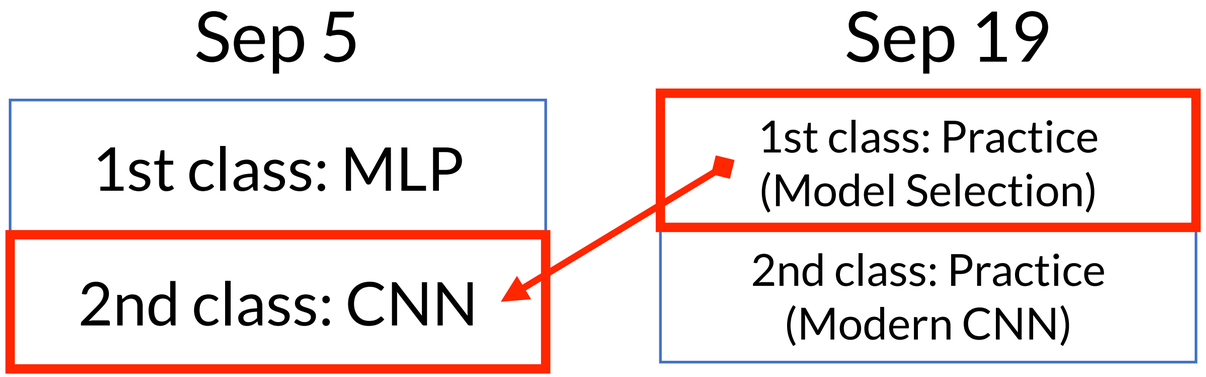
# Announcement

- We interchange today's Practice class with Theory class of Sep 19
  - there is no assignment today
  - two assignments will be given at Sep 19
  - submit your GitHub ID ASAP
  - bring your laptop for Practice class
- Recorded video will be shared for two weeks
- Check blackboard announcement weekly




the first assignment is easy
hence don't worry

# Multilayer Perceptrons

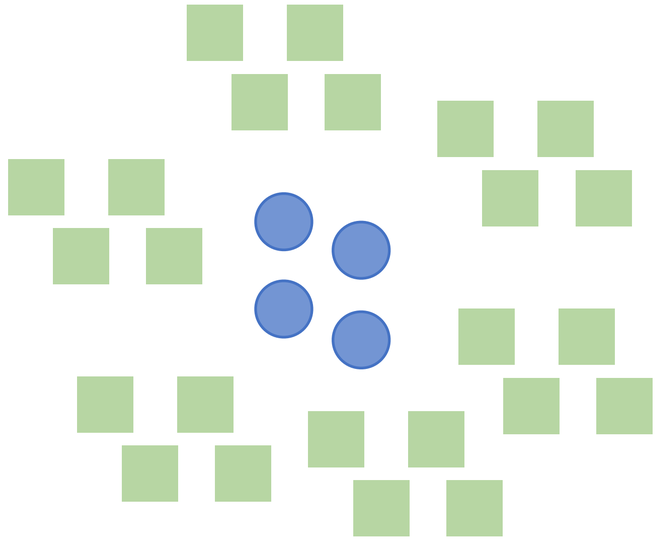
Principles of Deep Learning (AI502/IE408/IE511)

Sungbin Lim (UNIST AIGS & IE)


Contact: ai502deeplearning@gmail.com

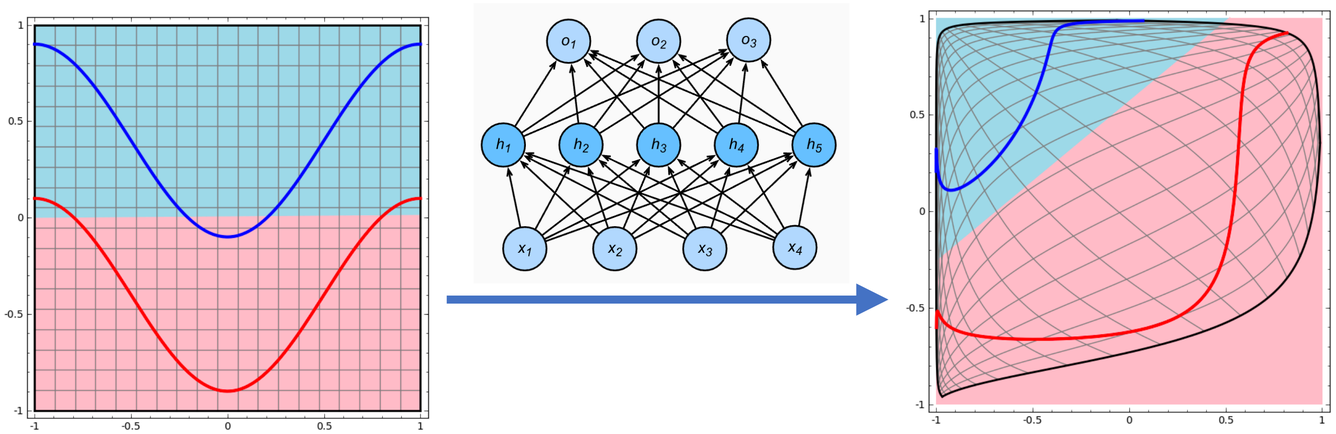
# (Review) Limitation of Linear Models

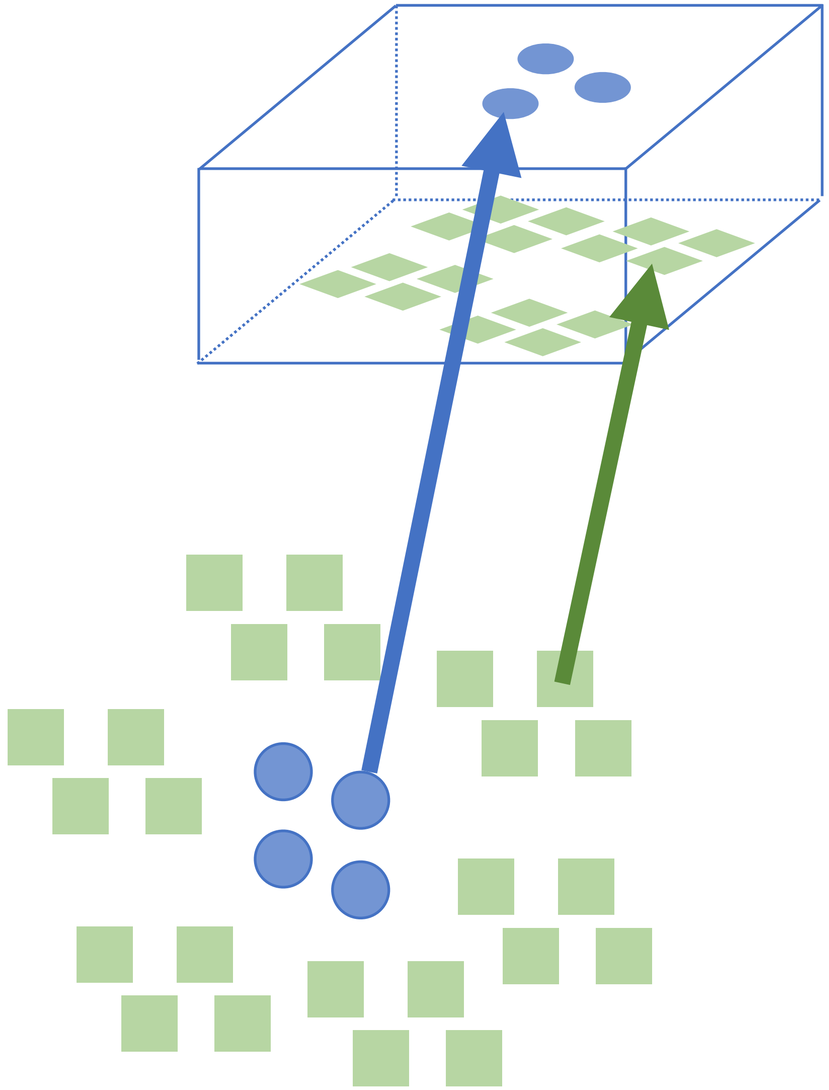
- If the decision boundary is nonlinear, then we have to find a continuous mapping which transforms the features from non-separable space to a separable one.
  - but linear function cannot do this!
- We find a manifold in some high-dimensional spaces by learning a smooth representation
  - this is called manifold learning



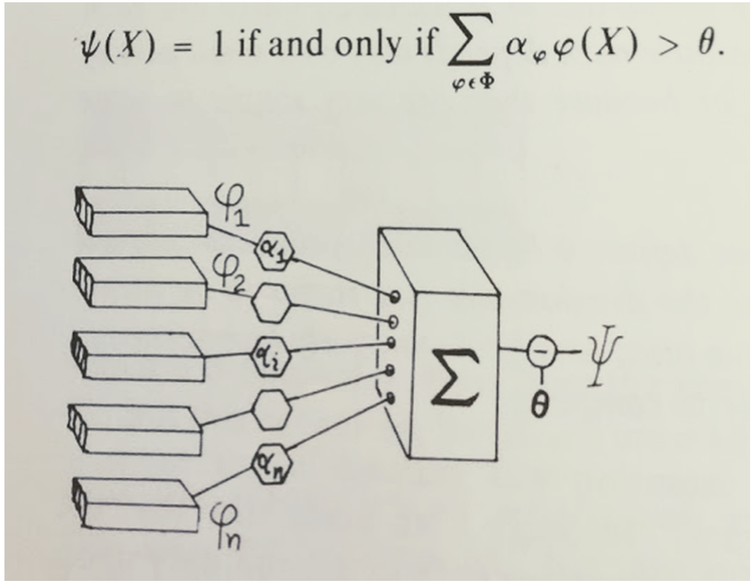
# (Review) Manifold Learning

- We find a manifold in some high-dimensional spaces by learning a smooth representation
  - this is called manifold learning
  - first we will use **multilayer perceptrons**



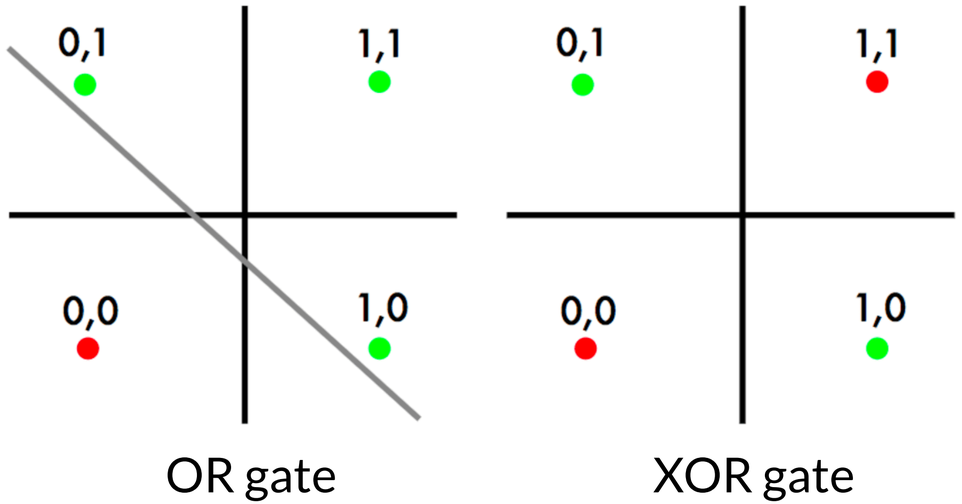



# Perceptron




- History
  - perceptron is proposed by F. Rosenblatt
- XOR problem

| | OR | XOR |
|---|---|---|
| (True, True) | 1 | 0 |
| (True, False) | 1 | 1 |
| (False, True) | 1 | 1 |
| (False, False) | 0 | 0 |



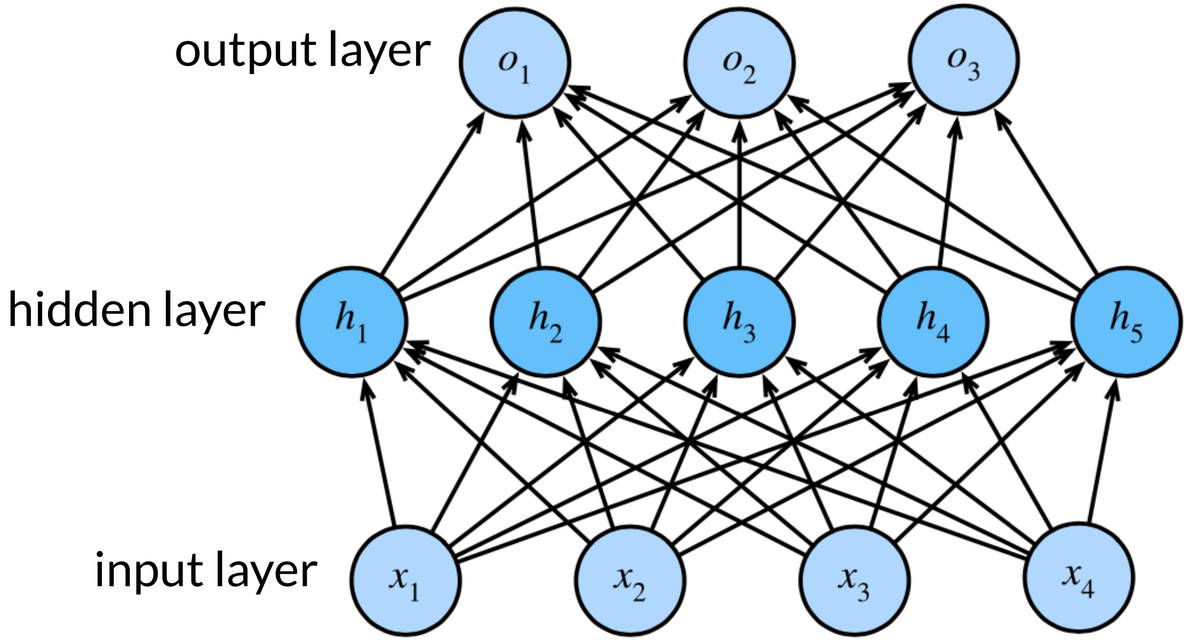

How can we find separating hyperplane for XOR problem?

# From Linear to Nonlinear

- We can solve the XOR problem by employing additional layers and nonlinear activation function
  - without nonlinear activation function, it does not work! (why?)
- Linear form

$$h_j = \sum_i w_{ij}^{(1)} x_i = \mathbf{w}_j^{(1)} \mathbf{x}$$

$$o_k = \sum_j w_{jk}^{(2)} h_j = \mathbf{w}_k^{(2)} \mathbf{h}$$

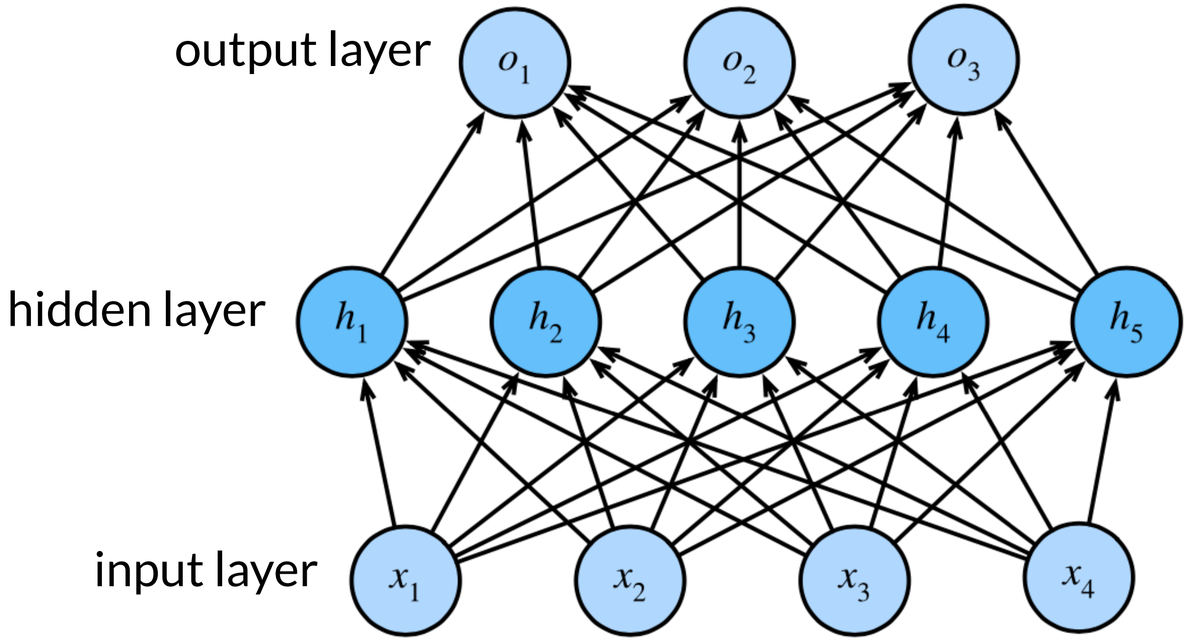


# From Linear to Nonlinear

- We can solve the XOR problem by employing additional layers and nonlinear activation function
  - without nonlinear activation function, it does not work! (why?)
- **Non**linear form

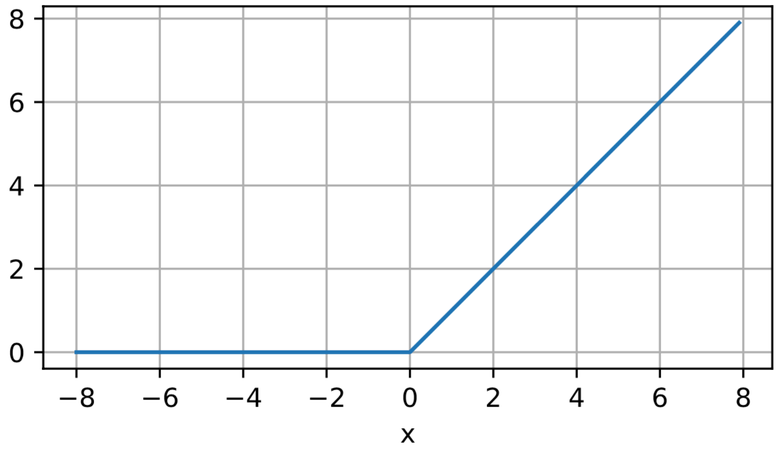
$$h_j = \varphi\left(\sum_i w_{ij}^{(1)} x_i\right) = \varphi(\mathbf{w}_j^{(1)}\mathbf{x})$$

$$o_k = \sum_j w_{jk}^{(2)} h_j = \mathbf{w}_k^{(2)}\mathbf{h}$$



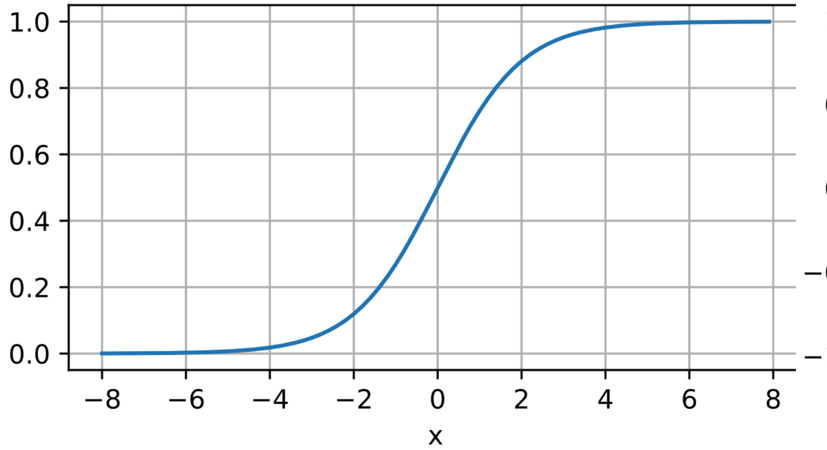
# Activation Functions

ReLU



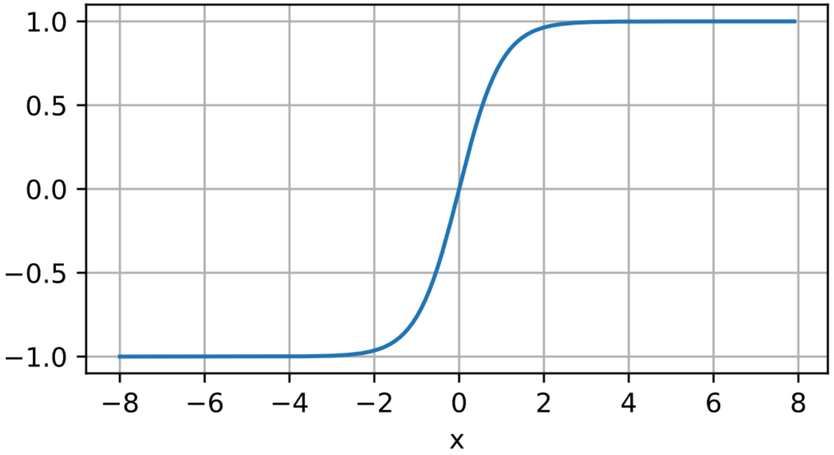

$$\mathrm{ReLU}(x) = \max(x, 0)$$

Sigmoid




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

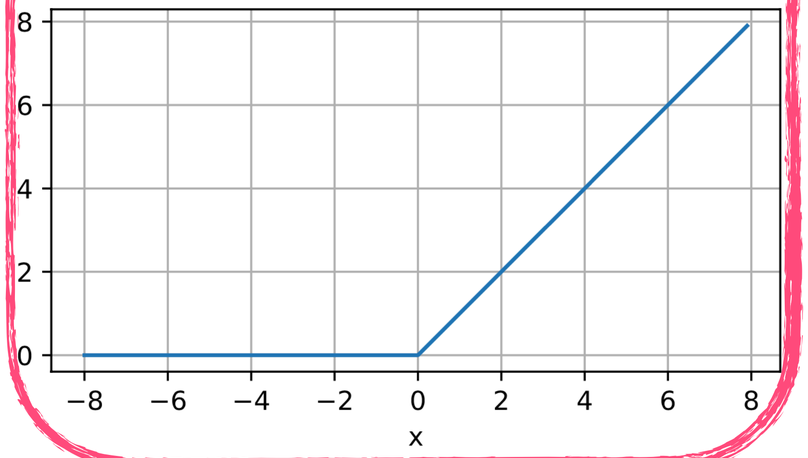
Hyperbolic Tangent



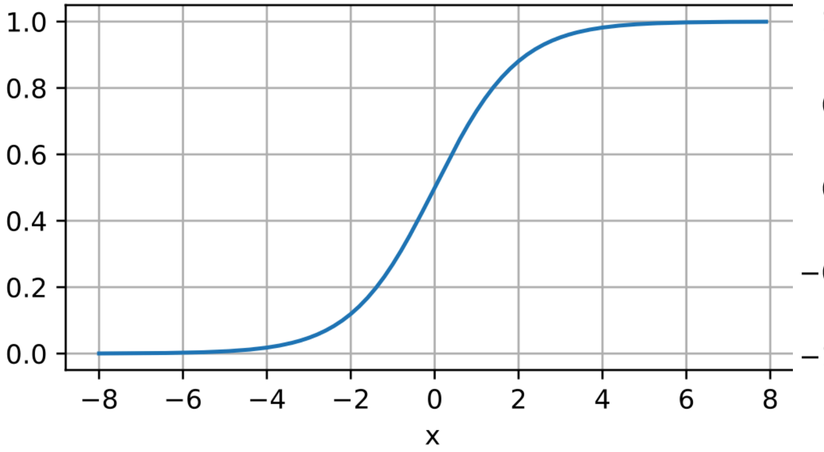

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

# Activation Functions

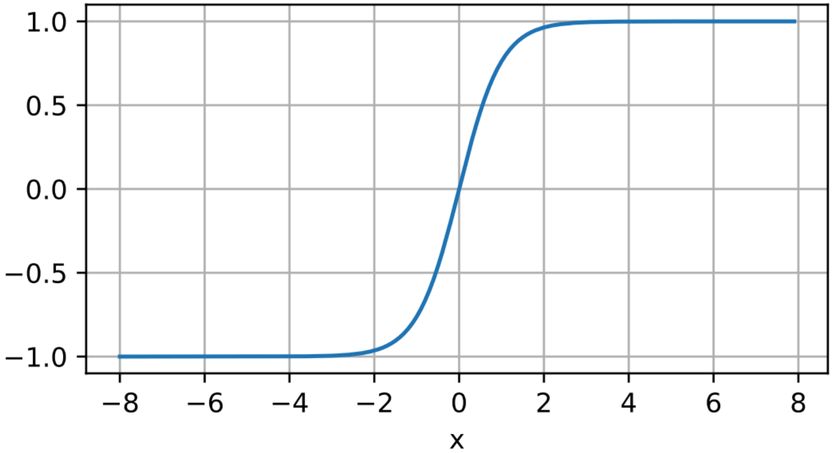
ReLU




Sigmoid



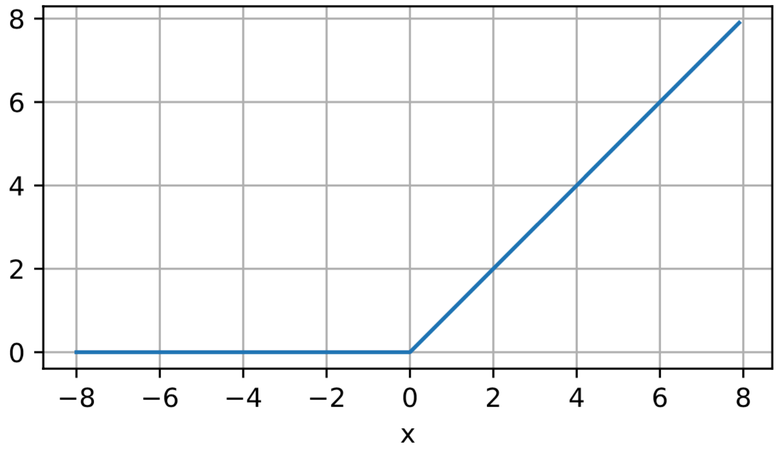

Hyperbolic Tangent



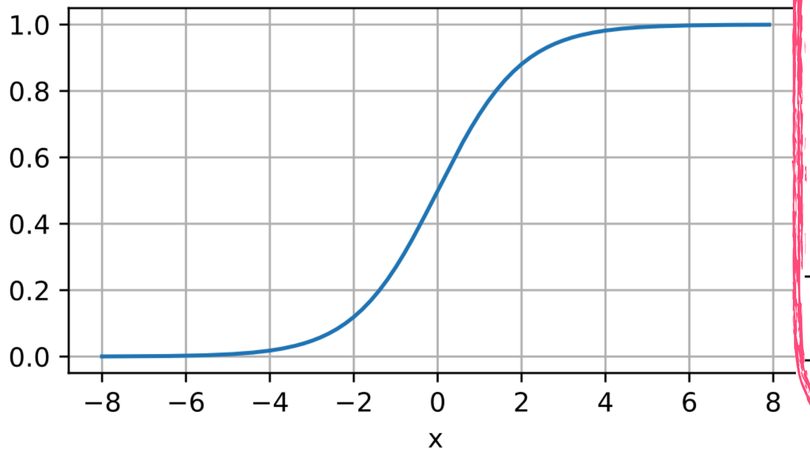

$$\mathrm{ReLU}(x) = \max(x, 0) \qquad \mathrm{Sigmoid}(x) = \frac{1}{1 + \exp(-x)} \qquad \tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

# Activation Functions

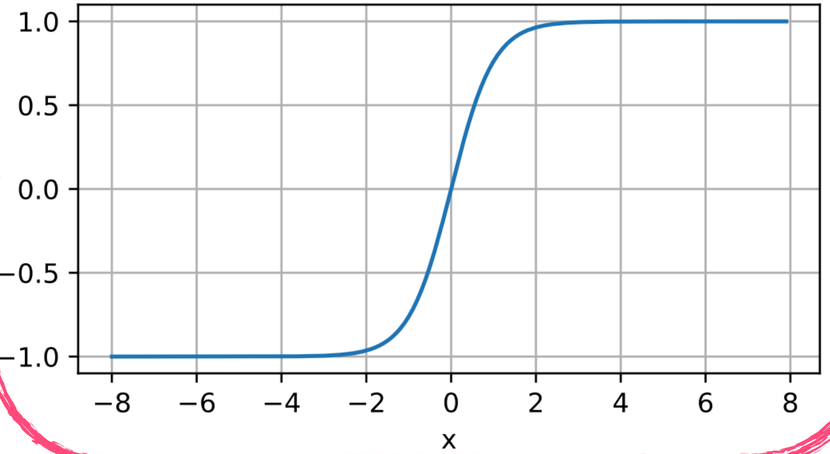
ReLU




Sigmoid



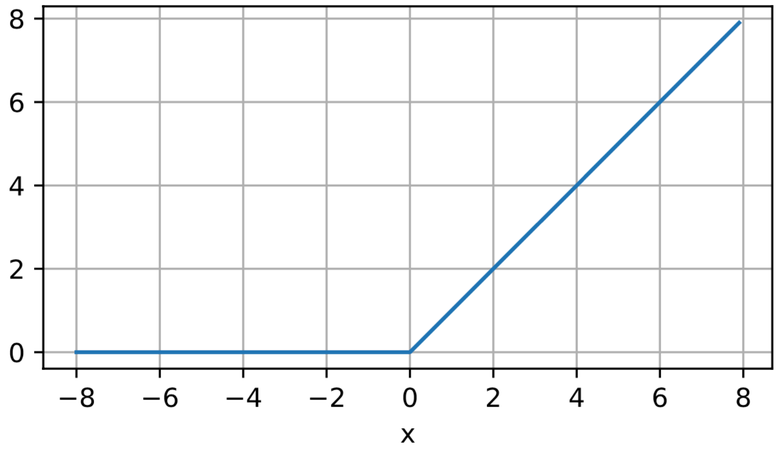

Hyperbolic Tangent



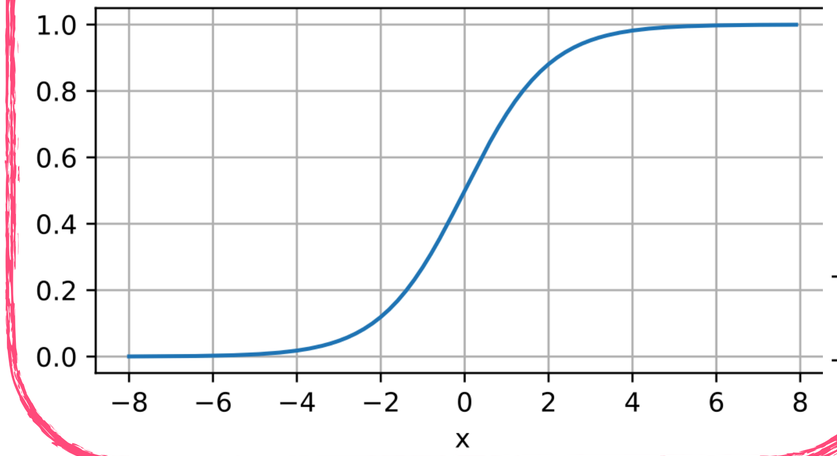

$$\mathrm{ReLU}(x) = \max(x, 0) \qquad \mathrm{Sigmoid}(x) = \frac{1}{1 + \exp(-x)} \qquad \tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

# Activation Functions

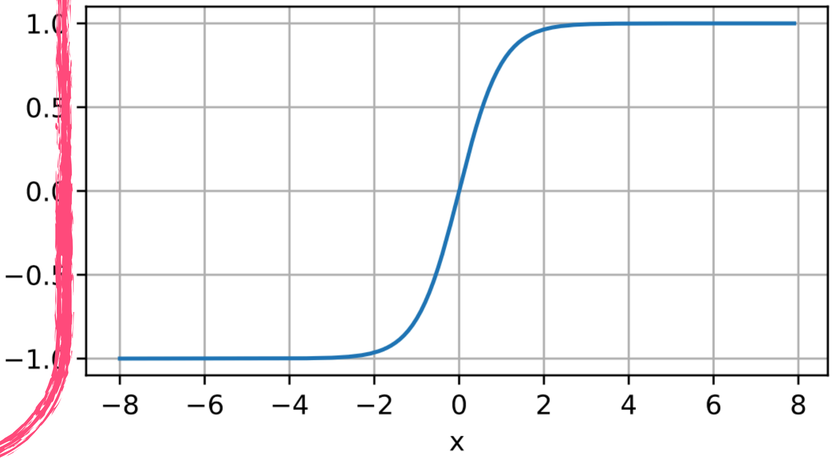
ReLU




Sigmoid




Hyperbolic Tangent




$$\mathrm{ReLU}(x) = \max(x, 0)$$

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

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

# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$

# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$

```
In[1]:= f[x_, y_, z_] := x Sqrt[Exp[-Sin[z]/y^2]]
        |제곱근 |지수 함수
```

symbolic calculus is good but slow

```
In[5]:= {D[f[x, y, z], x], D[f[x, y, z], y], D[f[x, y, z], z]}
        |미분 계수          |미분 계수          |미분 계수
```

$$\text{Out[5]=} \left\{\sqrt{e^{-\frac{\sin[z]}{y^2}}},\ \frac{\sqrt{e^{-\frac{\sin[z]}{y^2}}}\ x \sin[z]}{y^3},\ -\frac{\sqrt{e^{-\frac{\sin[z]}{y^2}}}\ x \cos[z]}{2\,y^2}\right\}$$

# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$

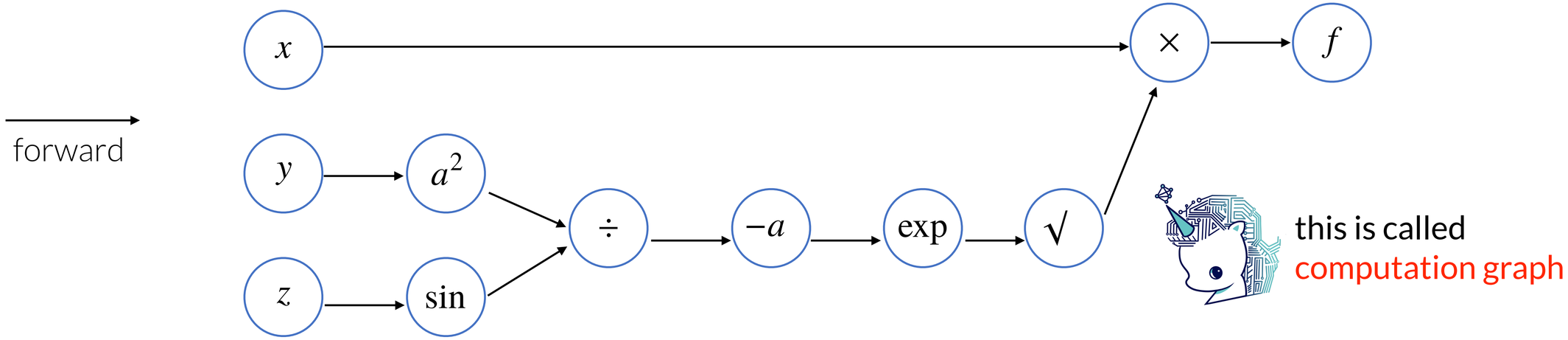
$$\left(\frac{f(x+\epsilon, y, z) - f(x, y, z)}{\epsilon}, \frac{f(x, y+\epsilon, z) - f(x, y, z)}{\epsilon}, \frac{f(x, y, z+\epsilon) - f(x, y, z)}{\epsilon}\right)$$

but difference method
accumulates errors gradually

# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

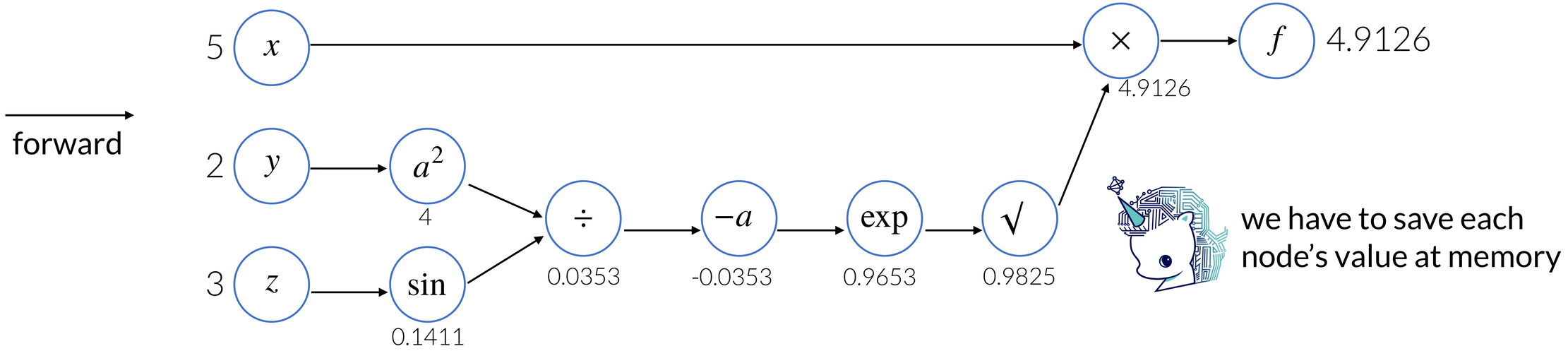
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

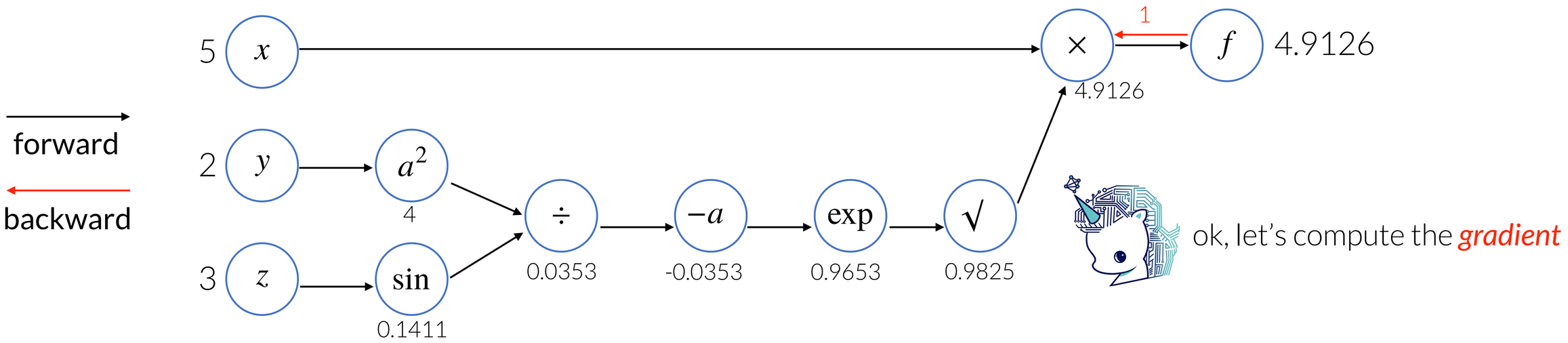
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

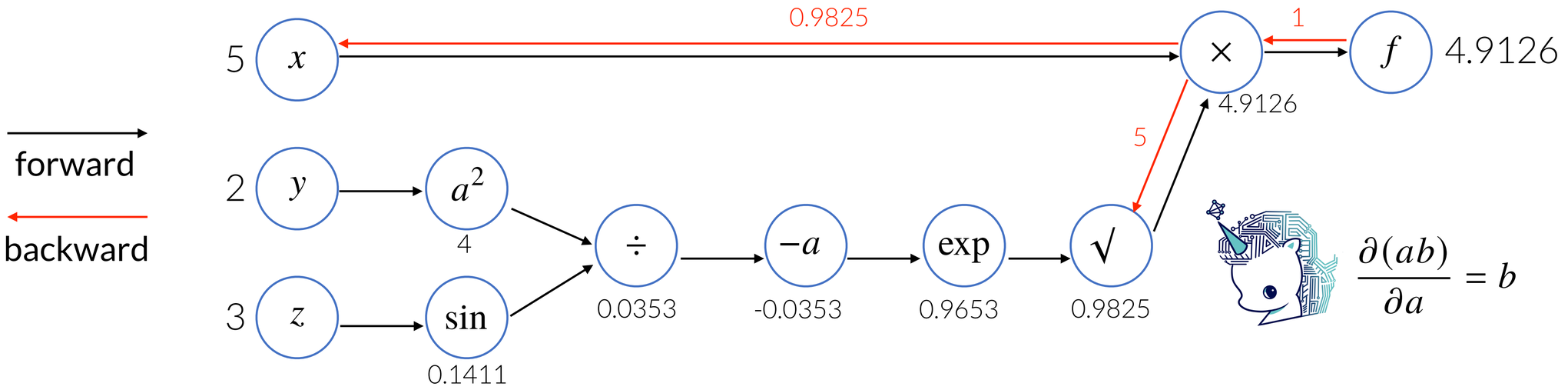
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

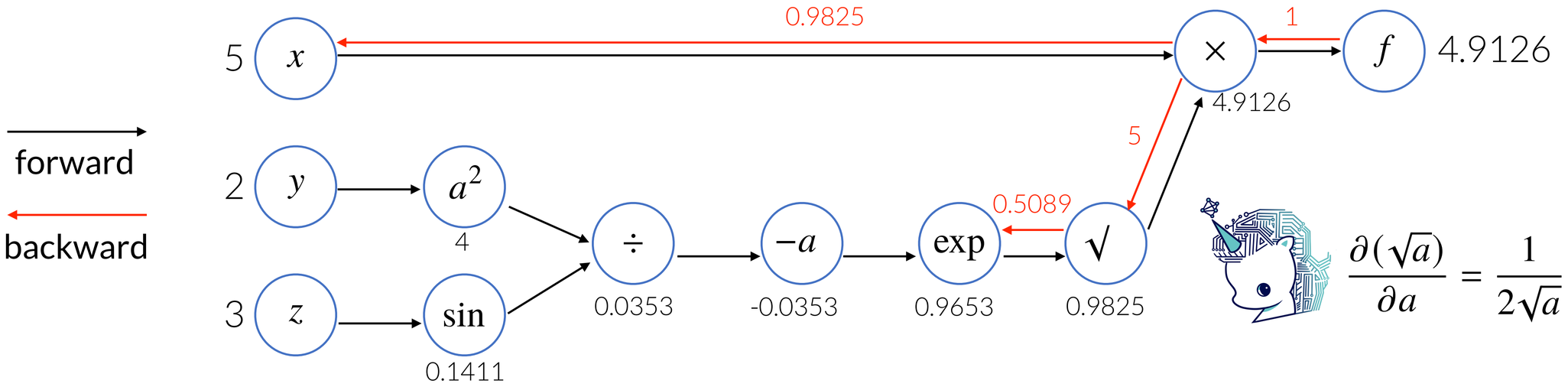
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

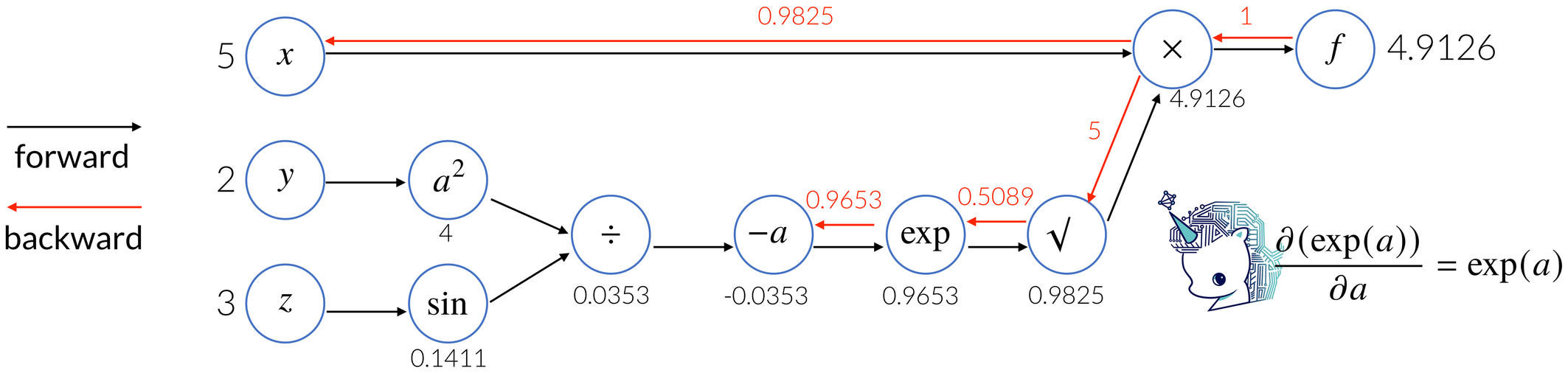
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

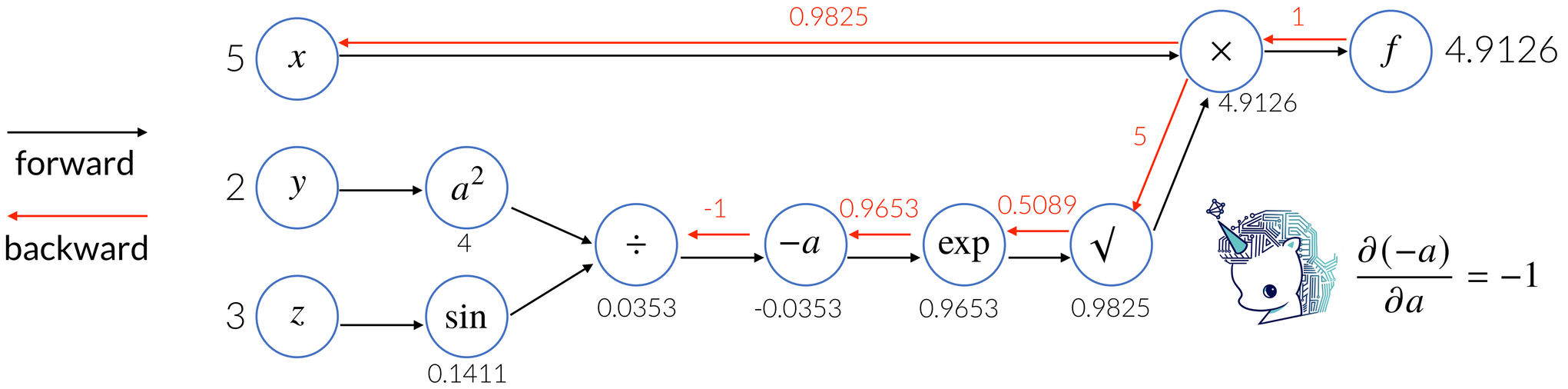
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

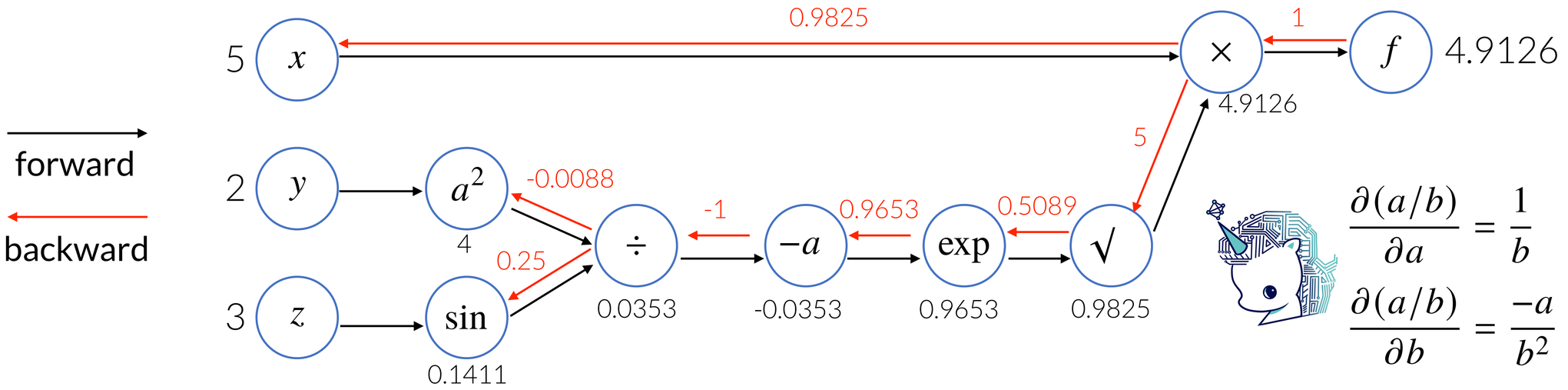
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

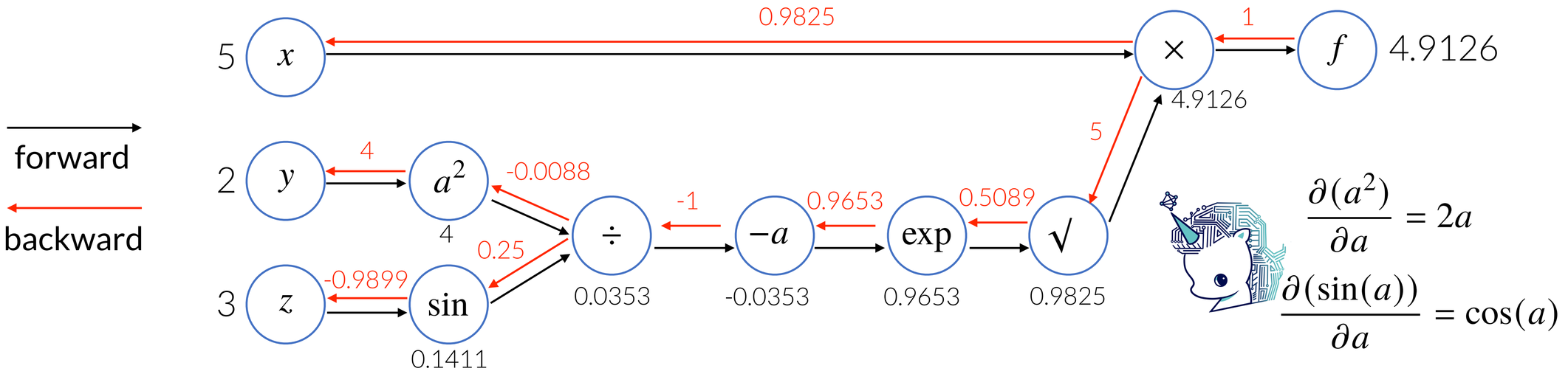
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

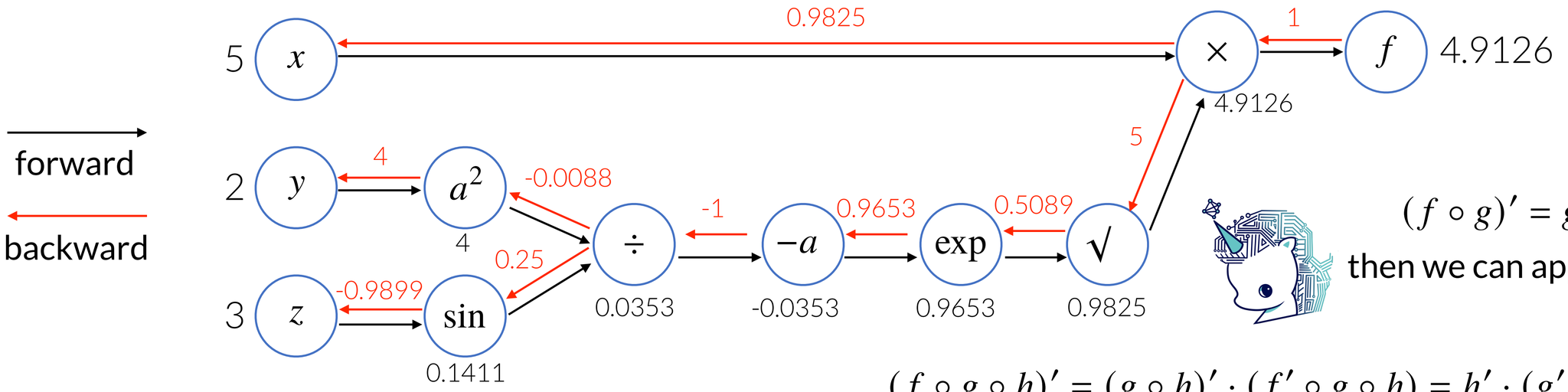
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$




$$(f \circ g)' = g' \cdot (f' \circ g)$$

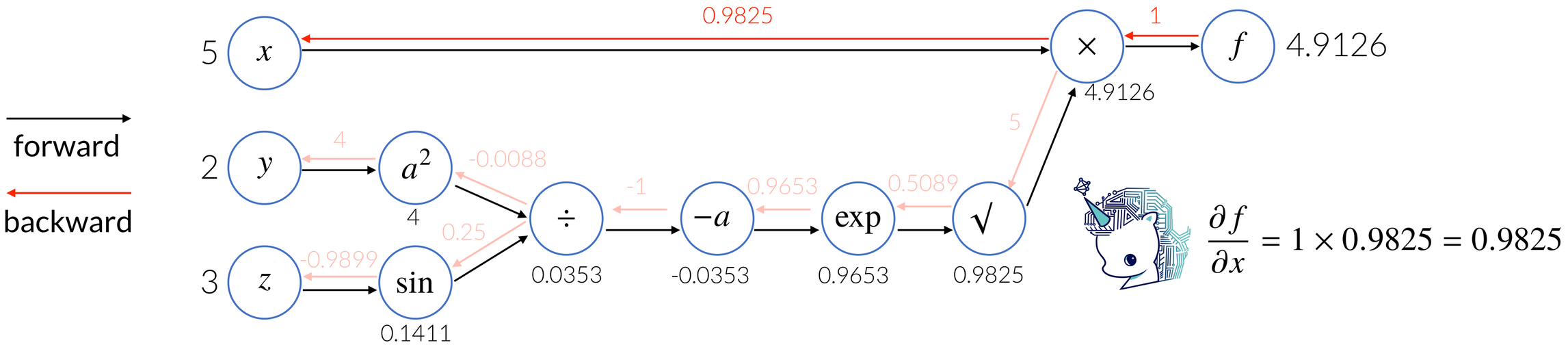
then we can apply chain rule!

$$(f \circ g \circ h)' = (g \circ h)' \cdot (f' \circ g \circ h) = h' \cdot (g' \circ h) \cdot (f' \circ g \circ h)$$

# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

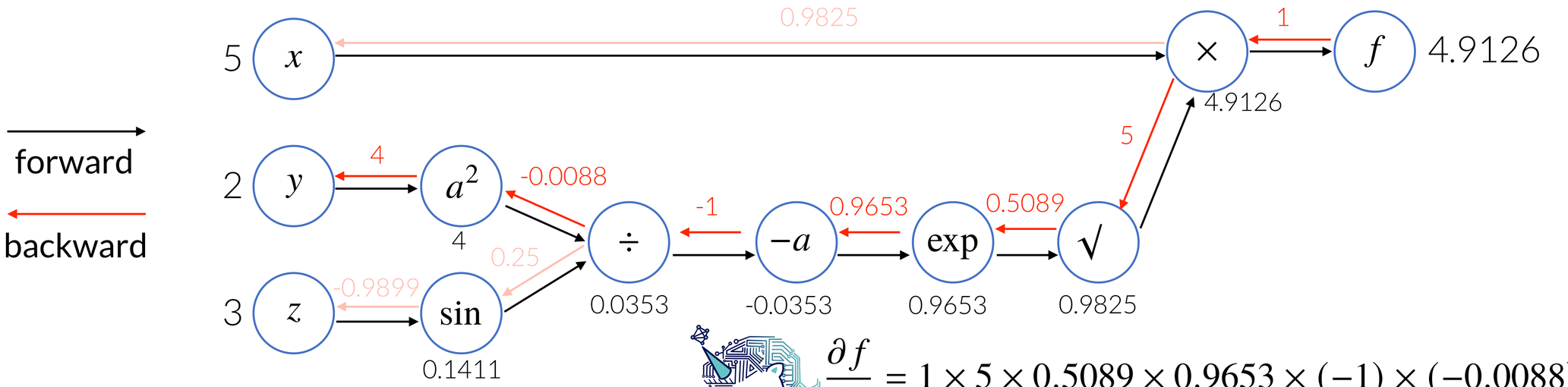
$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$



# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$

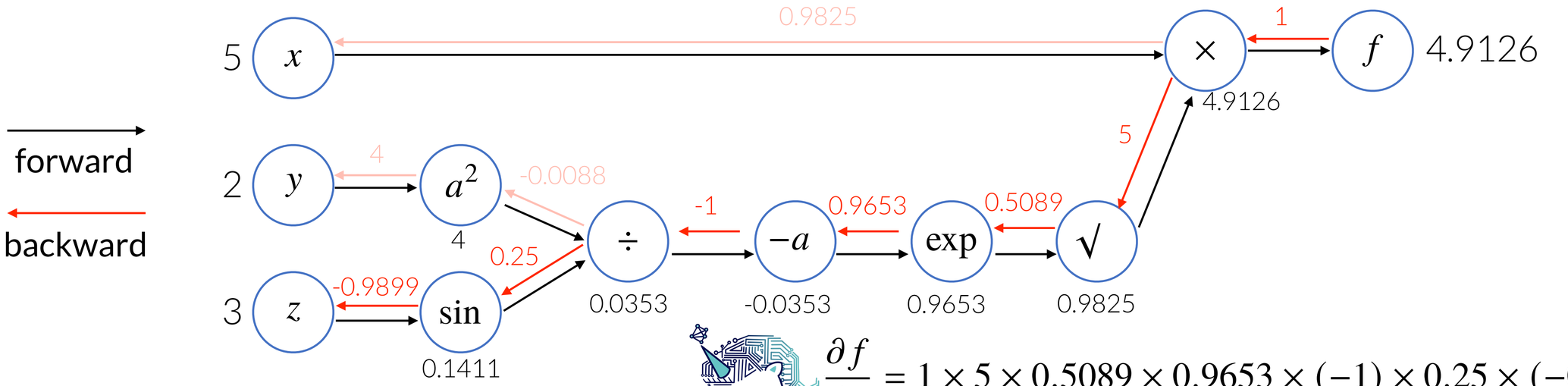



$$\frac{\partial f}{\partial y} = 1 \times 5 \times 0.5089 \times 0.9653 \times (-1) \times (-0.0088) \times 4 = 0.0865$$

# Backpropagation in a Nutshell

- Compute the gradient of $f$ with respect to $x, y, z$

$$f(x, y, z) = x\sqrt{\exp\left(-\frac{\sin z}{y^2}\right)} \qquad \nabla f = (\partial_x f, \partial_y f, \partial_z f) = ?$$




$$\frac{\partial f}{\partial z} = 1 \times 5 \times 0.5089 \times 0.9653 \times (-1) \times 0.25 \times (-0.9899) = 0.6079$$

# Computation Graph

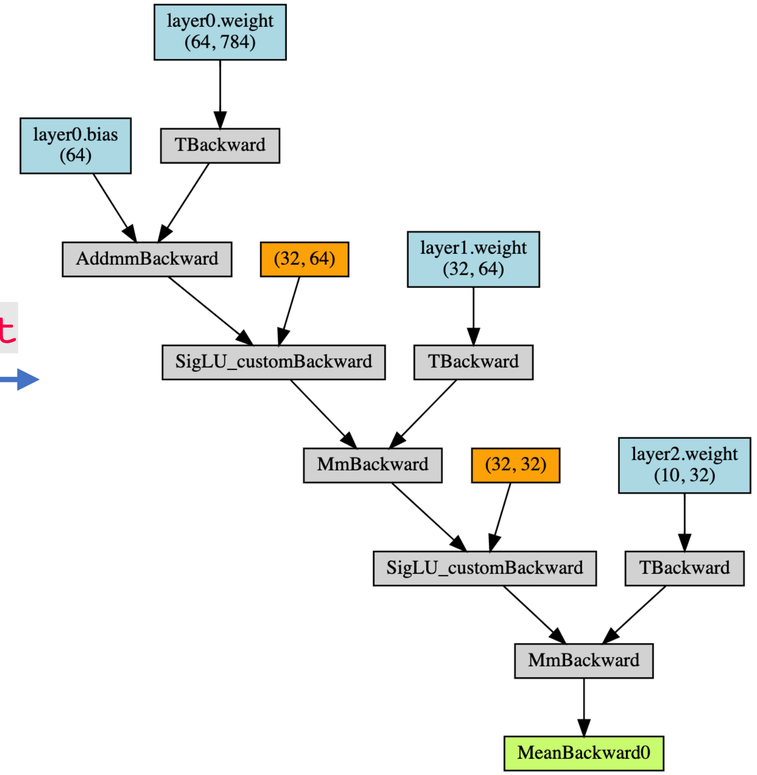
- How can I plot the computation graph? → use `torchviz`

```python
class MLP_SigLU(nn.Module):
    def __init__(self):
        super(MLP_SigLU, self).__init__()
        self.layer0 = torch.nn.Linear(28*28, 64, bias=True)
        # self.act = torch.nn.ReLU()
        self.act = SigLU_custom()
        self.layer1 = torch.nn.Linear(64, 32, bias=False)
        self.layer2 = torch.nn.Linear(32, 10, bias=False)

    def forward(self, inputs):
        inputs = inputs.view(-1, 28*28) # match dimension
        hidden = self.layer0(inputs)
        hidden = self.act.apply(hidden)
        hidden = self.layer1(hidden)
        hidden = self.act.apply(hidden)
        outputs = self.layer2(hidden)
        return outputs

model = MLP_SigLU()
```

`torchviz.make_dot` →



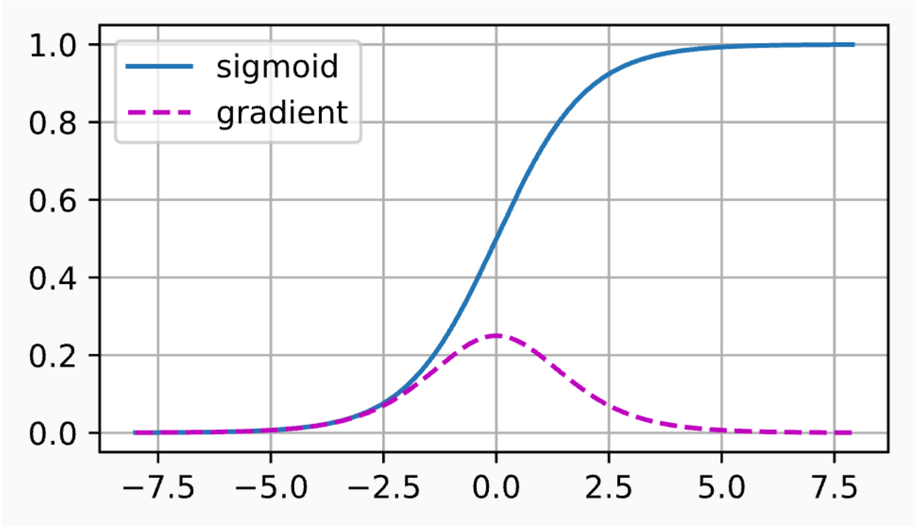
# Instability of Gradients

- In deep learning, the gradient for parameter update equals to the product of the gradient of hidden variables
- If we use sigmoid as an activation function in deep neural network architecture, then gradient vanishes

multi-layer

$$\mathbf{h}^{k+1} = f_k(\mathbf{h}^k) \quad y = f_L \circ \cdots \circ f_1(\mathbf{x})$$

$$\partial_\theta y = \partial_\theta \mathbf{h}^\ell \prod_{k=\ell+1}^{L} \partial_{\mathbf{h}^{k-1}} \mathbf{h}^k$$



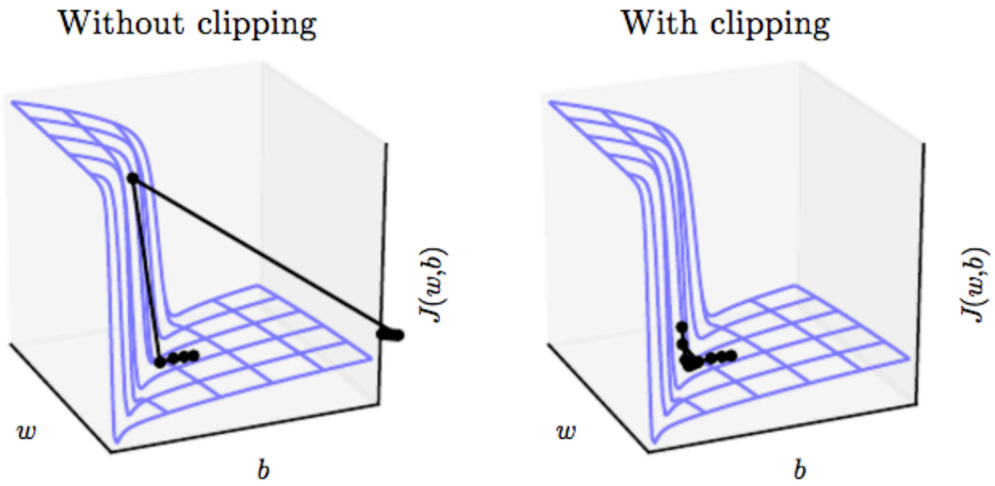
# Instability of Gradients

- In deep learning, the gradient for parameter update equals to the product of the gradient of hidden variables
- If we use sigmoid as an activation function in deep neural network architecture, then gradient vanishes
- On the other hand, gradient explosion can happen when the parameter norm is too huge → need gradient clipping!

multi-layer

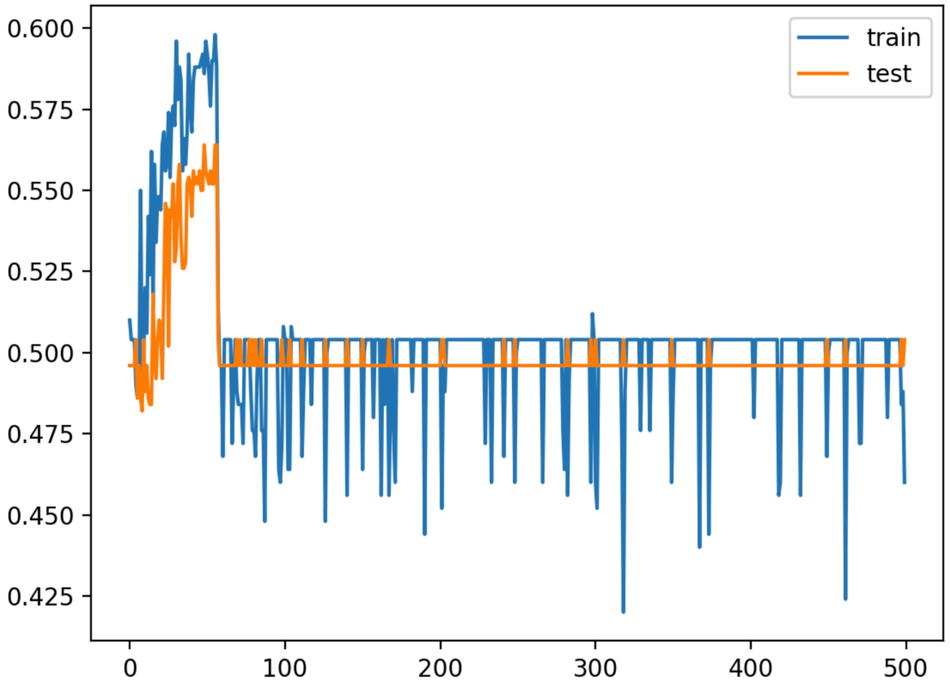
$$\mathbf{h}^{k+1} = f_k(\mathbf{h}^k) \quad y = f_L \circ \cdots \circ f_1(\mathbf{x})$$

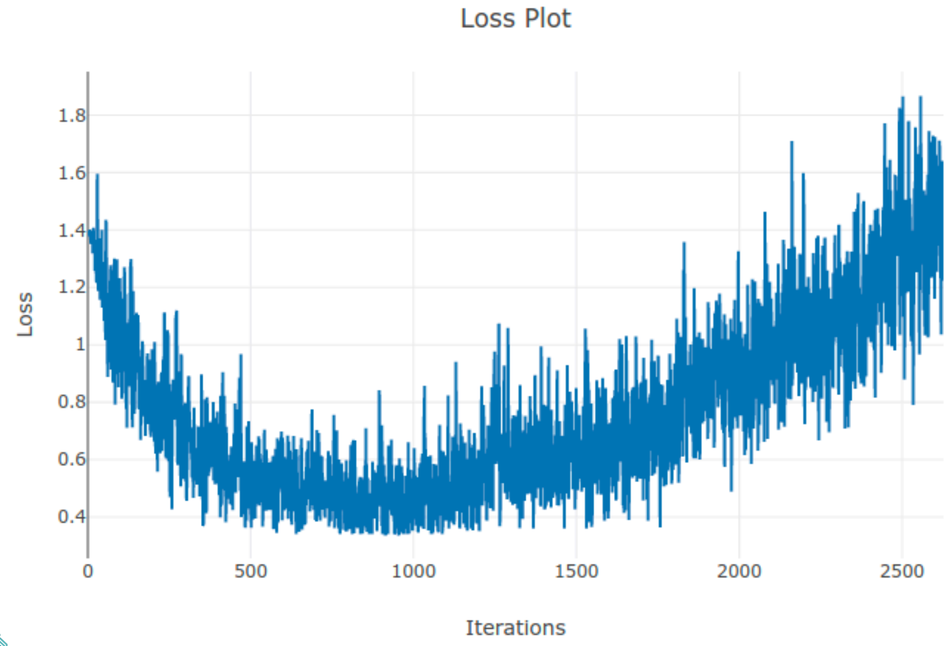
$$\partial_\theta y = \partial_\theta \mathbf{h}^\ell \prod_{k=\ell+1}^{L} \partial_{\mathbf{h}^{k-1}} \mathbf{h}^k$$



# How can we detect those things?

- Tip1: observe the training loss and metric carefully



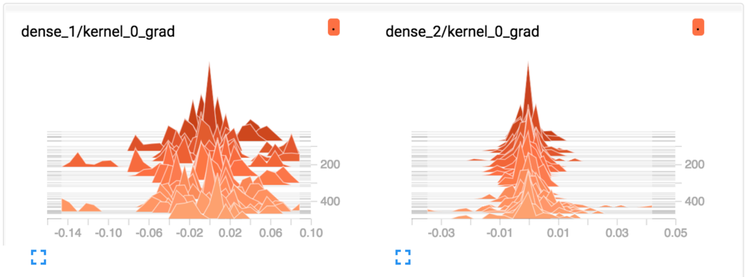




Both are typical examples of gradient explosion

# How can we detect those things?

- Tip1: observe the training loss and metric carefully
- Tip2: use TensorBoard
  - TensorBoard is the visualization tool in TensorFlow, but we can use it in PyTorch too!

you can use `TensorBoard` in PyTorch




TORCH.UTILS.TENSORBOARD

Before going further, more details on TensorBoard can be found at https://www.tensorflow.org/tensorboard/

# Multilayer Perceptron

- We define MLP as the multiple composition of perceptrons

$$\mathrm{MLP}(\mathbf{x};\boldsymbol{\Theta}) := f(\mathbf{h}^{(L)} \circ \cdots \circ \mathbf{h}^{(1)}) \qquad \ell \in \{1,\ldots,L\}$$

parameters $[\mathbf{w}^{(\ell)},\mathbf{b}^{(\ell)}]_{\ell=1}^{L}$

layer index

$$\mathbf{h}^{(\ell)} := \varphi(\mathbf{w}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}) \qquad \mathbf{h}^{(0)} = \mathbf{x} \in \mathbb{R}^{d}$$

- $\mathbf{h}^{(\ell)}$: hidden vector in the $\ell$-layer
- $\varphi$: activation
- $\mathbf{w}^{(\ell)}$: weight parameter (matrix)
- $\mathbf{b}^{(\ell)}$: bias parameter (vector)
- $\mathbf{h}^{(0)} = \mathbf{x}$: input vector

we can use different activation for each layer
but there is no learnable parameter

# Multilayer Perceptron

- We define MLP as the multiple composition of perceptrons

$$\mathrm{MLP}(\mathbf{x};\boldsymbol{\Theta}) := f(\mathbf{h}^{(L)} \circ \cdots \circ \mathbf{h}^{(1)}) \qquad \ell \in \{1,\ldots,L\}$$

parameters $[\mathbf{w}^{(\ell)},\mathbf{b}^{(\ell)}]_{\ell=1}^{L}$

layer index

$$\mathbf{h}^{(\ell)} := \varphi(\mathbf{w}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}) \qquad \mathbf{h}^{(0)} = \mathbf{x} \in \mathbb{R}^{d}$$

- $\mathbf{h}^{(\ell)}$: hidden vector in the $\ell$-layer
- $\varphi$: activation
- $\mathbf{w}^{(\ell)}$: weight parameter (matrix)
- $\mathbf{b}^{(\ell)}$: bias parameter (vector)
- $\mathbf{h}^{(0)} = \mathbf{x}$: input vector

- Question: how many parameters do we need for MLP?

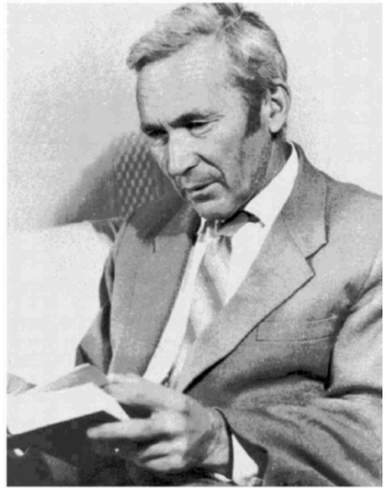
$$|\boldsymbol{\Theta}| \approx \sum_{\ell=1}^{L} \dim(\mathbf{h}^{(\ell)}) \cdot [\dim(\mathbf{h}^{(\ell-1)}) + 1]$$

# Properties of MLP

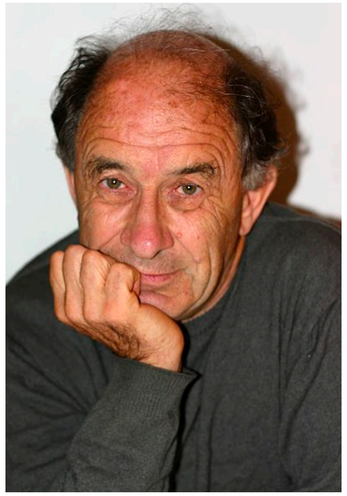
- Pros
  - simple-to-implement
  - no inductive bias  **it can be discontinuous!**
- Cons
  - too dense layers which are fully connected → curse of dimension!
  - computational cost is too heavy → slow runtime, heavy memory usage
  - overparameterization → prone to overfitting

# Universal Approximation Theorem

- Hornik-Cybenko Theorem
- Kolmogorov-Arnold representation theorem (Hilbert's 13th problem)



A. Kolmogorov



V. Arnold

K. Hornik

G. Cybenko

# UAT for Deep Neural Networks

- Arbitrary width case
  - classical result: approximates arbitrary continuous functions
- Arbitrary depth case
  - Zhou Lu et al. (2017): approximates any ***integrable*** functions
  - width must be bigger than $\mathbf{max}\{n+1, m\}$

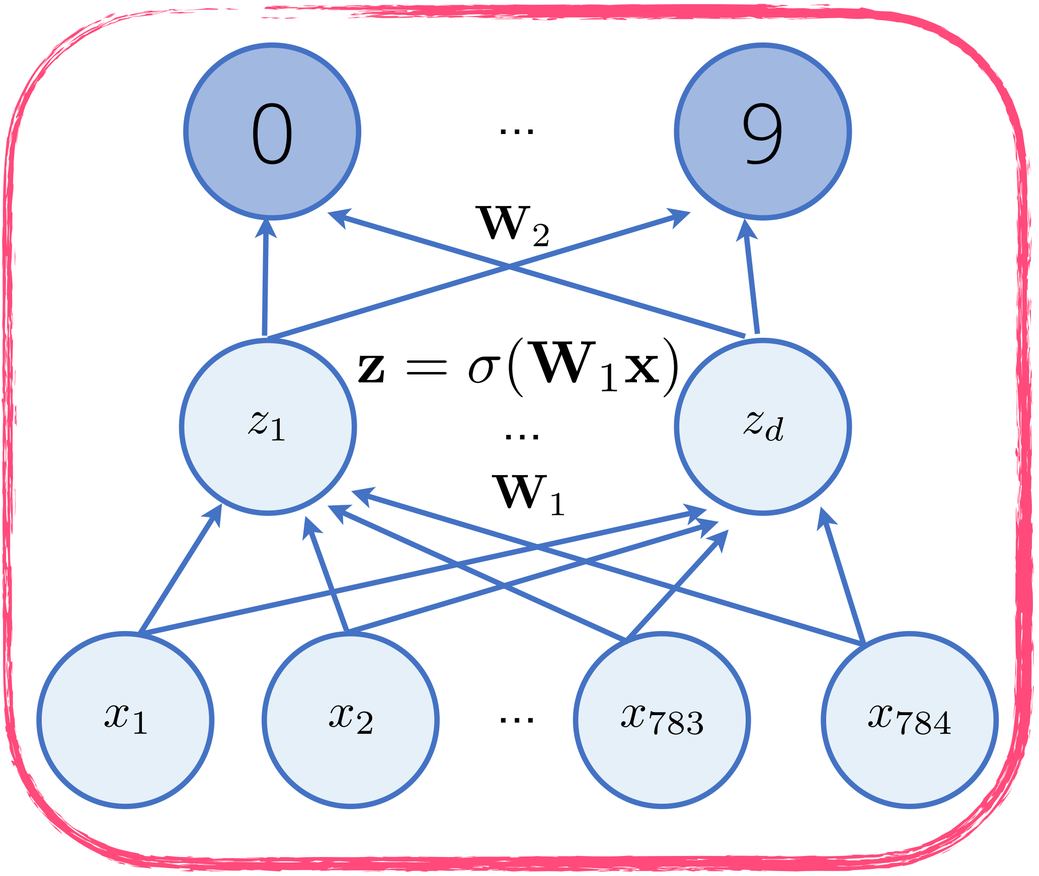
$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

it can be discontinuous!

- UAT for Graph Data
  - Brüel-Gabrielsson (2020)
  - graph representation with iso-injective properties

# PyTorch Implementation

we consider MLP with a single hidden layer and ReLU activation



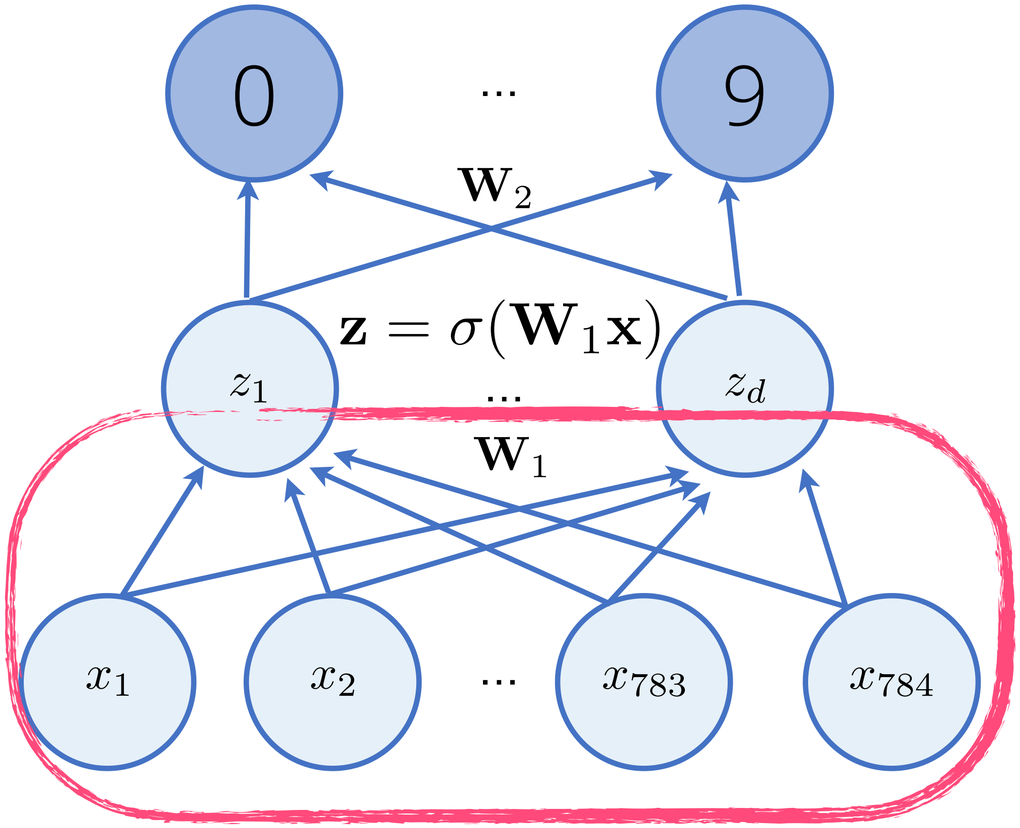

```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layer0 = torch.nn.Linear(28*28, 32, bias=True)
        self.act = torch.nn.ReLU()
        self.layer1 = torch.nn.Linear(32, 10, bias=False)

    def forward(self, inputs):
        inputs = inputs.view(-1, 28*28) # match dimension
        hidden = self.layer0(inputs)
        hidden = self.act(hidden)
        outputs = self.layer1(hidden)
        return outputs

model = MLP() # GPU
```

# PyTorch Implementation

here we assume 28x28 2D image as an input data



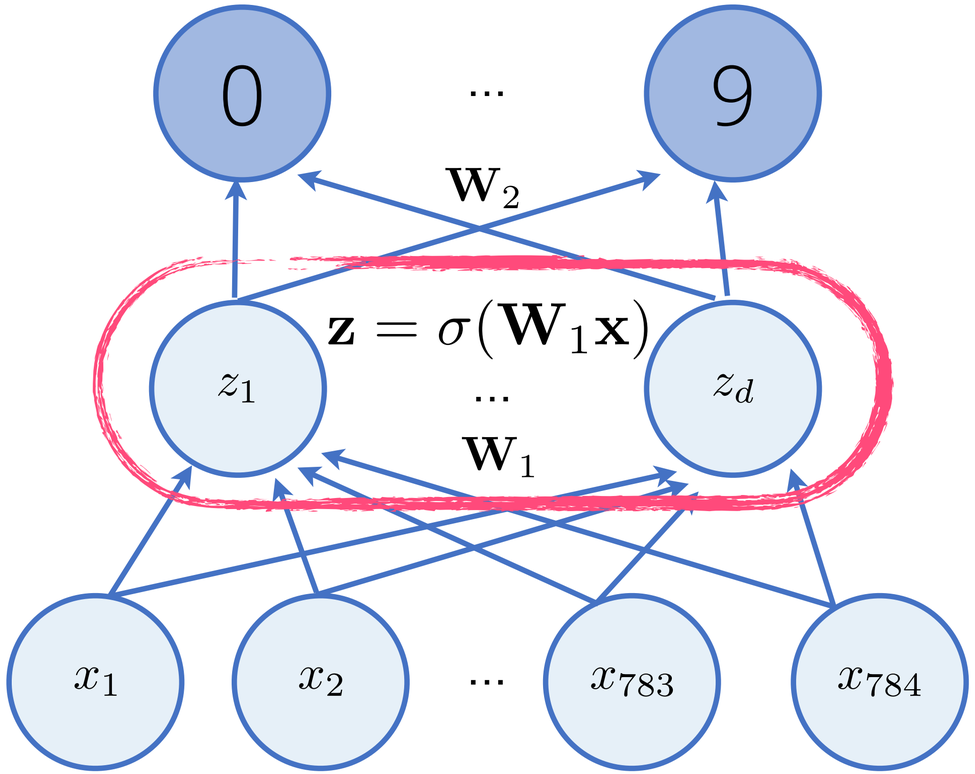

```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layer0 = torch.nn.Linear(28*28, 32, bias=True)
        self.act = torch.nn.ReLU()
        self.layer1 = torch.nn.Linear(32, 10, bias=False)

    def forward(self, inputs):
        inputs = inputs.view(-1, 28*28) # match dimension
        hidden = self.layer0(inputs)
        hidden = self.act(hidden)
        outputs = self.layer1(hidden)
        return outputs

model = MLP() # GPU
```

# PyTorch Implementation

there is no learning parameter
in the activation function



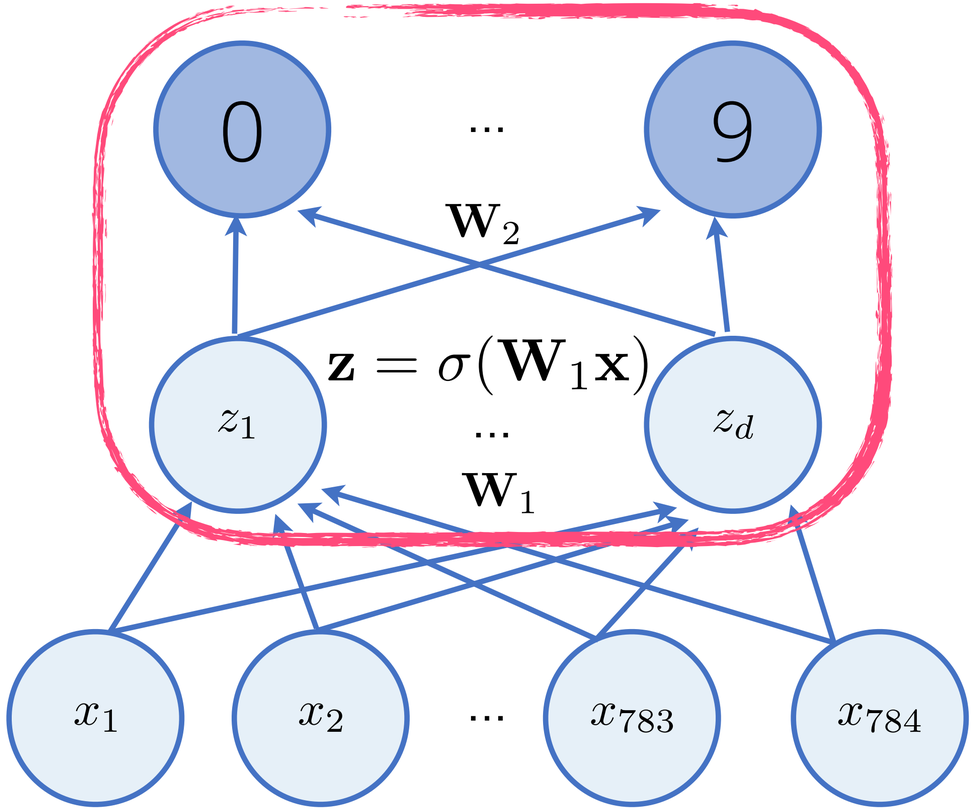

```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layer0 = torch.nn.Linear(28*28, 32, bias=True)
        self.act = torch.nn.ReLU()
        self.layer1 = torch.nn.Linear(32, 10, bias=False)

    def forward(self, inputs):
        inputs = inputs.view(-1, 28*28) # match dimension
        hidden = self.layer0(inputs)
        hidden = self.act(hidden)
        outputs = self.layer1(hidden)
        return outputs

model = MLP() # GPU
```

# PyTorch Implementation

how many parameters are there for this multilayer perceptron?




```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layer0 = torch.nn.Linear(28*28, 32, bias=True)
        self.act = torch.nn.ReLU()
        self.layer1 = torch.nn.Linear(32, 10, bias=False)

    def forward(self, inputs):
        inputs = inputs.view(-1, 28*28) # match dimension
        hidden = self.layer0(inputs)
        hidden = self.act(hidden)
        outputs = self.layer1(hidden)
        return outputs

model = MLP() # GPU
```

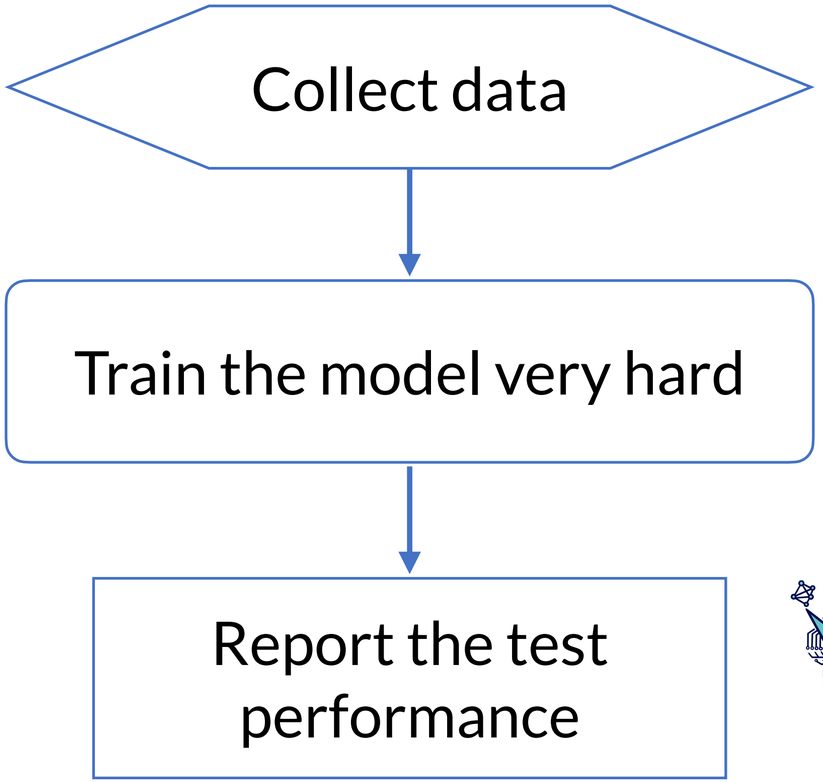
# Model Selection

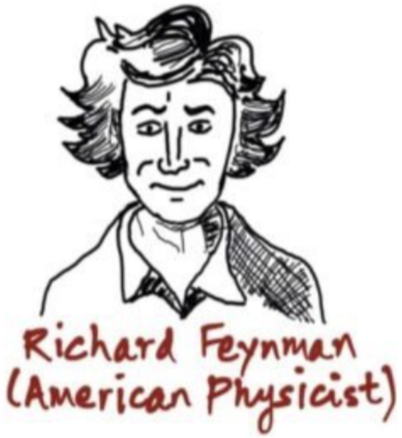
Principles of Deep Learning (AI502/IE408/IE511)

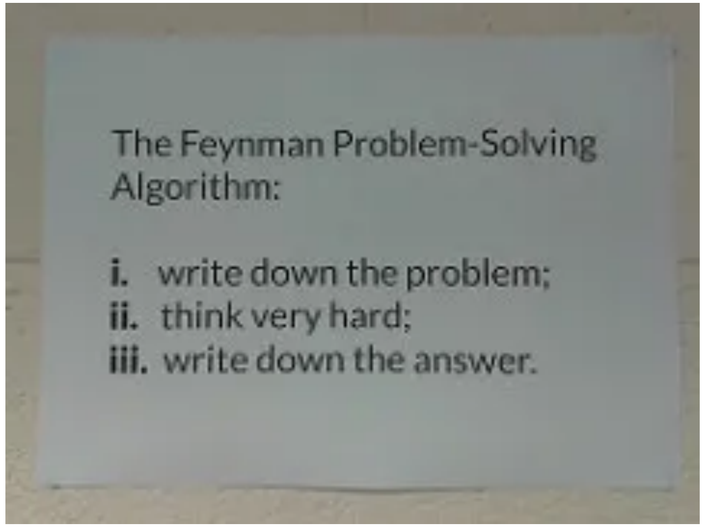
Sungbin Lim (UNIST AIGS & IE)


Contact: ai502deeplearning@gmail.com

# Machine Learning Algorithm



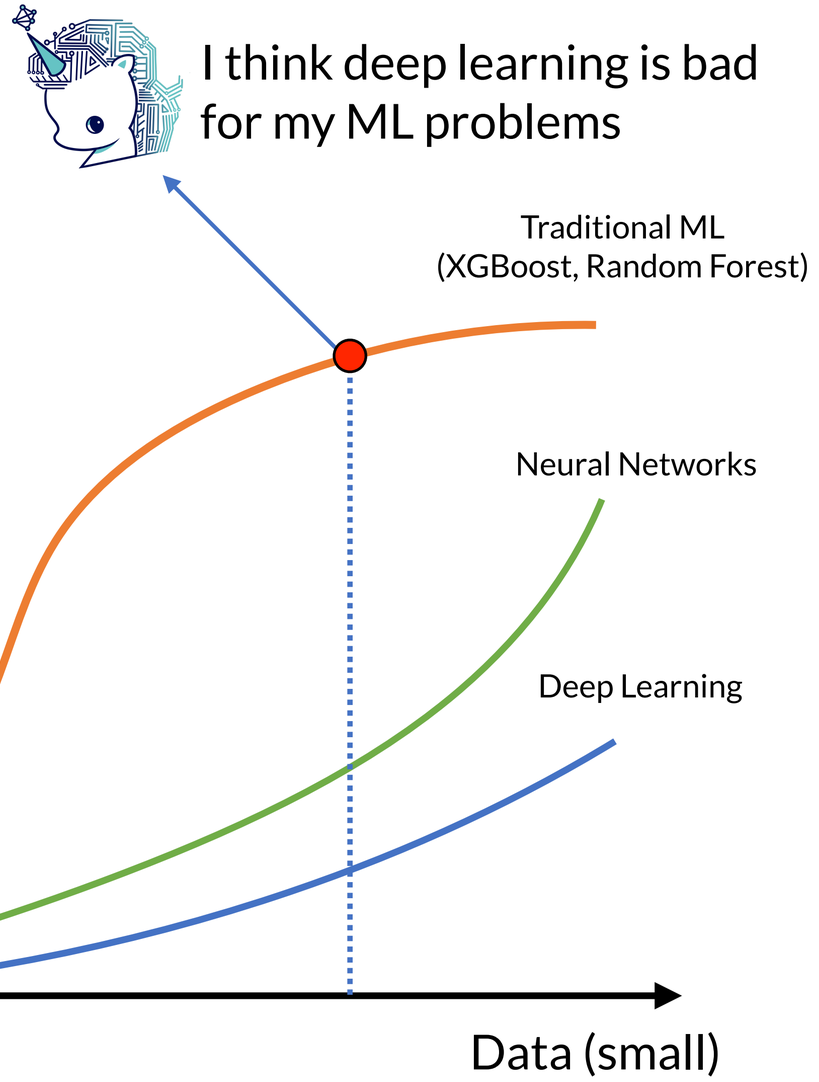




which one is most difficult?

# Gather more fine data

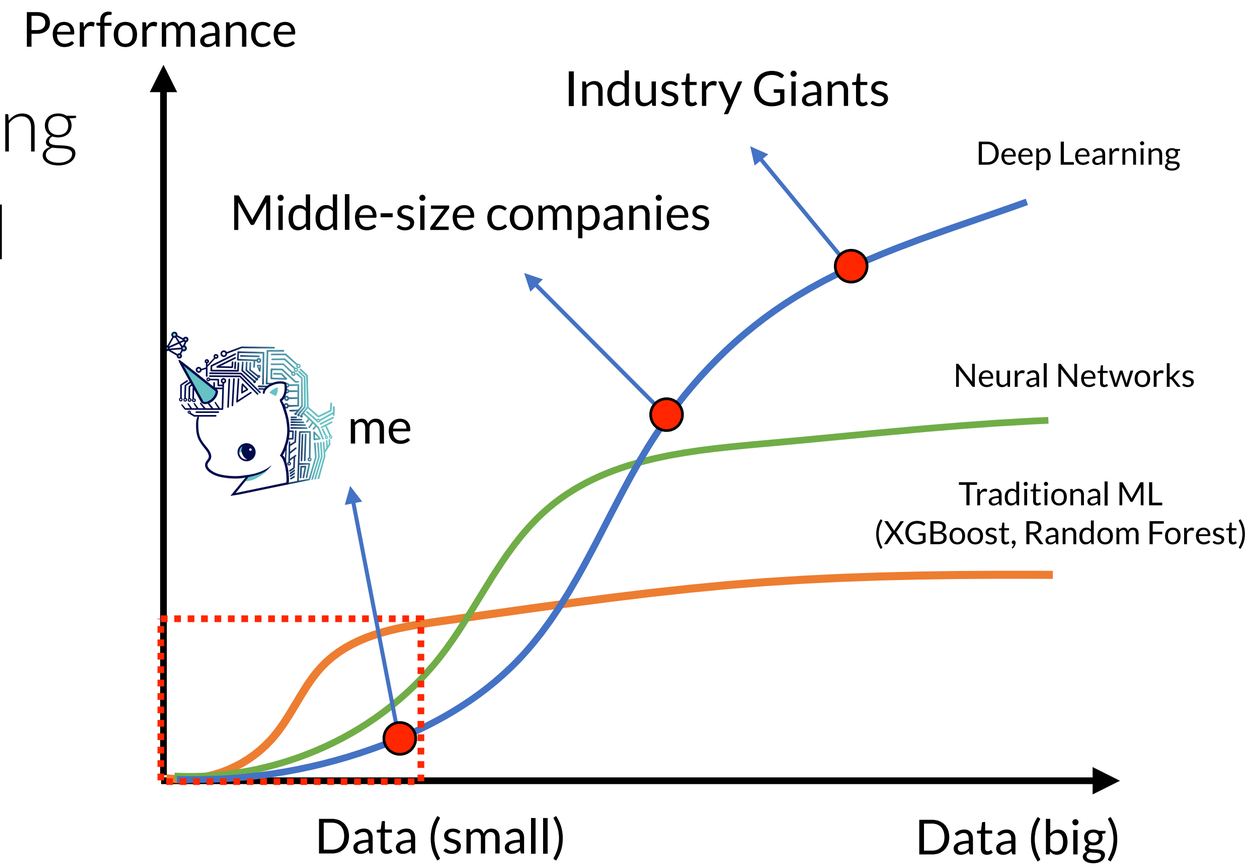
- Golden rule of deep learning
  - small data is not for deep learning




How Scale is Enabling Deep Learning, Andrew Ng, 2016

# Gather more fine data

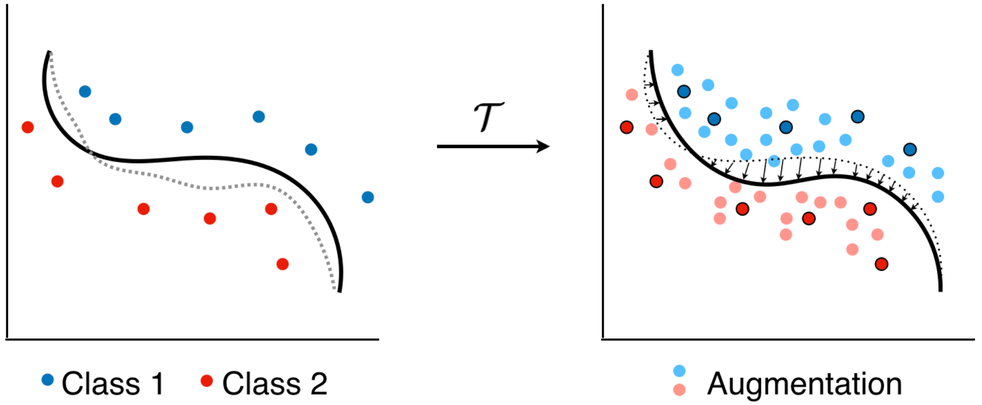
- Golden rule of deep learning
  - small data is not for deep learning
  - deep learning is good at big and high-dimensional data
- How big?
  - you need at least hundreds of GPUs to train those data

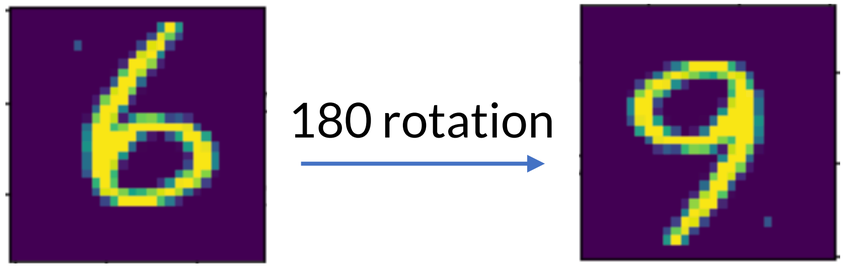



How Scale is Enabling Deep Learning, Andrew Ng, 2016

# Data Augmentation

- If you have insufficient data, then augment it!
  - Empirically, augmentation is the best strategy for data problem
  - ***Pros***
    - Well-suited for DL
    - Easier than network tuning
  - ***Cons***
    - Require domain knowledges




Fast AutoAugment, Lim et al., ***NeurIPS*** 2019

# Weight decay

- Another technique to reduce overfitting is adding the square of L2-norm of weight parameters to the loss function

$$\mathcal{L}(\boldsymbol{\Theta}, \mathcal{D}_{\text{train}}) + \lambda \|\boldsymbol{\Theta}\|_2^2$$

weight decay term

weight decay coefficient

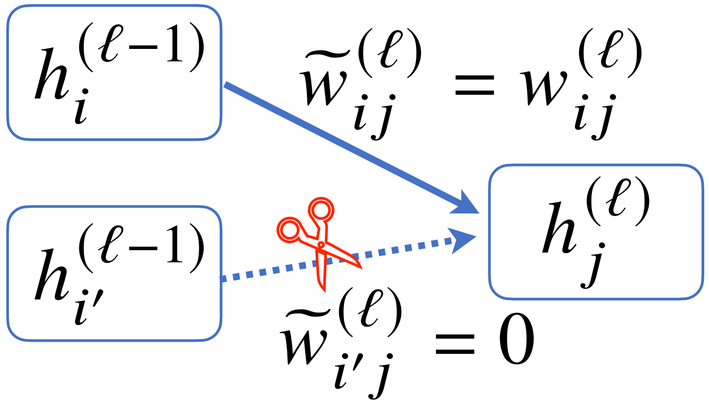
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} (\mathcal{L} + \lambda \|\mathbf{w}\|_2^2) = (1 - \eta\lambda)\mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}$$

- As you apply the weight decay longer, the weight parameter vanishes
- Weight decay is easy-to-implement

```
optimizer = torch.optim.SGD(model.parameters(),
                            lr=0.03,
                            weight_decay=0.1)
```

# Dropout

- Dropout is a simple but powerful technique to reduce overfitting by cutting connection between hidden variables
  - plugging zero weight parameters randomly during the training
  - recover trained parameters during prediction
- Training result can be stochastic and hinder the convergence

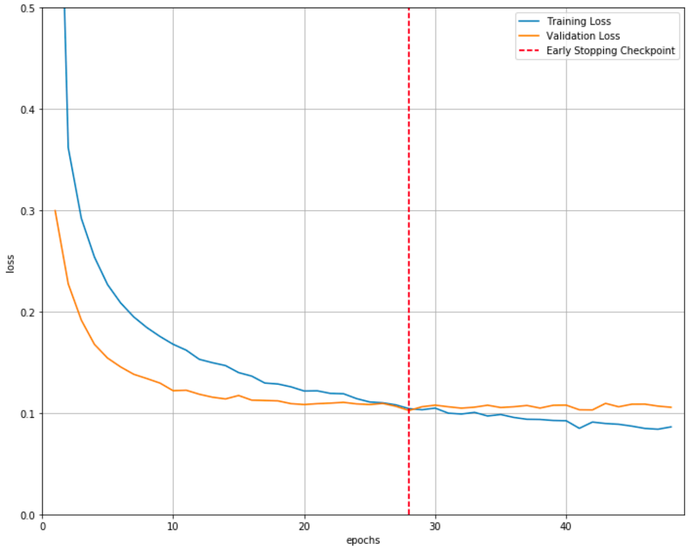



$$\tilde{w}_{ij}^{(\ell)} = \begin{cases} 0 & \text{with probability } p \\ w_{ij}^{(\ell)} & \text{with probability } 1-p \end{cases}$$

# Early Stopping

- Stop the training before the model begins overfitting
  - cross validation is necessary (**DON'T** use test set!)
  - ***Pros***
    - simple but powerful
  - ***Cons***
    - needs some heuristics

please do not touch test set
until the end of overall training



# Label Smoothing

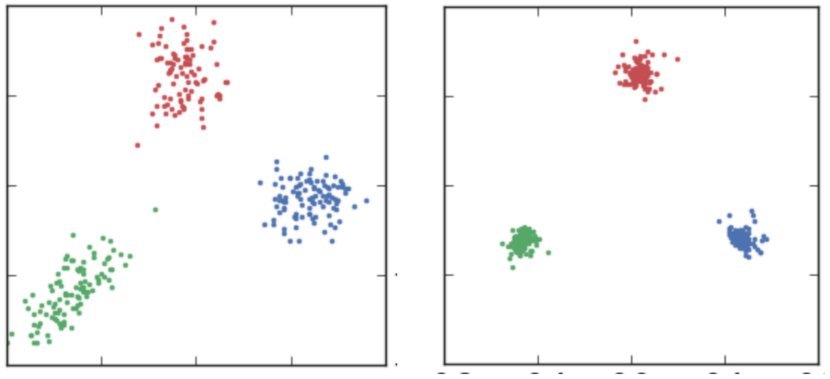
- Solution for over-confidence problem
  - Modify one-hot vector to smoothed label:
  - ***Pros***
    - Works quite well for insufficient data
    - Robust to mislabeled data
  - ***Cons***
    - Only for classification tasks

$$\tilde{y} = (1 - \epsilon)y + \frac{\epsilon}{C}$$

hyperparameter

one-hot vector

# of classes

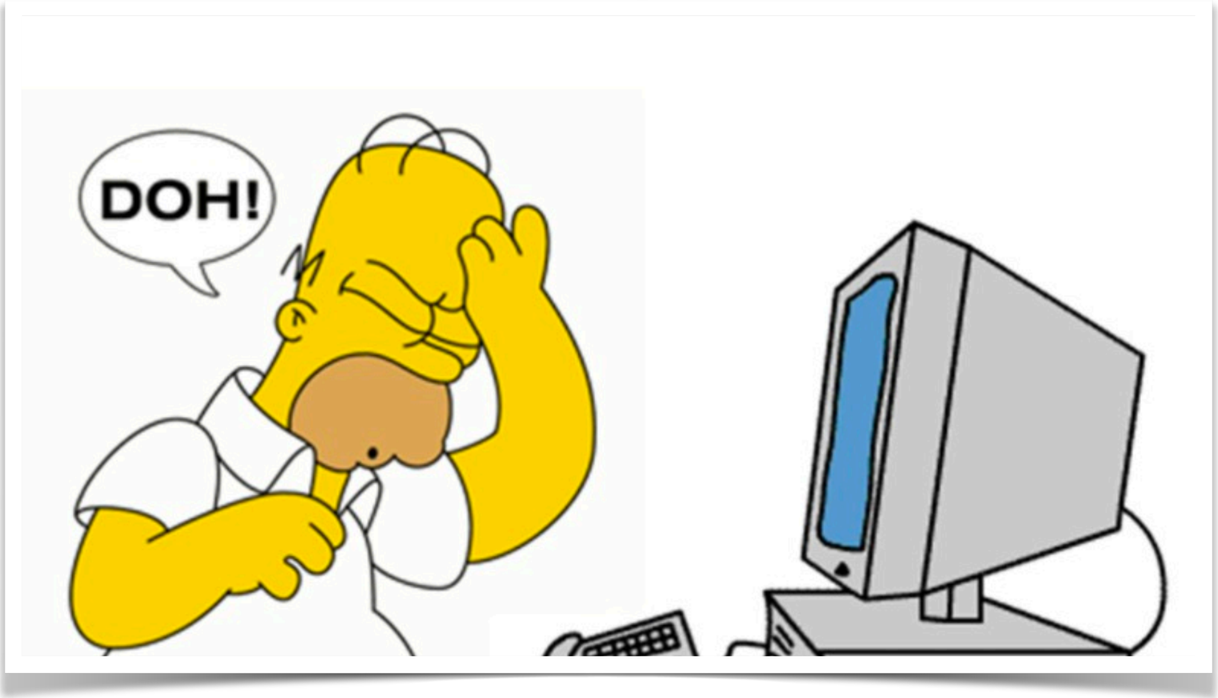


without LS

with LS

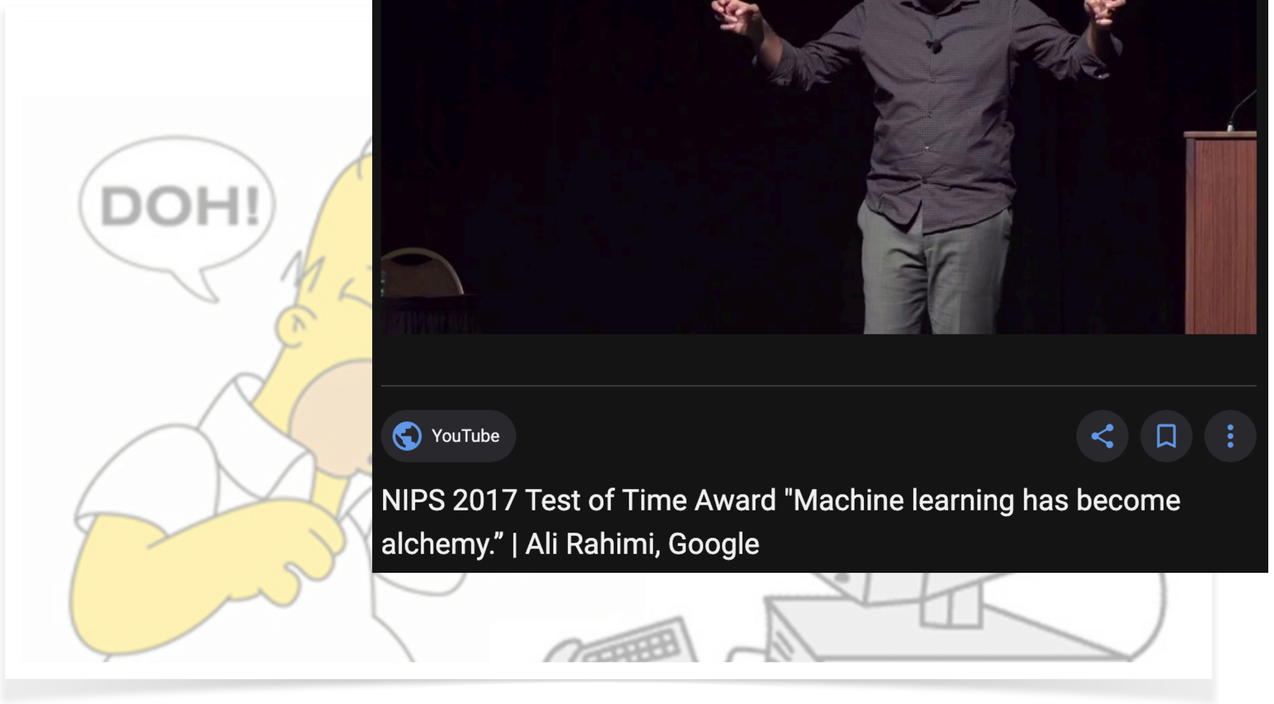
# More Hyperparameters!

- Epoch numbers
- Learning rate
- Mini-batch size
- Number of layers
- Activation functions
- Dimension of hidden-variables



# More Hyperparameters!

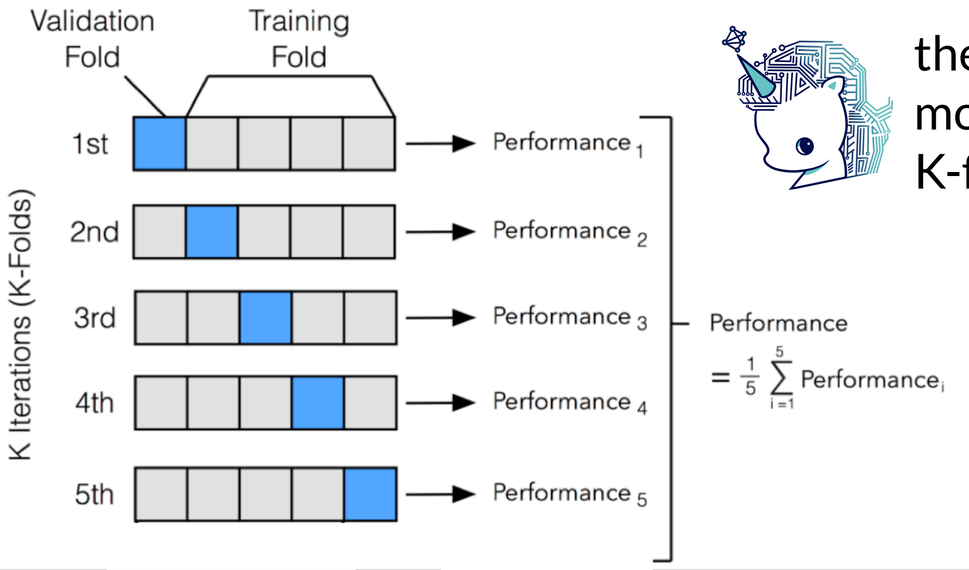
- Epoch numbers + *early stopping*
- Learning rate + *scheduling*
- Mini-batch size + *normalization*
- Number of layers + *multi-branch*
- Activation functions + *layer-wise*
- Dimension of hidden-variables




**how can we choose the best model?**

# How to choose the best model?

- Many researchers pick the best model according to the test performance
  - this is not robust! (why?)
- We have to compare the results in the language of statistics
  - K-fold cross-validation
  - Bootstrapping
  - Deep Ensemble






the first assignment is model selection using K-fold cross validation

# Q&A