function of the form

$$\mathbf{y} = f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{W}\mathbf{x} + b$$

  where $\boldsymbol{\theta} = \{\boldsymbol{W}, b\}$.

- The linearity assumption in these models is **restricting**.

- One way to alleviate this is by instead mapping some feature **transformation** $\phi(\boldsymbol{x})$ of input $\boldsymbol{x}$

$$\mathbf{y} = \boldsymbol{W}\phi(\mathbf{x}) + b$$

- The model is constructed of linear combination of fixed basis functions.

- $\boldsymbol{y}$ is now an **expansion** in the basis function ϕ.

Supervised Learning

- These models are easy to optimize as the model is **linear** in the **parameter** space.
- **Hand designing** ϕ poses **limits** on this approach and makes it less efficient.
- Also, when the **dimension** of the data increases, the applicability of these models gets limited.

Neural Networks

- These models are easy to optimize as the model is **linear** in the **parameter** space.
  - **Hand designing** ϕ poses **limits** on this approach and makes it less efficient.
  - Also, when the **dimension** of the data increases, the applicability of these models gets limited.
- One remedy to this problem is using basis functions that are **adapted** to the data.
- In other words, we extend the previous model and instead of defining ϕ, **learn** ϕ.

Neural Networks

- This can be done by **parameterizing** ϕ as

$$\mathbf{y} = \boldsymbol{W}\phi_{\boldsymbol{\theta}_2}(\mathbf{x}) + b$$

  where $\boldsymbol{\theta}_2$ is some parameter for basis function ϕ.

- The learned parameters include $\boldsymbol{\theta}_1 = \{\boldsymbol{W}, b\}$ and $\boldsymbol{\theta}_2$.

Neural Networks

- This can be done by **parameterizing** ϕ as

$$\mathbf{y} = \boldsymbol{W}\phi_{\boldsymbol{\theta}_2}(\mathbf{x}) + b$$

  where $\boldsymbol{\theta}_2$ is some parameter for basis function ϕ.

- The learned parameters include $\boldsymbol{\theta}_1 = \{\boldsymbol{W}, b\}$ and $\boldsymbol{\theta}_2$.
- **Neural networks** are one such approach.
- A **recursive** application of this parameterized basis function

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = f_L(f_{L-1}(\dots f_1(\mathbf{x})\dots))$$

  can learn more **complex** functions.

- This is the main idea behind **deep neural networks**.

Neural Networks

- A fully-connected neural network f is a non-linear function parametrized by $\boldsymbol{W}$, that maps a set of input variables $\boldsymbol{x}$ to output variables $\boldsymbol{y}$.

$$\mathbf{y} = f_{\boldsymbol{\theta}}(\mathbf{x})$$

and consists of a cascade of transformations

$$\mathbf{y}_{\ell} = \sigma(\boldsymbol{W}_{\ell-1,\ell}\mathbf{y}_{\ell-1} + b_{\ell-1})$$

with $\boldsymbol{\theta} = \{\boldsymbol{W}_{\ell-1,\ell}, b_{\ell-1}\}$ for $0 < \ell \leq \mathrm{L}$ and $\boldsymbol{y}_0 = \boldsymbol{x}$, and σ is non-linearity.

Neural Networks

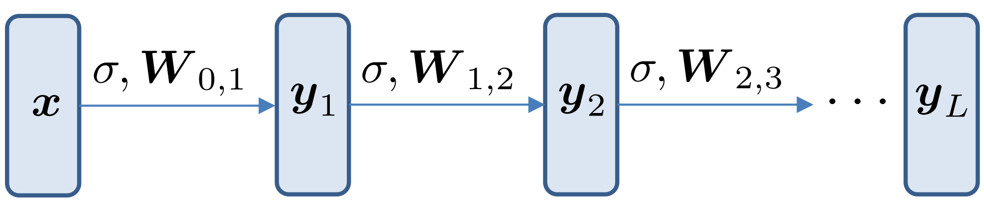
- A fully-connected neural network f is a non-linear function parametrized by $\boldsymbol{W}$, that maps a set of input variables $\boldsymbol{x}$ to output variables $\boldsymbol{y}$.

$$\mathbf{y} = f_{\boldsymbol{\theta}}(\mathbf{x})$$

and consists of a cascade of transformations

$$\mathbf{y}_\ell = \sigma(\boldsymbol{W}_{\ell-1,\ell}\mathbf{y}_{\ell-1} + b_{\ell-1})$$

with $\boldsymbol{\theta} = \{\boldsymbol{W}_{\ell-1,\ell}, b_{\ell-1}\}$ for $0 < \ell \leq \mathrm{L}$ and $\boldsymbol{y}_0 = \boldsymbol{x}$, and σ is non-linearity.



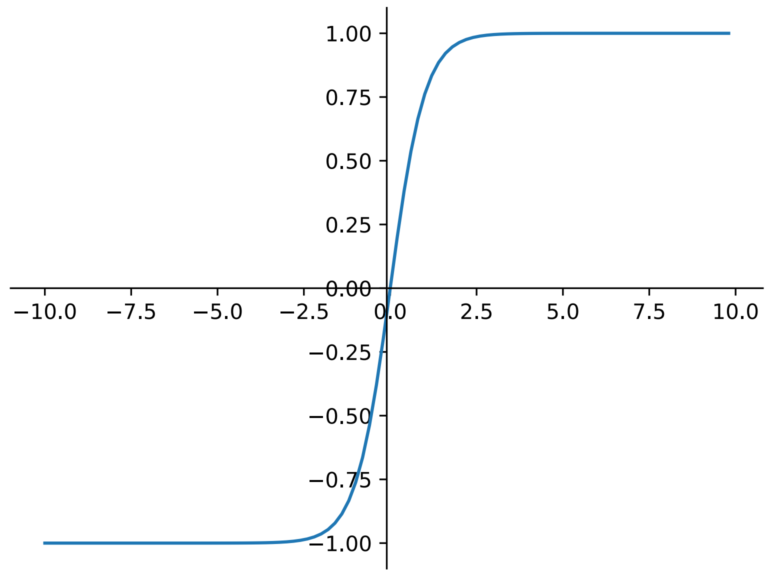
- Each intermediate output $\boldsymbol{y}_\ell$ is called a hidden layer.

Activation Function

- There are a number of non-linear activation functions that are used.

- Tanh

$$\sigma(x) = \frac{2}{1+\exp(-2x)} - 1$$

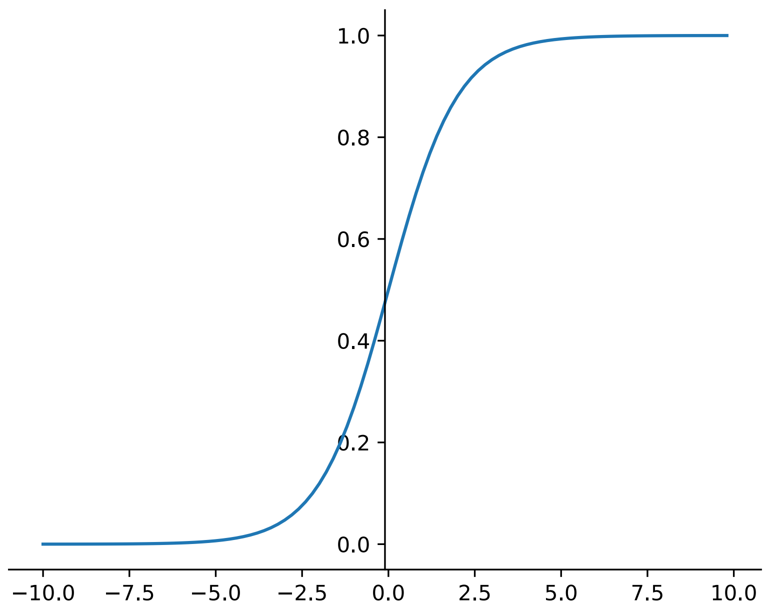


Activation Function

- There are a number of non-linear activation functions that are used.

  - Sigmoid

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

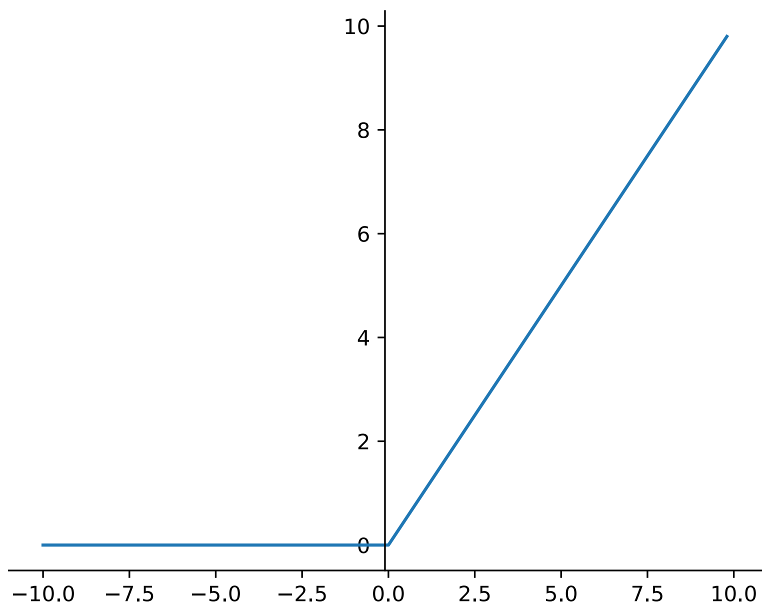


Activation Function

- There are a number of non-linear activation functions that are used.

- ReLU

$$\sigma(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

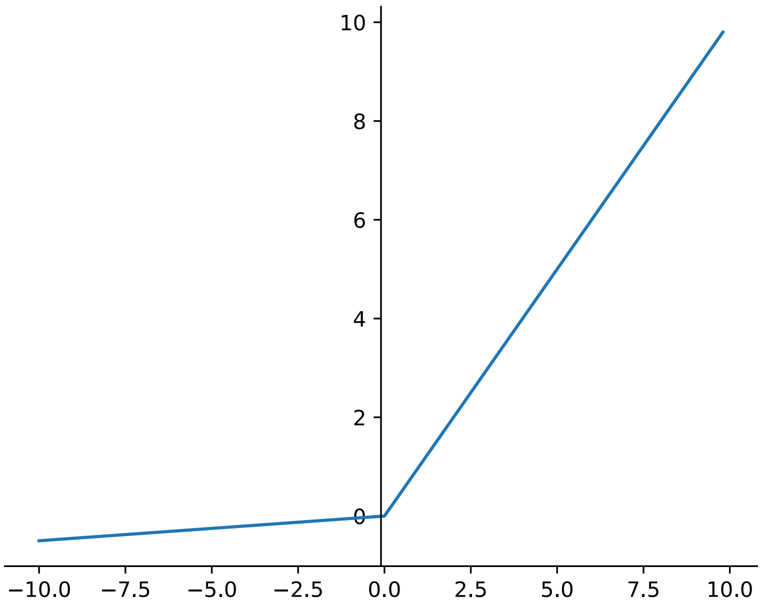


Activation Function

- There are a number of non-linear activation functions that are used.

- LeakyReLU

$$\sigma(x) = \begin{cases} x & \text{if } x \geq 0 \\ \beta x & \text{if } x < 0 \end{cases}$$

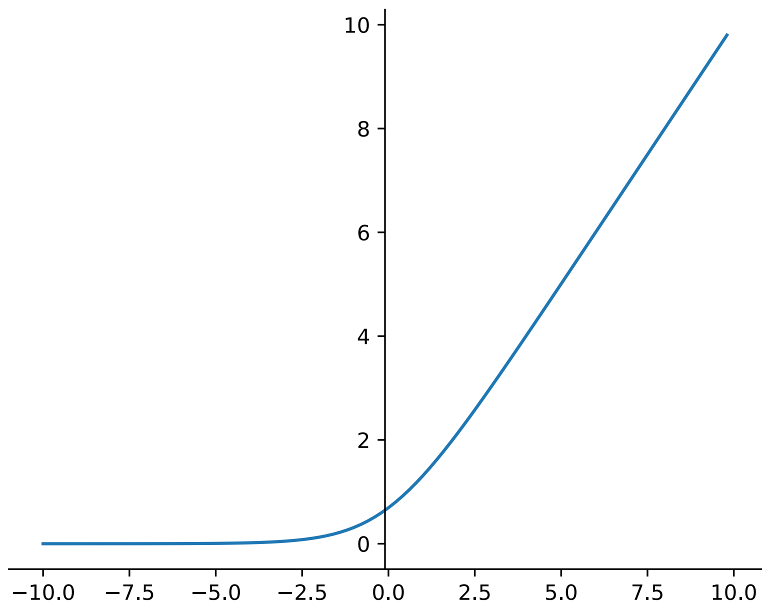


Activation Function

- There are a number of non-linear activation functions that are used.

- SoftPlus

$$\sigma(x) = \frac{1}{\beta}\log(1 + \exp(\beta x))$$



Training

- Constructing a model requires
  - An **architecture**.
  - An **objective** function.
  - An **optimization** procedure.

Training

- Constructing a model requires
  - An **architecture**.
  - An **objective** function.
  - An **optimization** procedure.

- In a **data-driven** approach, an objective function based on the training data $\mathcal{D}$ is optimized to yield the model parameters.

- This is referred to as **training** or **model fitting**.

- Assuming that training data is sampled from a **true** distribution p_{data}, we define the model p_θ and find parameters that give high probability to the observed data.

Training

- In most occasions, the model represents a **parametric** conditional probability **distribution**,

$$p_{\boldsymbol{\theta}}(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$$

- Assuming data points are **independently** sampled from p_{data}, we rewrite likelihood as

$$p_{\boldsymbol{\theta}}(\mathcal{D}) = \prod_{i=1}^{N} p_{\boldsymbol{\theta}}(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$$

- This is assuming training data is Independent and identically distributed (**iid**).

- Given training data $\mathcal{D} = \{x^{(i)}, y^{(i)}\}_{i \leq N}$, the objective is to **maximize** the probability of the **observed** data $\mathcal{D}$.

Training

- In other words

$$\hat{\boldsymbol{\theta}} = \arg\ \max_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\mathcal{D})$$

where

$$p_{\boldsymbol{\theta}}(\mathcal{D}) = \prod_{i=1}^{N} p_{\boldsymbol{\theta}}(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$$

- Alternatively, we use negative log-likelihood for numerical stability

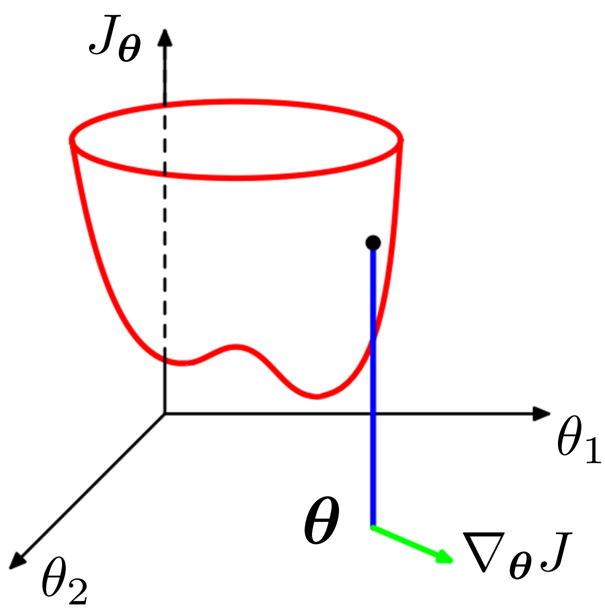
$$\hat{\boldsymbol{\theta}}_{\mathrm{MLE}} = \arg\min_{\boldsymbol{\theta}} -\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}\left(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}\right)$$

Optimization

- Optimizing a function $J(\boldsymbol{\theta})$ refers to minimizing or maximizing $J(\boldsymbol{\theta})$ by altering parameter $\boldsymbol{\theta}$.
- A **gradient-based** optimization method uses gradient of the objective function to **guide** the parameter space.

Optimization

- Optimizing a function $J(\boldsymbol{\theta})$ refers to minimizing or maximizing $J(\boldsymbol{\theta})$ by altering parameter $\boldsymbol{\theta}$.
- A **gradient-based** optimization method uses gradient of the objective function to **guide** the parameter space.



Optimization

- Optimizing a function $J(\boldsymbol{\theta})$ refers to minimizing or maximizing $J(\boldsymbol{\theta})$ by altering parameter $\boldsymbol{\theta}$.
- A **gradient-based** optimization method uses gradient of the objective function to **guide** the parameter space.
- The derivative of $J(x)$ returns the slope of J at point x.
- To **minimize** a function f, we move in the direction of the **negative** of the derivative.

$$\frac{dJ(x)}{dx}$$

- For **multivariate** function f, we need to compute the **partial** derivative of $f(\boldsymbol{x})$ with respect to variables x_i.

Optimization

- For a multivariate function $f(\boldsymbol{x})$: $\mathbb{R}^n \rightarrow \mathbb{R}$ and variable $\boldsymbol{x} \in \mathbb{R}^n$, we represent the vector of partial derivatives using **gradient**

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left[\frac{\partial}{\partial x_1} f(\boldsymbol{x}), \ldots, \frac{\partial}{\partial x_n} f(\boldsymbol{x})\right]$$

Stochastic Gradient Decent

- For a multivariate function $f(\boldsymbol{x})\colon \mathbb{R}^n \to \mathbb{R}$ and variable $\boldsymbol{x} \in \mathbb{R}^n$, we represent the vector of partial derivatives using **gradient**

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left[\frac{\partial}{\partial x_1} f(\boldsymbol{x}), \ldots, \frac{\partial}{\partial x_n} f(\boldsymbol{x})\right]$$

- In deep learning algorithms, the objective is usually to **minimize** a **loss** function computed based on the training **data**

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} l(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), y^{(i)})$$

- However, computing this gradient for whole dataset $\mathcal{D}$ is computationally **expensive**.

Stochastic Gradient Decent

- Most deep learning models instead rely on a method called **stochastic gradient decent** (SGD).
- This approach instead of computing the gradient for whole dataset, approximates the gradient using a **mini-batch** of data sampled uniformly from $\mathcal{D}$

$$\mathcal{L} \approx \frac{1}{N'} \sum_{i=1}^{N'} l\left(f_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}, y^{(i)}\right)\right)$$

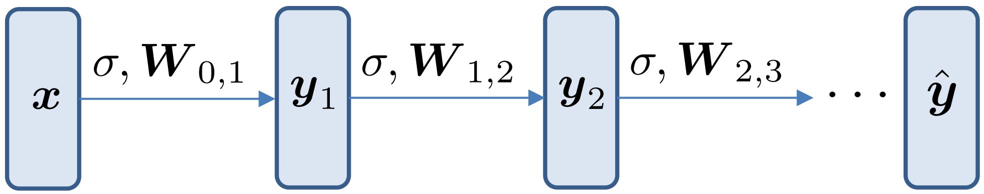
$$\nabla_{\boldsymbol{\theta}} \mathcal{L} \approx \frac{1}{N'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{N'} l(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}, y^{(i)}))$$

- The parameters are then **updated** iteratively using

$$\boldsymbol{\theta}^{n+1} = \boldsymbol{\theta}^{n} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$$

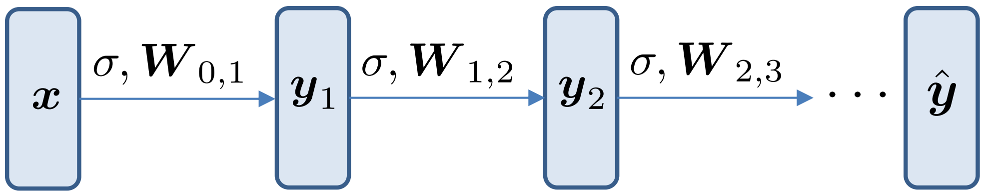
Backprop

- Given an input $\boldsymbol{x}$ to a function f, representing a feed forward network, information **propagate forward** to yield prediction $\hat{y}$.



Backprop

- Given an input $\boldsymbol{x}$ to a function f, representing a feed forward network, information **propagate forward** to yield prediction $\hat{y}$.

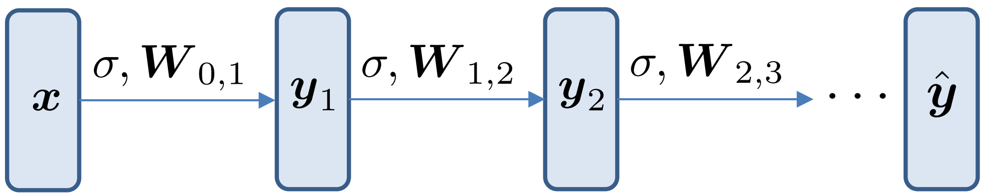


- During the training, this prediction $\hat{y}$ is plugged into the **objective** function to **evaluate** the model.

- To **update** the model parameter through a gradient-based scheme, we need to compute the **gradient of objective** with respect to each parameter.

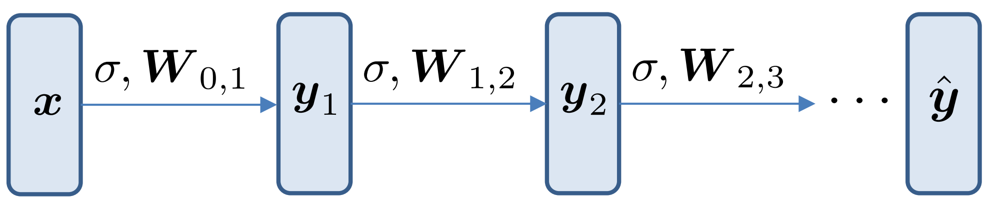
Backprop

- To compute the **derivative** of **objective** function, we use an algorithm called error backpropagation, or in short **backprop**.
- In this algorithm, derivatives of the loss function are **propagated backward** through the network to compute the gradient of objective with respect to the weight parameters in the earlier layers.



Backprop

- To compute the **derivative** of **objective** function, we use an algorithm called error backpropagation, or in short **backprop**.
- In this algorithm, derivatives of the loss function are **propagated backward** through the network to compute the gradient of objective with respect to the weight parameters in the earlier layers.



- Backprop takes advantage of **chain rule** of calculus to efficiently compute derivatives of a function.

Chain Rule

- In calculus, **chain rule** is used to **compute derivative** of a function f, which is composed of other functions and variables with known derivatives.

- Let

$$z = f(y) \qquad \text{and} \qquad y = g(x)$$

  where $f: \mathbb{R} \rightarrow \mathbb{R}$ and $g: \mathbb{R} \rightarrow \mathbb{R}$ are two functions.

Chain Rule

- In calculus, **chain rule** is used to **compute derivative** of a function f, which is composed of other functions and variables with known derivatives.
- Let

$$z = f(y) \qquad \text{and} \qquad y = g(x)$$

where $f\colon \mathbb{R} \to \mathbb{R}$ and $g\colon \mathbb{R} \to \mathbb{R}$ are two functions.

- We can use chain rule to compute the derivative of z with respect to x.

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

Chain Rule

- In calculus, **chain rule** is used to **compute derivative** of a function f, which is composed of other functions and variables with known derivatives.
- Let

$$z = f(y) \qquad \text{and} \qquad y = g(x)$$

where $f: \mathbb{R} \to \mathbb{R}$ and $g: \mathbb{R} \to \mathbb{R}$ are two functions.

- We can use chain rule to compute the derivative of z with respect to x.

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

- This can be extended to multivariate functions.

Chain Rule

- Let

$$z = f(\boldsymbol{y}) \qquad \text{and} \qquad \boldsymbol{y} = g(\boldsymbol{x})$$

where $f\colon \mathbb{R}^m \to \mathbb{R}$ and $g\colon \mathbb{R}^n \to \mathbb{R}^m$ are two multivariate functions.

Chain Rule

- Let

$$z = f(\boldsymbol{y}) \qquad \text{and} \qquad \boldsymbol{y} = g(\boldsymbol{x})$$

  where $f: \mathbb{R}^m \to \mathbb{R}$ and $g: \mathbb{R}^n \to \mathbb{R}^m$ are two multivariate functions.

- We can use chain rule to compute the derivatives

$$\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i}$$

Chain Rule

- Let

$$z = f(\boldsymbol{y}) \qquad \text{and} \qquad \boldsymbol{y} = g(\boldsymbol{x})$$

where $f : \mathbb{R}^m \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ are two multivariate functions.

- We can use chain rule to compute the derivatives

$$\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i}$$

- In vector notation

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

Chain Rule

- Let

$$z = f(\boldsymbol{y}) \qquad \text{and} \qquad \boldsymbol{y} = g(\boldsymbol{x})$$

  where $f: \mathbb{R}^m \rightarrow \mathbb{R}$ and $g: \mathbb{R}^n \rightarrow \mathbb{R}^m$ are two multivariate functions.

- We can use chain rule to compute the derivatives

$$\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i}$$

- In vector notation

$$\nabla_{\boldsymbol{x}} z = \underbrace{\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T}_{\text{Jacobian}} \underbrace{\nabla_{\boldsymbol{y}} z}_{\text{Gradient}}$$

Jacobian

Gradient

Computational Graph

- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.

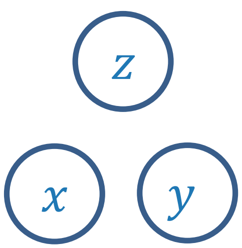
Computational Graph

- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.

$$z = xy$$

Computational Graph

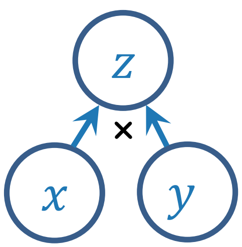
- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



$$z = xy$$

Computational Graph

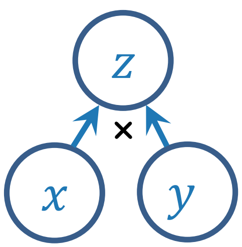
- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



$$z = xy$$

Computational Graph

- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.

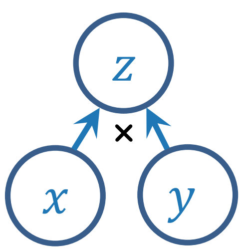


$$z = xy$$

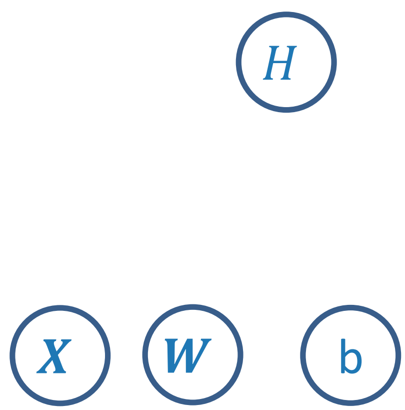
$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

Computational Graph

- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



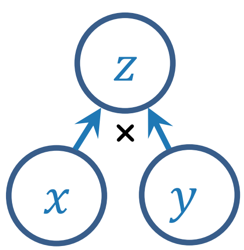
$$z = xy$$



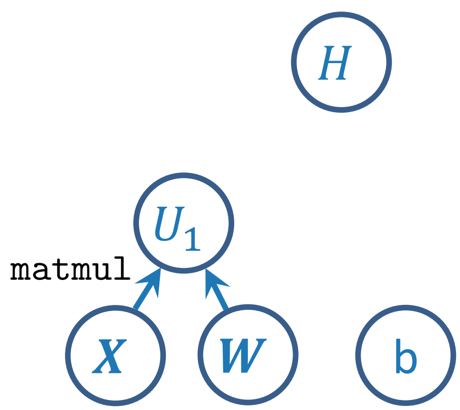
$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

Computational Graph

- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



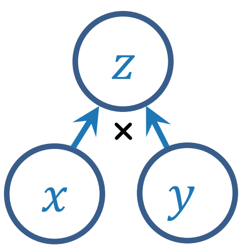
$$z = xy$$



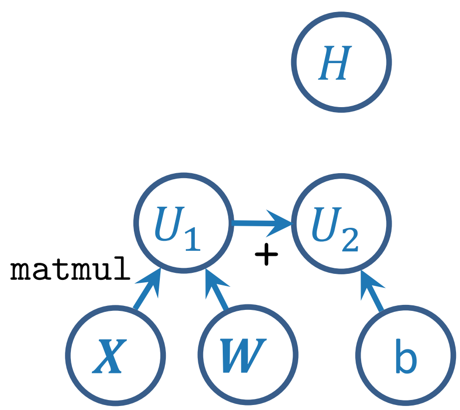
$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

Computational Graph

- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



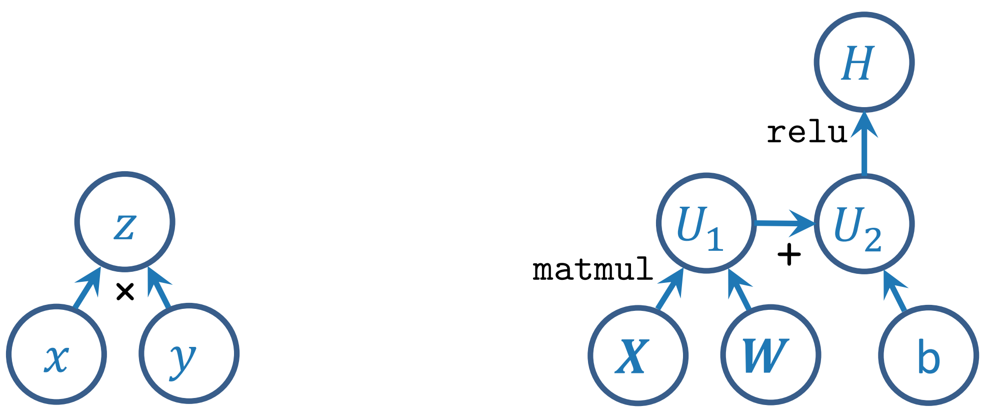
$$z = xy$$



$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

Computational Graph

- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.

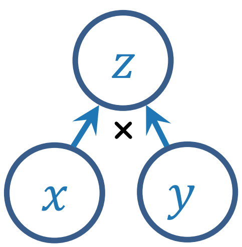


$$z = xy$$

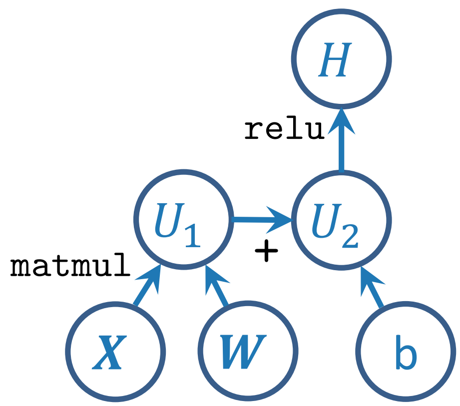
$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

Computational Graph

- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



$$z = xy$$



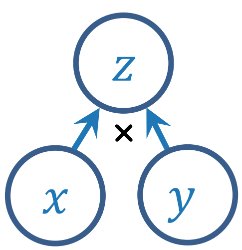
$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{w}\boldsymbol{x}$$

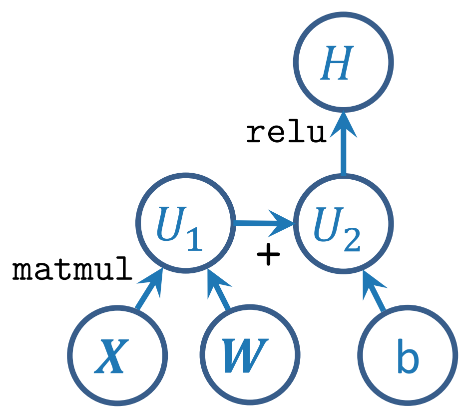
$$\lambda \sum_i w_i^2$$

Computational Graph

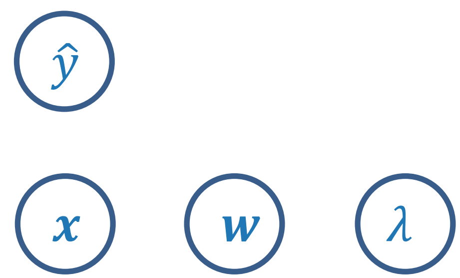
- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



$$z = xy$$



$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

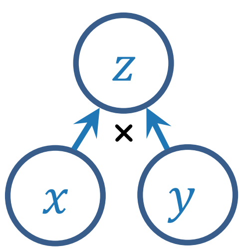


$$\hat{\boldsymbol{y}} = \boldsymbol{w}\boldsymbol{x}$$

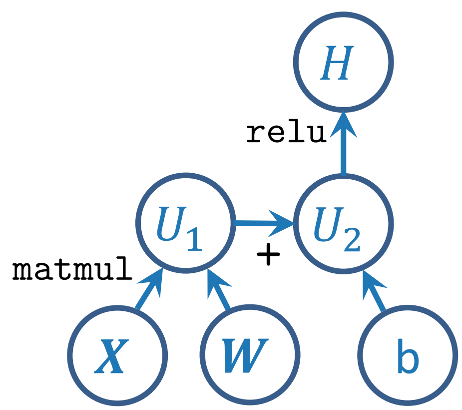
$$\lambda \sum_i w_i^2$$

Computational Graph

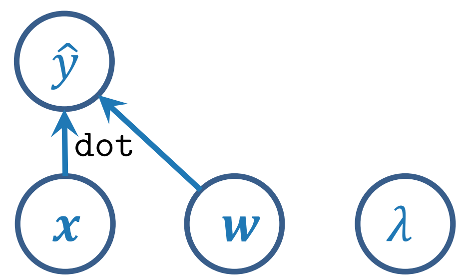
- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



$z = xy$



$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$

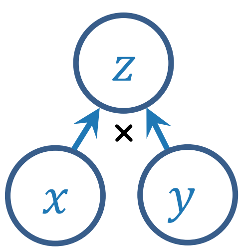


$\hat{\boldsymbol{y}} = \boldsymbol{w}\boldsymbol{x}$

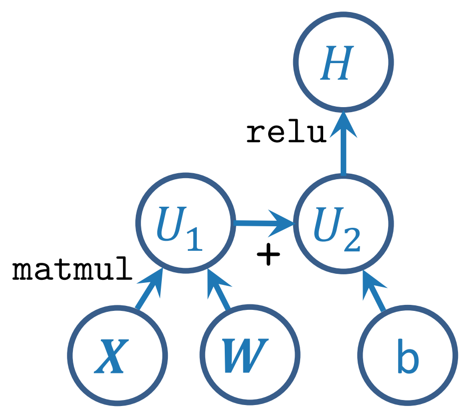
$\lambda \sum_i w_i^2$

Computational Graph

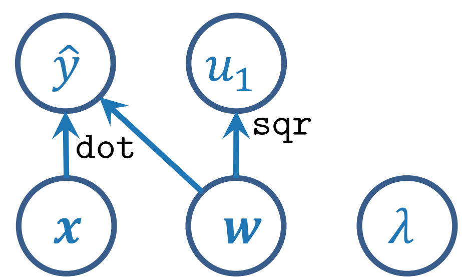
- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



$$z = xy$$



$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

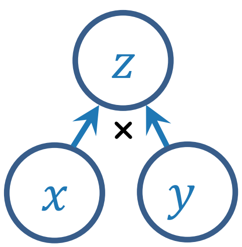


$$\hat{\boldsymbol{y}} = \boldsymbol{w}\boldsymbol{x}$$

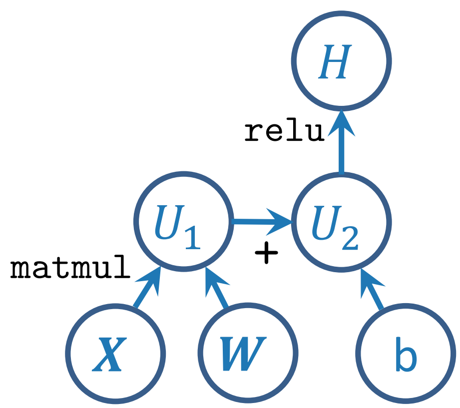
$$\lambda \sum_i w_i^2$$

Computational Graph

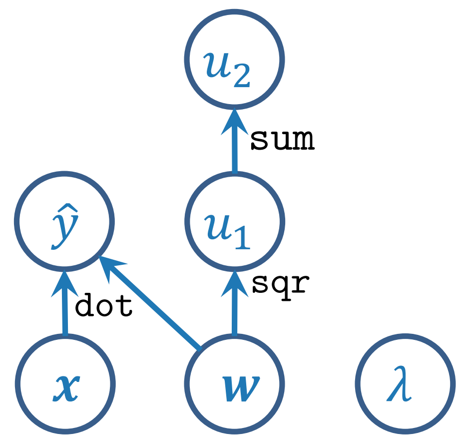
- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



$$z = xy$$



$$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$$

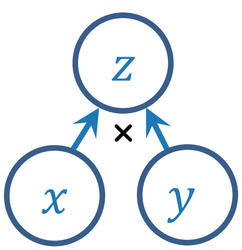


$$\hat{\boldsymbol{y}} = \boldsymbol{w}\boldsymbol{x}$$

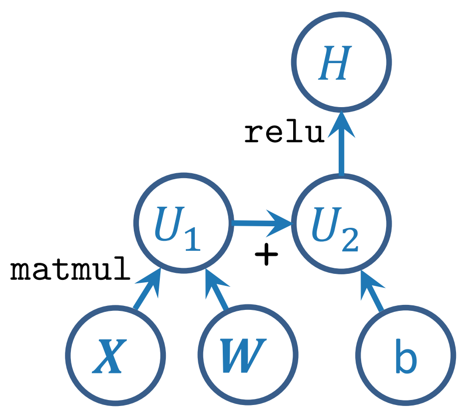
$$\lambda \sum_i w_i^2$$

Computational Graph

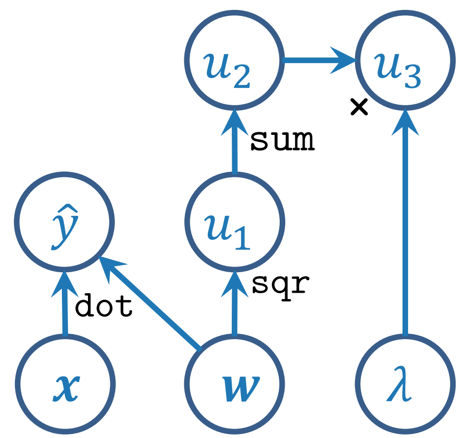
- Backprop can best be demonstrated using computational graph.
- We can use graphs to describe computations and operations
- An operation is a function that takes one or more variables as input and returns a single output.



$z = xy$



$\boldsymbol{H} = \max\{0, \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}\}$



$\hat{\boldsymbol{y}} = \boldsymbol{w}\boldsymbol{x}$

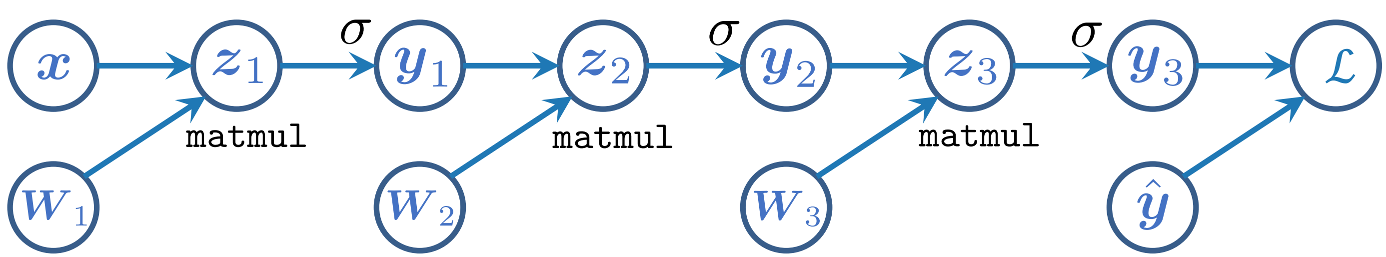
$\lambda \sum_i w_i^2$

Computational Graph

- Let's define a feedforward network using

$$\mathbf{y}_\ell = \sigma\left(\mathbf{z}_\ell\right)$$
$$\mathbf{z}_\ell = \mathbf{w}_{\ell-1,\ell}\mathbf{y}_{\ell-1}$$

- We can represent the computational graph of the network as



$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_2} =$$

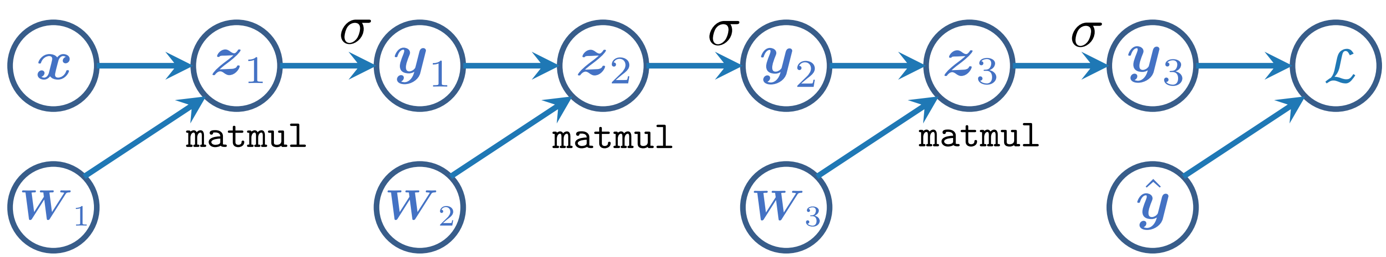
Computational Graph

- Let's define a feedforward network using

$$\mathbf{y}_\ell = \sigma(\mathbf{z}_\ell)$$

$$\mathbf{z}_\ell = \mathbf{w}_{\ell-1,\ell}\mathbf{y}_{\ell-1}$$

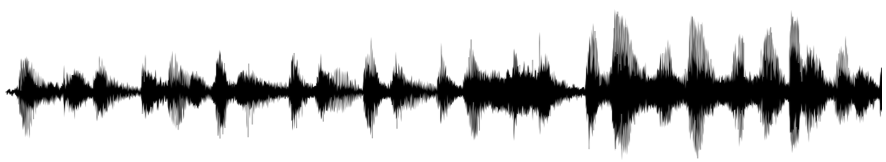
- We can represent the computational graph of the network as

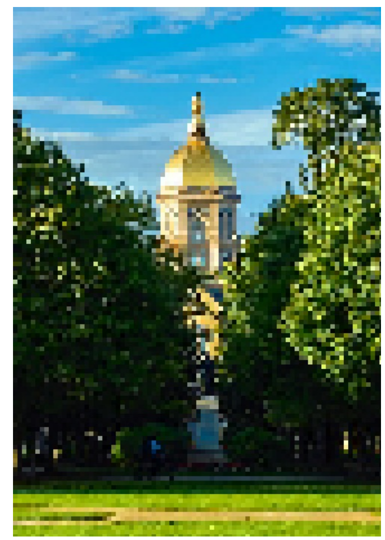


$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_3}\frac{\partial \mathbf{y}_3}{\partial \mathbf{z}_3}\frac{\partial \mathbf{z}_3}{\partial \mathbf{y}_2}\frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{w}_2}$$

Convolutional Networks

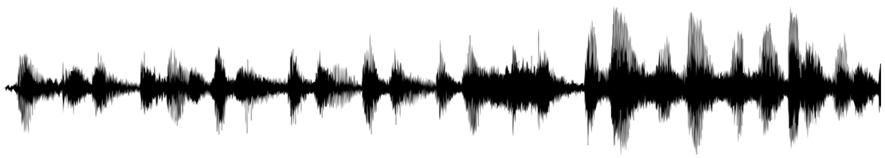
- To analyze data that resides on a **grid-like topology**, we use a group of networks named convolutional neural networks.
  - Audio, Images





Convolutional Networks

- To analyze data that resides on a **grid-like topology**, we use a group of networks named convolutional neural networks.
  - Audio, Images



- Convolutional neural networks, include layers that instead of matrix multiplication, use **convolution operations** on input data.
- The filter used in convolution operations is small, which **reduces** the number of unknown parameters in the model.

Convolutional Networks

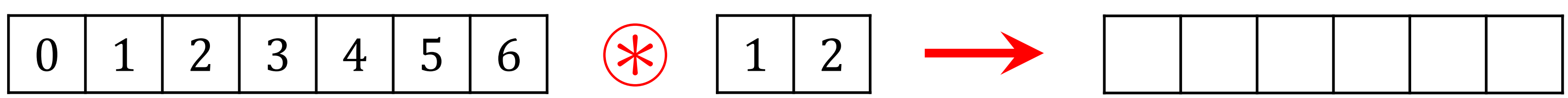
- Using finite length vectors, one can represent convolution on 1D space.
- Let $\boldsymbol{w}$ be a weight vector on domain $\{0, \dots, L-1\}$ and $\boldsymbol{x}$ be a signal on the domain $\{0, \dots, N-1\}$.
- Then, the convolution between $\boldsymbol{x}$ and $\boldsymbol{w}$ is defined as

$$[\boldsymbol{w} \circledast \boldsymbol{x}]_i = \sum_{m=0}^{L-1} w_m x_{i+m}$$

Convolutional Networks

- Using finite length vectors, one can represent convolution on 1D space.
- Let $\boldsymbol{w}$ be a weight vector on domain $\{0, \dots, L-1\}$ and $\boldsymbol{x}$ be a signal on the domain $\{0, \dots, N-1\}$.
- Then, the convolution between $\boldsymbol{x}$ and $\boldsymbol{w}$ is defined as

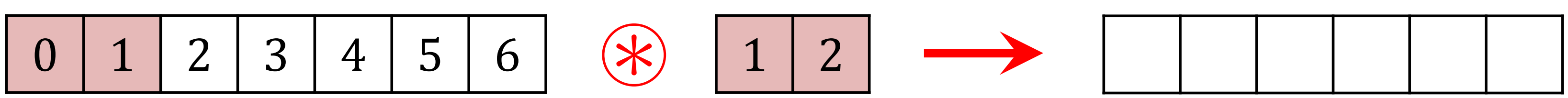
➢ $$[\boldsymbol{w} \circledast \boldsymbol{x}]_i = \sum_{m=0}^{L-1} w_m x_{i+m}$$



Convolutional Networks

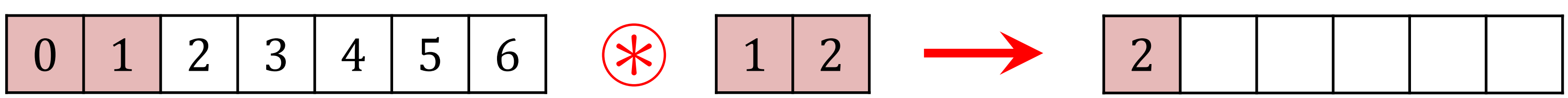
- Using finite length vectors, one can represent convolution on 1D space.
- Let $\boldsymbol{w}$ be a weight vector on domain $\{0, \dots, L-1\}$ and $\boldsymbol{x}$ be a signal on the domain $\{0, \dots, N-1\}$.
- Then, the convolution between $\boldsymbol{x}$ and $\boldsymbol{w}$ is defined as

$$[\boldsymbol{w} \circledast \boldsymbol{x}]_i = \sum_{m=0}^{L-1} w_m x_{i+m}$$



Convolutional Networks

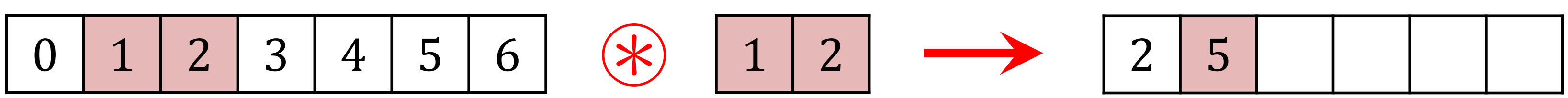
- Using finite length vectors, one can represent convolution on 1D space.
- Let $\boldsymbol{w}$ be a weight vector on domain $\{0, \ldots, L-1\}$ and $\boldsymbol{x}$ be a signal on the domain $\{0, \ldots, N-1\}$.
- Then, the convolution between $\boldsymbol{x}$ and $\boldsymbol{w}$ is defined as
  - $$[\boldsymbol{w} \circledast \boldsymbol{x}]_i = \sum_{m=0}^{L-1} w_m x_{i+m}$$



Convolutional Networks

- Using finite length vectors, one can represent convolution on 1D space.
- Let $\boldsymbol{w}$ be a weight vector on domain $\{0, \ldots, L-1\}$ and $\boldsymbol{x}$ be a signal on the domain $\{0, \ldots, N-1\}$.
- Then, the convolution between $\boldsymbol{x}$ and $\boldsymbol{w}$ is defined as

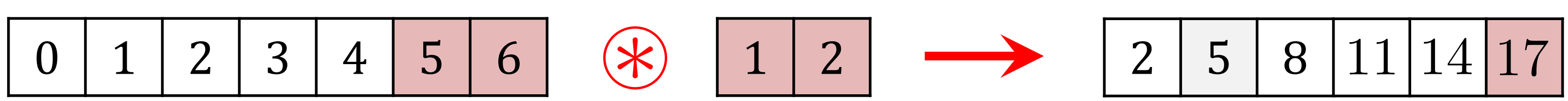
$$[\boldsymbol{w} \circledast \boldsymbol{x}]_i = \sum_{m=0}^{L-1} w_m x_{i+m}$$



Convolutional Networks

- Using finite length vectors, one can represent convolution on 1D space.
- Let $\boldsymbol{w}$ be a weight vector on domain $\{0, \dots, L-1\}$ and $\boldsymbol{x}$ be a signal on the domain $\{0, \dots, N-1\}$.
- Then, the convolution between $\boldsymbol{x}$ and $\boldsymbol{w}$ is defined as

$$[\boldsymbol{w} \circledast \boldsymbol{x}]_i = \sum_{m=0}^{L-1} w_m x_{i+m}$$



Convolutional Networks

- Similar to the 1D case, convolutions can be applied on 2D signals.
- Let $\boldsymbol{W}$ be a 2D filter with size $M \times N$ and $\boldsymbol{X}$ be a signal on the 2D domain.
- Then, the

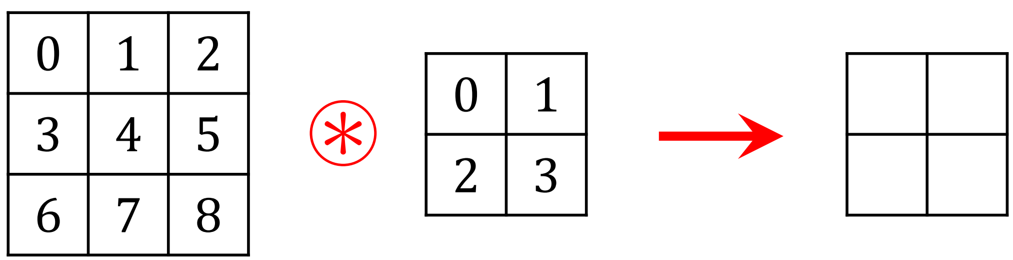
$$[\mathbf{W} \circledast \mathbf{X}]_{ij} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{mn} X_{i+m,j+n}$$

Convolutional Networks

- Similar to the 1D case, convolutions can be applied on 2D signals.
- Let $\boldsymbol{W}$ be a 2D filter with size $M \times N$ and $\boldsymbol{X}$ be a signal on the 2D domain.
- Then, the

$$[\mathbf{W} \circledast \mathbf{X}]_{ij} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{mn} X_{i+m,j+n}$$

➢

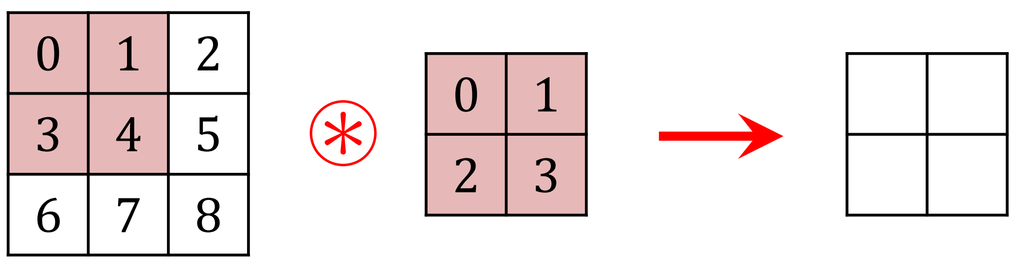


Convolutional Networks

- Similar to the 1D case, convolutions can be applied on 2D signals.
- Let $\boldsymbol{W}$ be a 2D filter with size $M \times N$ and $\boldsymbol{X}$ be a signal on the 2D domain.
- Then, the

$$[\mathbf{W} \circledast \mathbf{X}]_{ij} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{mn} X_{i+m,j+n}$$

➢

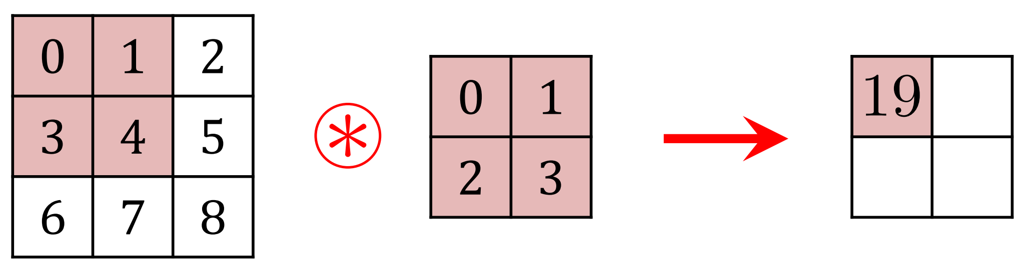


Convolutional Networks

- Similar to the 1D case, convolutions can be applied on 2D signals.
- Let $\boldsymbol{W}$ be a 2D filter with size $M \times N$ and $\boldsymbol{X}$ be a signal on the 2D domain.
- Then, the

$$[\mathbf{W} \circledast \mathbf{X}]_{ij} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{mn} X_{i+m,j+n}$$

➢

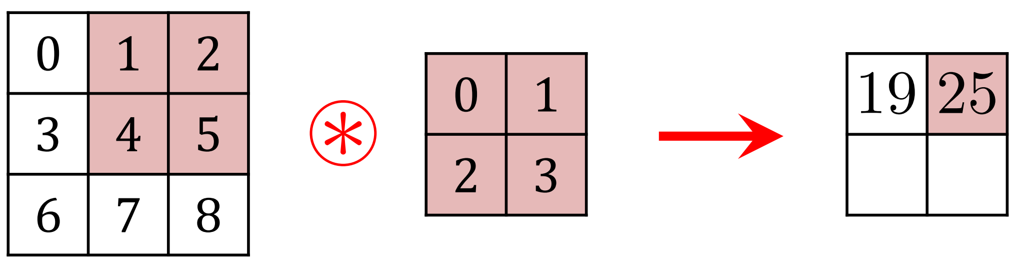


Convolutional Networks

- Similar to the 1D case, convolutions can be applied on 2D signals.
- Let $\boldsymbol{W}$ be a 2D filter with size $M \times N$ and $\boldsymbol{X}$ be a signal on the 2D domain.
- Then, the

$$[\mathbf{W} \circledast \mathbf{X}]_{ij} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{mn} X_{i+m,j+n}$$

➢

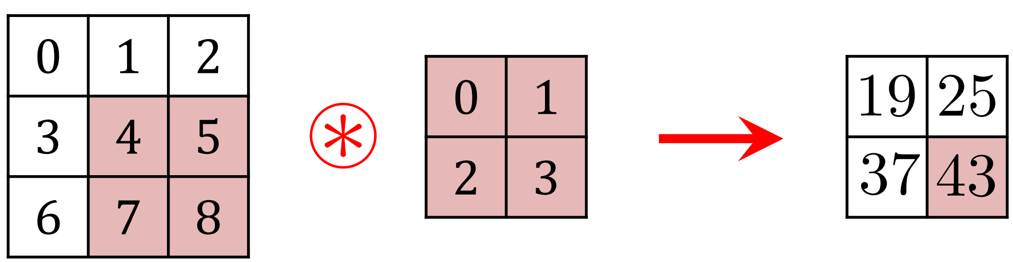


Convolutional Networks

- Similar to the 1D case, convolutions can be applied on 2D signals.
- Let $\boldsymbol{W}$ be a 2D filter with size $M \times N$ and $\boldsymbol{X}$ be a signal on the 2D domain.
- Then, the

$$[\mathbf{W} \circledast \mathbf{X}]_{ij} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{mn} X_{i+m,j+n}$$

➤



Output Unit

- An **output** activation function determines the form of the model's output.
- An output activation takes features extracted by the hidden layers and transforms it to the **desired output form**.
- The choice of the **cost function** is closely related to the choice of the output layer.
- In the **regression** problems, the output layer only includes a linear transformation with no non-linearity.

Output Unit

- A 2 class classification problem requires predicting a **binary variable**.
- In the maximum likelihood approach, we define a **Bernoulli** distribution over variable y conditioned on x.
- We only need to define $p(y = 1|x)$.

Output Unit

- A 2 class classification problem requires predicting a **binary variable**.
- In the maximum likelihood approach, we define a **Bernoulli** distribution over variable y conditioned on x.
- We only need to define $p(y = 1|x)$.
- Consider using

$$\max\{0, \min\{1, w^T h + b\}\}$$

Output Unit

- A 2 class classification problem requires predicting a **binary variable**.
- In the maximum likelihood approach, we define a **Bernoulli** distribution over variable y conditioned on x.
- We only need to define $p(y = 1|x)$.
- Consider using

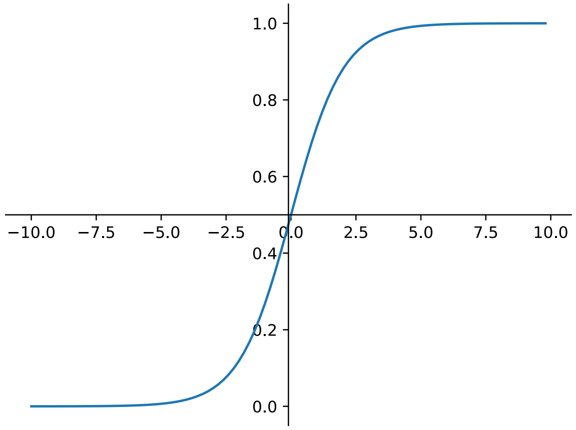
$$\max\{0, \min\{1, w^T h + b\}\}$$

- While between 0 and 1, it does have zero gradients outside $[0, 1]$.
- This makes it **impractical** as use with gradient-based optimization methods.

Output Unit

- Another approach is to use sigmoid.
- A sigmoid function is defined as

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$



- Given the extracted features from the layer before last

$$\mathbf{z}_L = \mathbf{W}_{L-1,L}\mathbf{y}_{L-1}$$

as input, the sigmoid output unit is defined as

$$\mathbf{y}_L = \text{sigmoid}(\mathbf{W}_{L-1,L}\mathbf{y}_{L-1})$$

Output Unit

- When computing the probability districbution on a discrete variable y with n possible values, we can use softmax function.
- A **softmax** function takes a vector of real values as input and returns a vector of probability values,

$$\text{softmax}(\boldsymbol{x}) := \left[\frac{\exp(x_1)}{\sum_{c=1}^{C}\exp(x_c)}, \ldots, \frac{\exp(x_C)}{\sum_{c=1}^{C}\exp(x_c)}\right]$$

where $\text{softmax}: \mathbb{R}^C \rightarrow [0,1]^C$ with C total number of possible outcomes.

- Given the extracted features from the layer before last

$$\mathbf{z}_L = \mathbf{W}_{L-1,L}\mathbf{y}_{L-1}$$

as input, the sigmoid output unit is defined as

$$\mathbf{y}_L = \text{softmax}(\mathbf{W}_{L-1,L}\mathbf{y}_{L-1})$$

Summary

- Shallow embedding
- Neural Networks
- Training
- Optimization
- Stochastic Gradient Descent
- Backprop
- Chain rule of calculus
- Computational graph
- Convolutional networks
- Output unit