Training neural networks

EECE454 Intro. to Machine Learning Systems

Fall 2024

Recap

- **Last week.** What deep learning is, and how we train deep neural networks
    - Basic algorithm: Stochastic Gradient Descent (SGD)

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \hat{\nabla}_\theta L(\theta)$$

Recap

- **Last week.** What deep learning is, and how we train deep neural networks
    - Basic algorithm: Stochastic Gradient Descent (SGD)

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \hat{\nabla}_\theta L(\theta)$$

- Evaluating the gradients required backpropagation:
    - Forward: Compute intermediate activations and store them in memory

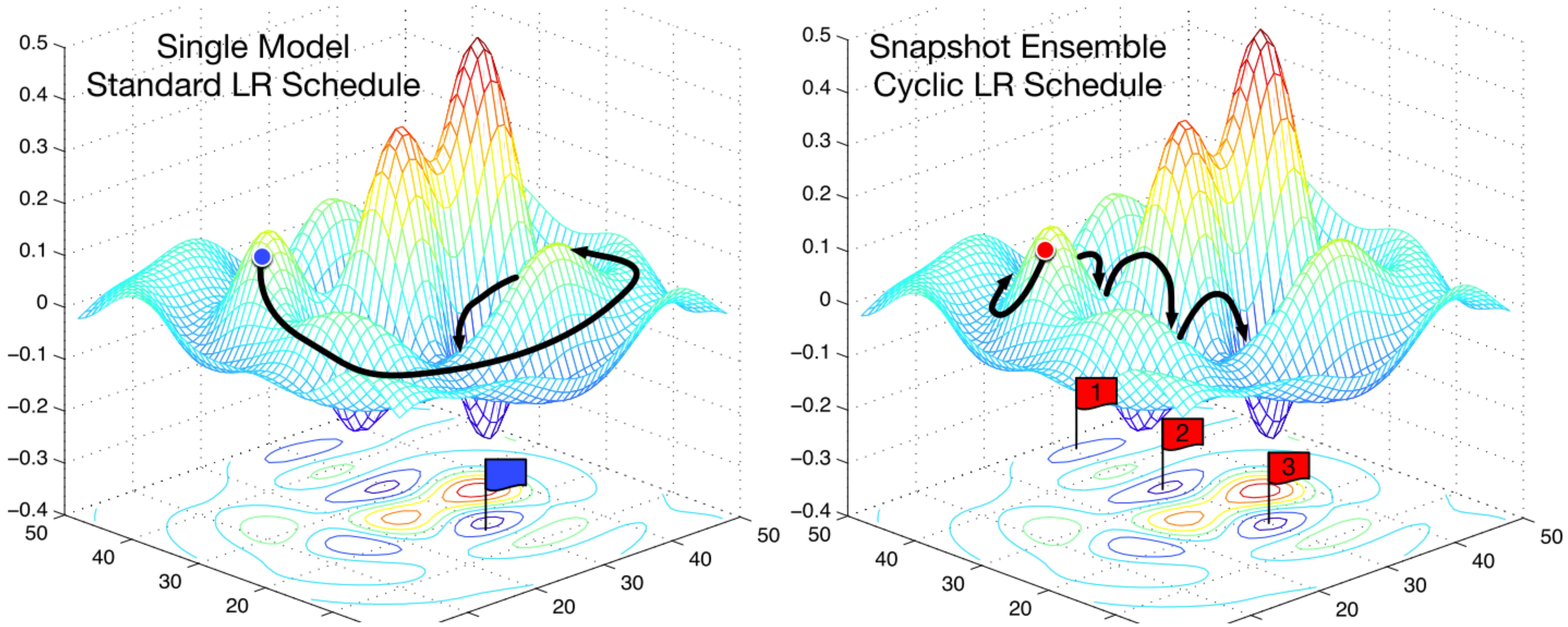
$$\mathbf{z} = f_1(\mathbf{x}; \mathbf{W}_1), \qquad f(\mathbf{x}) = f_2(\mathbf{z}; \mathbf{W}_2)$$

- Backward: Combine modular gradients to compute the gradient via chain rule

$$\frac{\partial f}{\partial \mathbf{W}_1} = \frac{\partial f_2}{\partial \mathbf{z}} \frac{\partial f_1}{\partial \mathbf{W}_1}$$

This week

- Neural net training is actually quite difficult; can lead to ...
    - Fails to converge to a well-generalizing
    - Excessive time / computation for convergence



Source: Meta AI, "OPT: Open Pre-trained Transformer Language Models," 2022

2.5 Training Processes

Here we describe significant training process adjustments that arose during OPT-175B pre-training.

Hardware Failures We faced a significant number of hardware failures in our compute cluster while training OPT-175B. In total, hardware failures contributed to at least 35 manual restarts and the cycling of over 100 hosts over the course of 2 months. During manual restarts, the training run was paused, and a series of diagnostics tests were conducted to detect problematic nodes. Flagged nodes were then cordoned off and training was resumed from the last saved checkpoint. Given the difference between the number of hosts cycled out and the number of manual restarts, we estimate 70+ automatic restarts due to hardware failures.

Loss Divergences Loss divergences were also an issue in our training run. When the loss diverged, we found that lowering the learning rate and restarting from an earlier checkpoint allowed for the job to recover and continue training. We noticed a correlation between loss divergence, our dynamic loss scalar crashing to 0, and the l^2-norm of the activations of the final layer spiking. These observations led us to pick restart points for which our dynamic loss scalar was still in a "healthy" state (≥ 1.0), and after which our activation norms would trend downward instead of growing unboundedly. Our empirical LR schedule is shown in Figure 1. Early in training, we also noticed that lowering gradient clipping from 1.0 to 0.3 helped with stability; see our released logbook for exact details. Figure 2 shows our validation loss with respect to training iterations.

Other Mid-flight Changes We conducted a number of other experimental mid-flight changes to handle loss divergences. These included: switching to vanilla SGD (optimization plateaued quickly, and we reverted back to AdamW); resetting the dynamic loss scalar (this helped recover some but not all divergences); and switching to a newer version of Megatron (this reduced pressure on activation norms and improved throughput).

https://github.com/google-research/tuning_playbook

README License

Deep Learning Tuning Playbook

This is not an officially supported Google product.

Varun Godbole[†], George E. Dahl[†], Justin Gilmer[†], Christopher J. Shallue[‡], Zachary Nado[†]

[†] Google Research, Brain Team

[‡] Harvard University

Table of Contents

This week

- Fortunately, people tend to agree on basic principles
    - **Today.** Setting up the training
        - Gradients and activation functions
        - Data preprocessing
        - Normalization layers
        - Parameter Initialization

This week

- Fortunately, people tend to agree on basic principles
  - **Today.** Setting up the training
    - Gradients and activation functions
    - Data preprocessing
    - Normalization
    - Parameter Initialization
  - **Next class.** Tuning the training process
    - Learning rates
    - Batch size
    - Regularizers
    - Optimizers
    - Hyperparameter tuning and troubleshooting

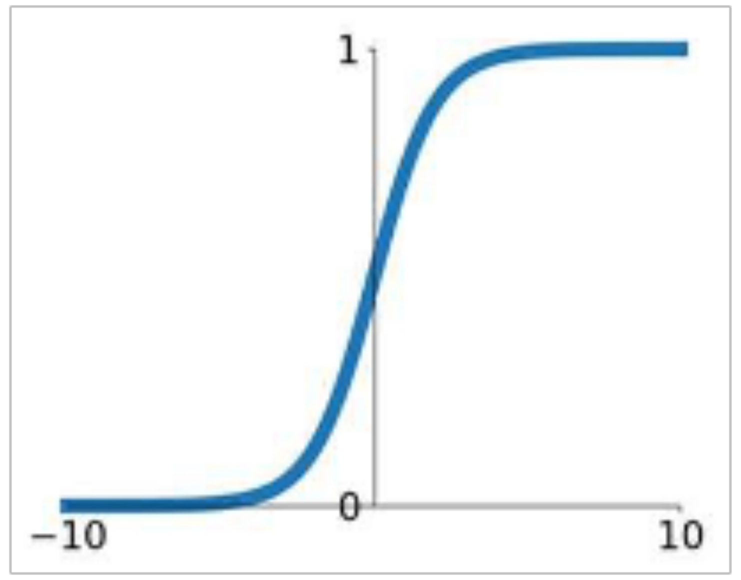
Activation functions

Fall of sigmoids

- **Recall.** Sigmoidal activations were popular in the past
  - Similar to $\mathbf{1}[\,\cdot\,]$, and serves as a good surrogate
  - Biological interpretation as a firing rate of a neuron
  - Easy to compute the gradient — $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$

Sigmoid

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



Fall of sigmoids

- **Recall.** Sigmoidal activations were popular in the past
    - Similar to $\mathbf{1}[\,\cdot\,]$, and serves as a good surrogate
    - Biological interpretation as a firing rate of a neuron
    - Easy to compute the gradient — $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$

Sigmoid

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

- **Eventually.** Became less popular, due to several reasons
    - Vanishing gradient problem
    - Not zero-centered
    - Not memory-/computation-efficient

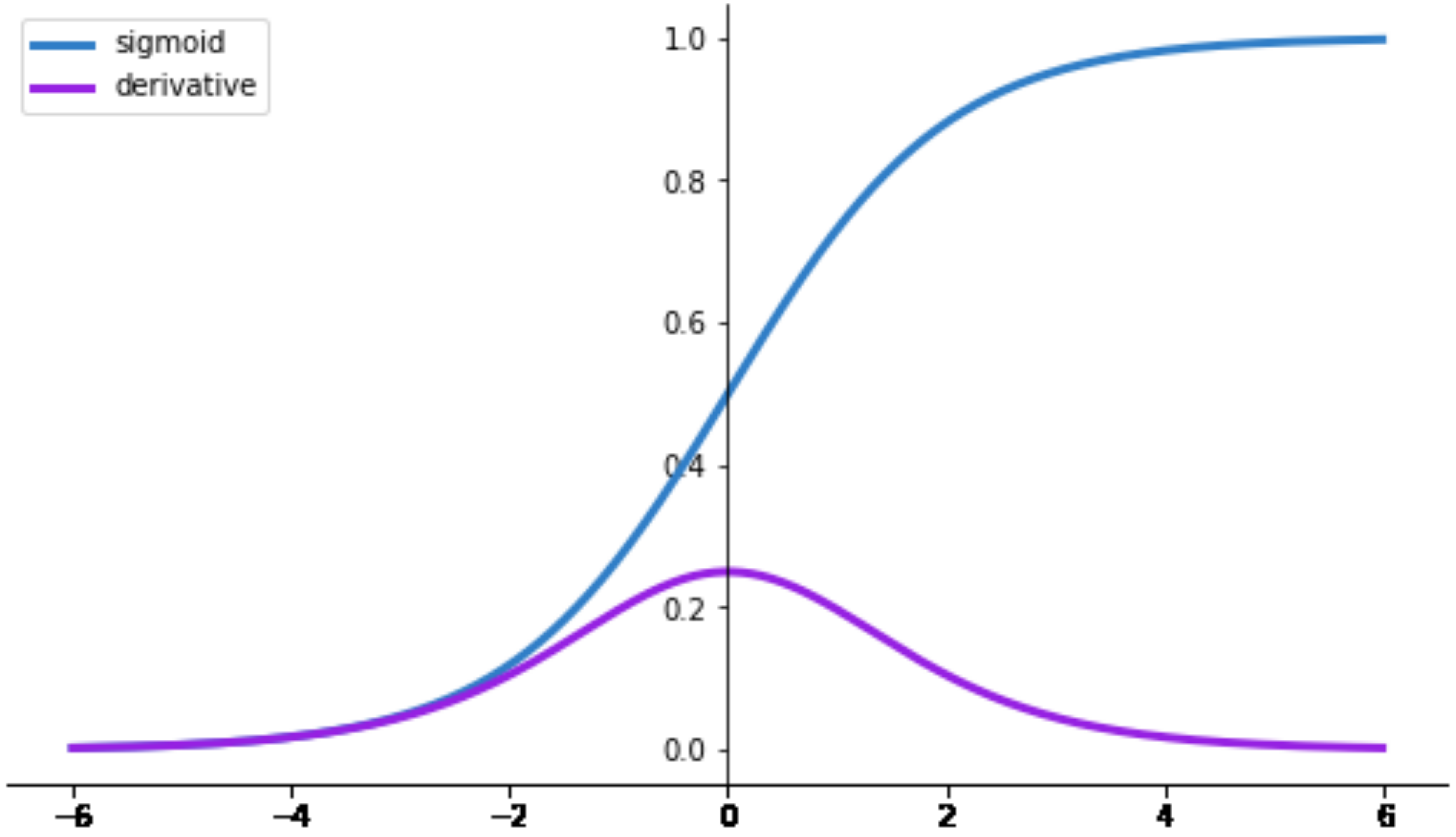
1. Vanishing gradients

- **Problem.** If we make networks deeper, sigmoids make the gradient vanish for certain layers!

1. Vanishing gradients

- **Problem.** If we make networks deeper, sigmoids make the gradient vanish for certain layers!

- **1-layer net.** Suppose that we have a predictor $f(x) = \sigma(wx)$
  - Gradient. $\nabla_w f(x) = \sigma'(wx) \cdot x$
    - Max scale: $x/4$ (mostly zero)

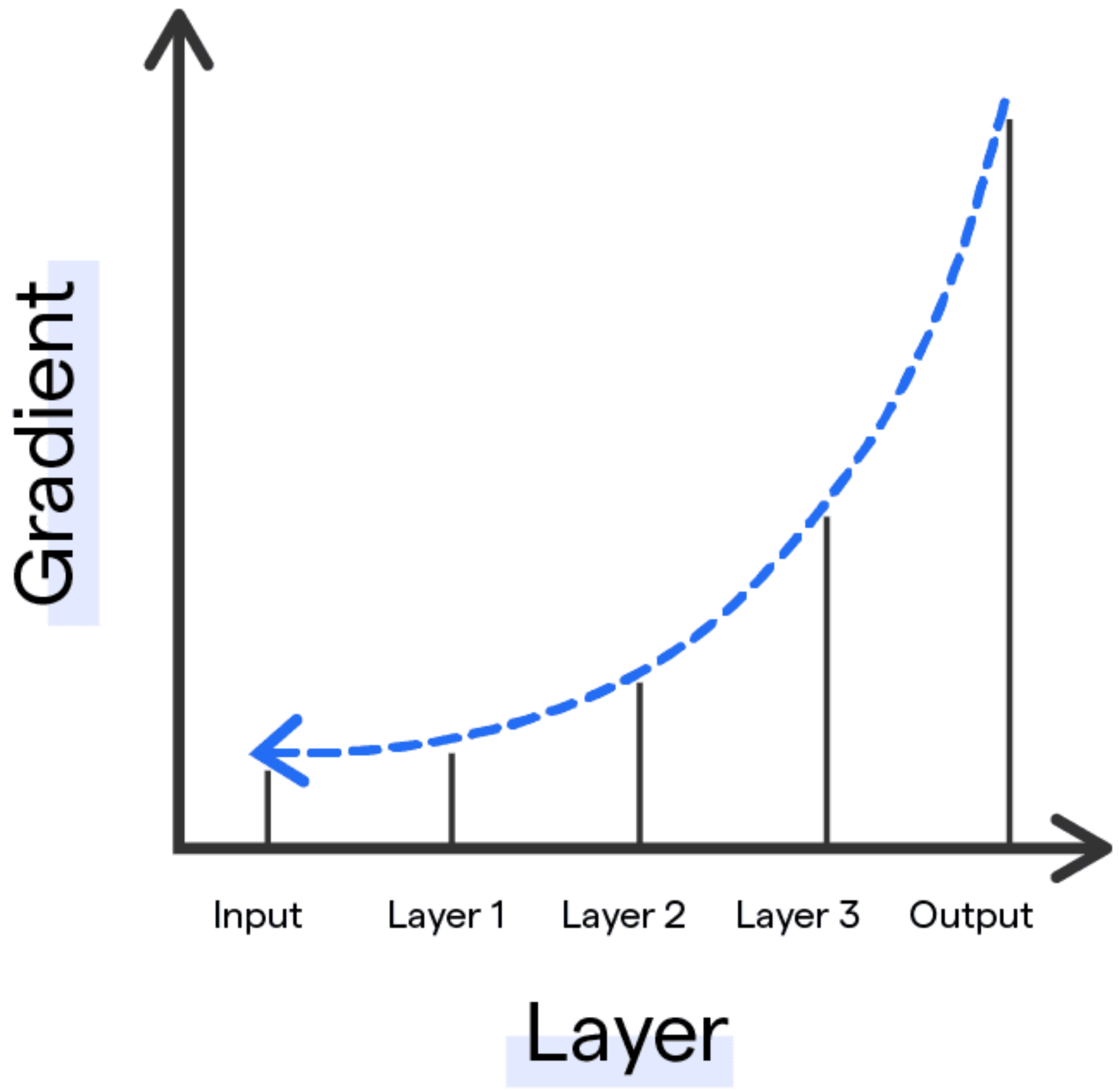


1. Vanishing gradients

- **Problem.** If we make networks deeper, sigmoids make the gradient vanish for certain layers!
- **1-layer net.** Suppose that we have a predictor $f(x) = \sigma(wx)$
  - Gradient. $\nabla_w f(x) = \sigma'(wx) \cdot x$
    - Max scale: $x/4$
- **Deep net.** Suppose that we have a predictor $f(x) = \sigma(w_L \cdot \sigma(\cdots \sigma(w_1 \cdot x) \cdots))$
  - 1st layer gradient. $\nabla_{w_1} f(x) = \sigma'(w_L \cdot z_L) \cdot \sigma'(w_{L-1} z_{L-1}) \cdot \cdots \cdot \sigma'(w_1 \cdot x) \cdot x$
    - Max scale: $x/4^L$
  - Lth layer gradient. $\nabla_{w_L} f(x) = \sigma'(w_L \cdot z_L) \cdot z_L$

1. Vanishing gradients

- This results in a severe imbalance in layer-wise gradient
  - The parameters in the early layers will not be utilized well



2. Not zero-centered

- **Problem.** Gradients of sigmoidal net is either all-positive or all-negative!

2. Not zero-centered

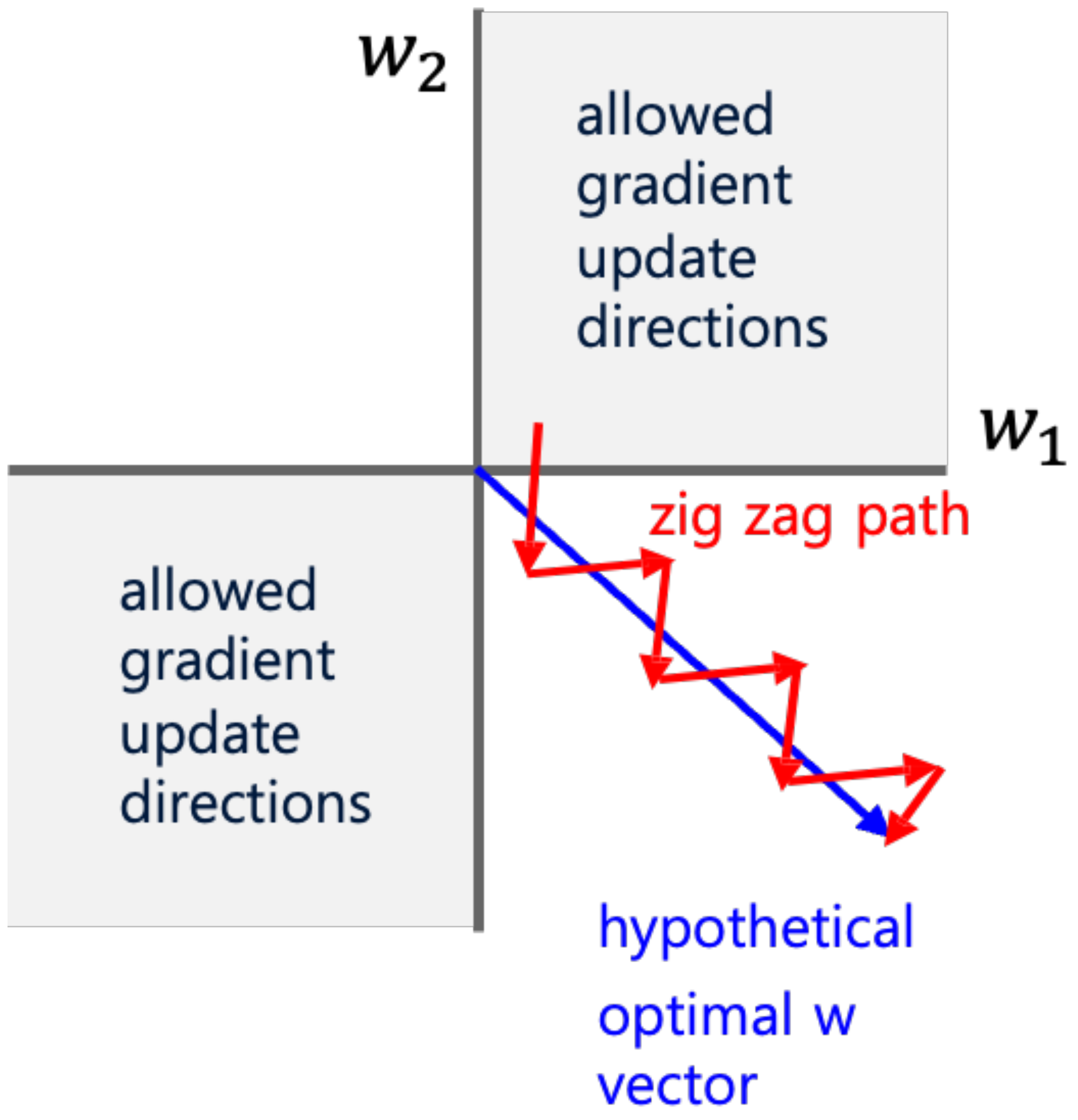
- **Problem.** Gradients of sigmoidal net is either all-positive or all-negative!
  - Consider a sigmoidal neuron $f(x) = \sigma\left(\mathbf{w}^\top \mathbf{x}\right)$
    - <u>Gradient for ith weight</u>. $\nabla_{w_i} f(\mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot x_i$

positive

positive, if also sigmoid outputs

2. Not zero-centered

- **Problem.** Gradients of sigmoidal net is either all-positive or all-negative!
  - Consider a sigmoidal neuron $f(x) = \sigma\left(\mathbf{w}^\top \mathbf{x}\right)$
    - Gradient for ith weight. $\nabla_{w_i} f(\mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot x_i$
    - If the loss derivative is positive —> all gradients are positive
    - If the loss derivative is negative —> all gradients are negative
      - Results in a suboptimal zig zag path
        (less problematic when we use multiple samples)
      - Can be mitigated if inputs $\{x_i\}$ are zero-centered



3. Efficiency

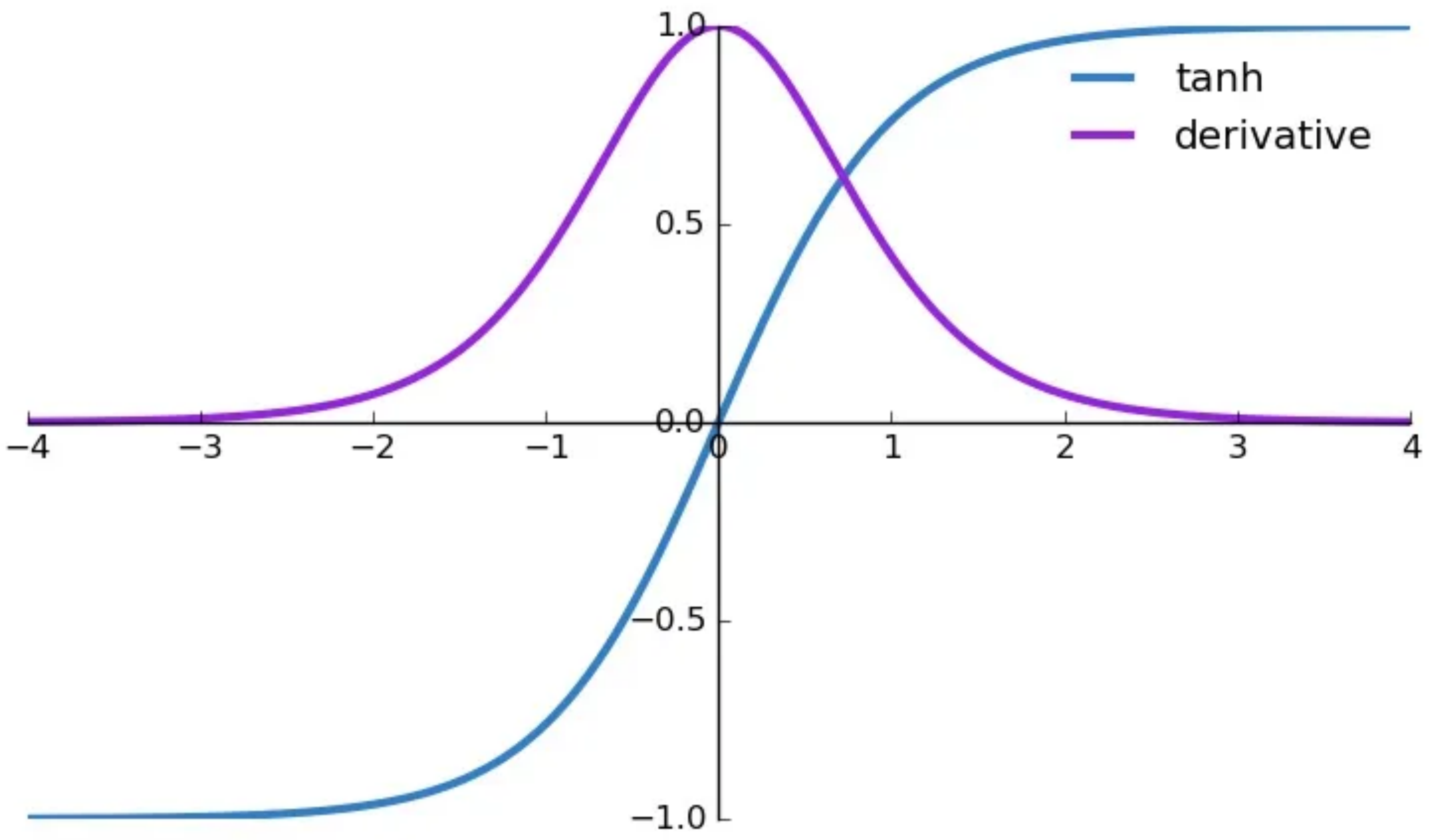
- **Inference.** Need to compute the function $\sigma(t) = 1/(1 + \exp(-t))$
    - Complicated to implement with hardwares…
        - Speedup by utilizing look-up tables

3. Efficiency

- **Inference.** Need to compute the function $\sigma(t) = 1/(1 + \exp(-t))$
  - Complicated to implement with hardwares…
    - Speedup by utilizing look-up tables
- **Training.** Need to compute the gradient $\sigma'(t) = \sigma(t)(1 - \sigma(t))$
  - Requires storing the $\sigma(t)$ computed during the forward phase
  - Requires floating point multiplications

Better activations

- Noticing this problem, alternative activations have been used.
  - **Tanh.** Zero-centered V
    Non-vanishing gradient X
    Computational efficiency X



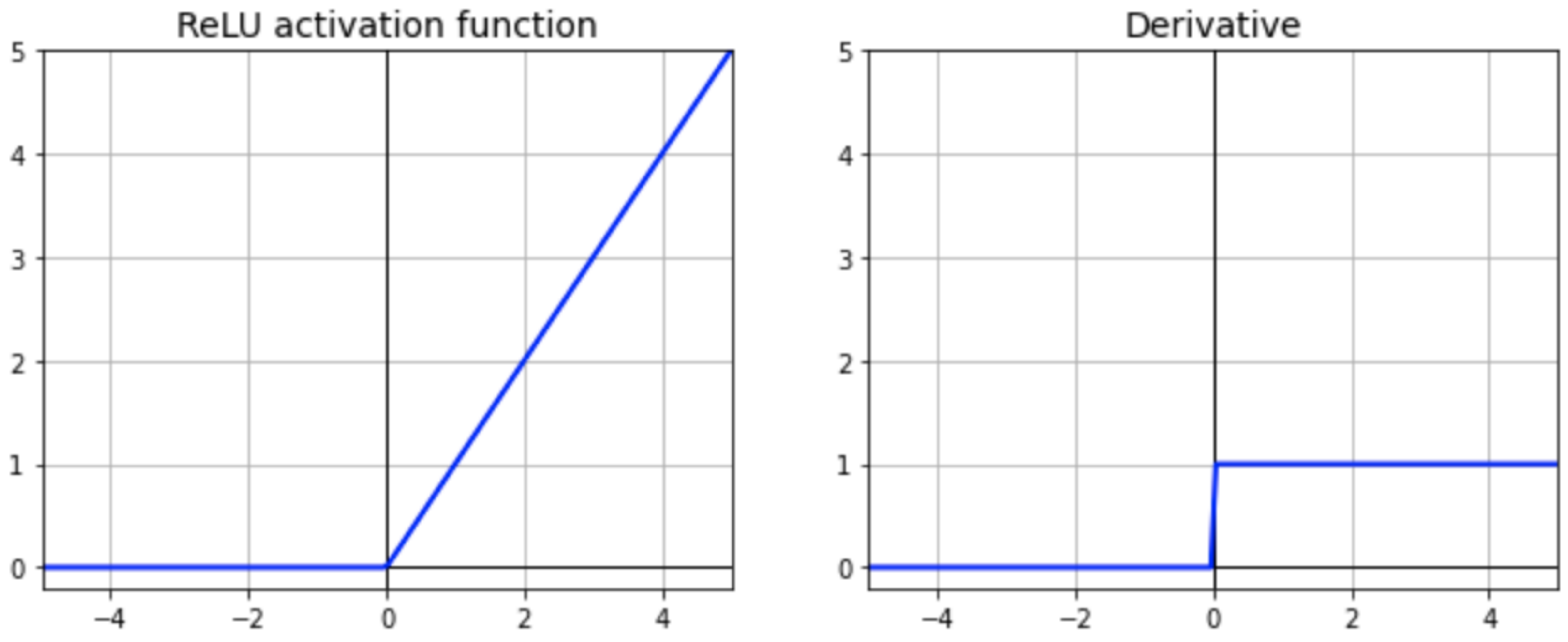
Better activations

- Noticing this problem, alternative activations have been used.

  - **Tanh.** Zero-centered **V**
    Non-vanishing gradient **X**
    Computational efficiency **X**

  - **ReLU.** Zero-centered **X**/**V**
    Non-vanishing gradient **?**
    Computational efficiency **V**
    - Converges faster in practice (e.g., 6x)
    - Can be made zero-centered via normalization (later)
    - Requires careful initialization to avoid vanishing gradients (later)

    - Sadly, experiences dead neuron



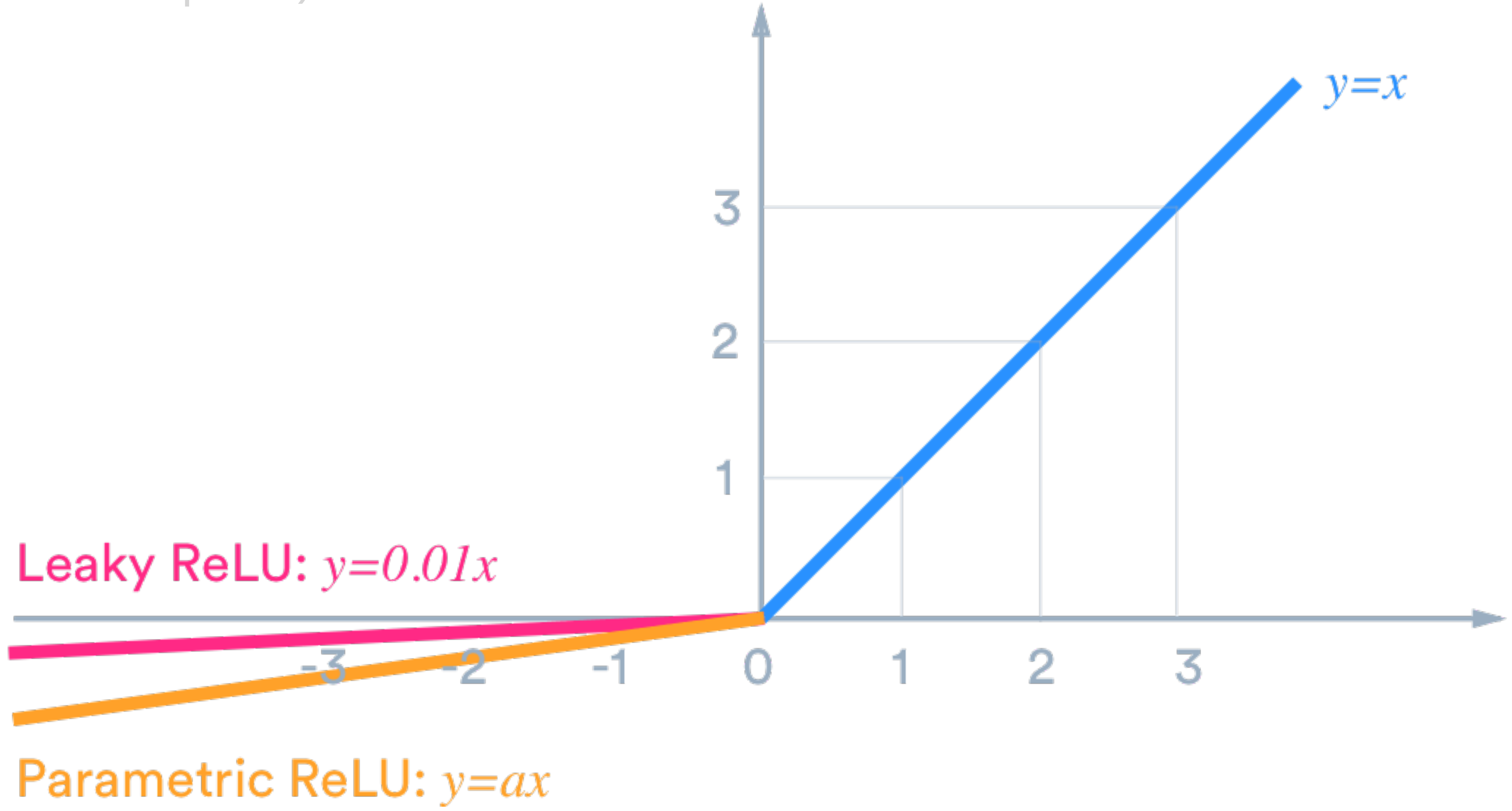
Dying ReLU

- **Problem.** Some neurons never activate!
  - Suppose that we have a ReLU neuron $\sigma(\mathbf{w}^\top \mathbf{x} + b)$
    - <u>Gradient for ith connection</u>. $\mathbf{1}[\mathbf{w}^\top \mathbf{x} + b \geq 0] \cdot x_i$
      - Zero if $\mathbf{w}^\top \mathbf{x} < -b$ for most $\mathbf{x}$

        (e.g., most weights are negative and $\mathbf{x}$ is a ReLU output)

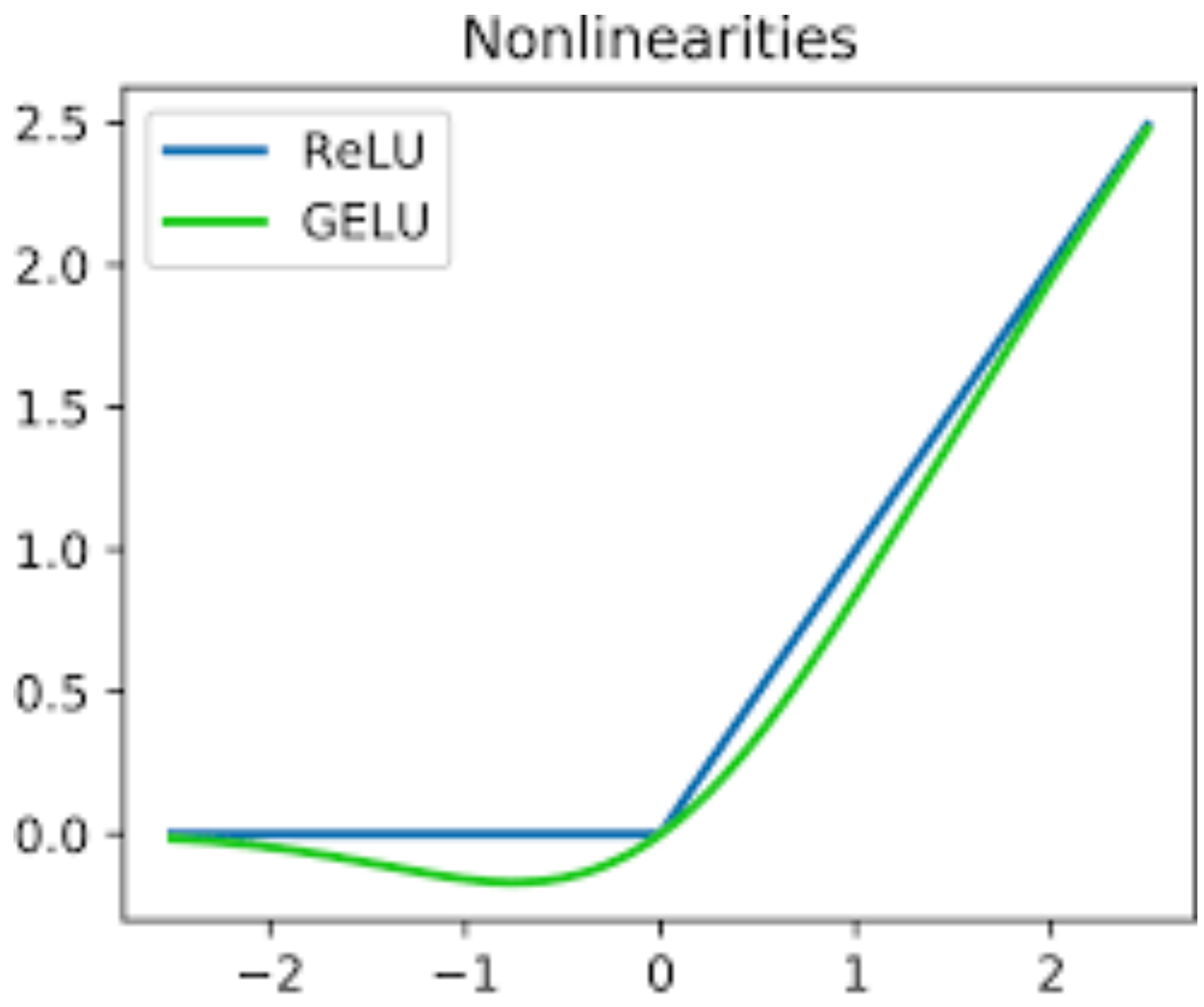
Dying ReLU

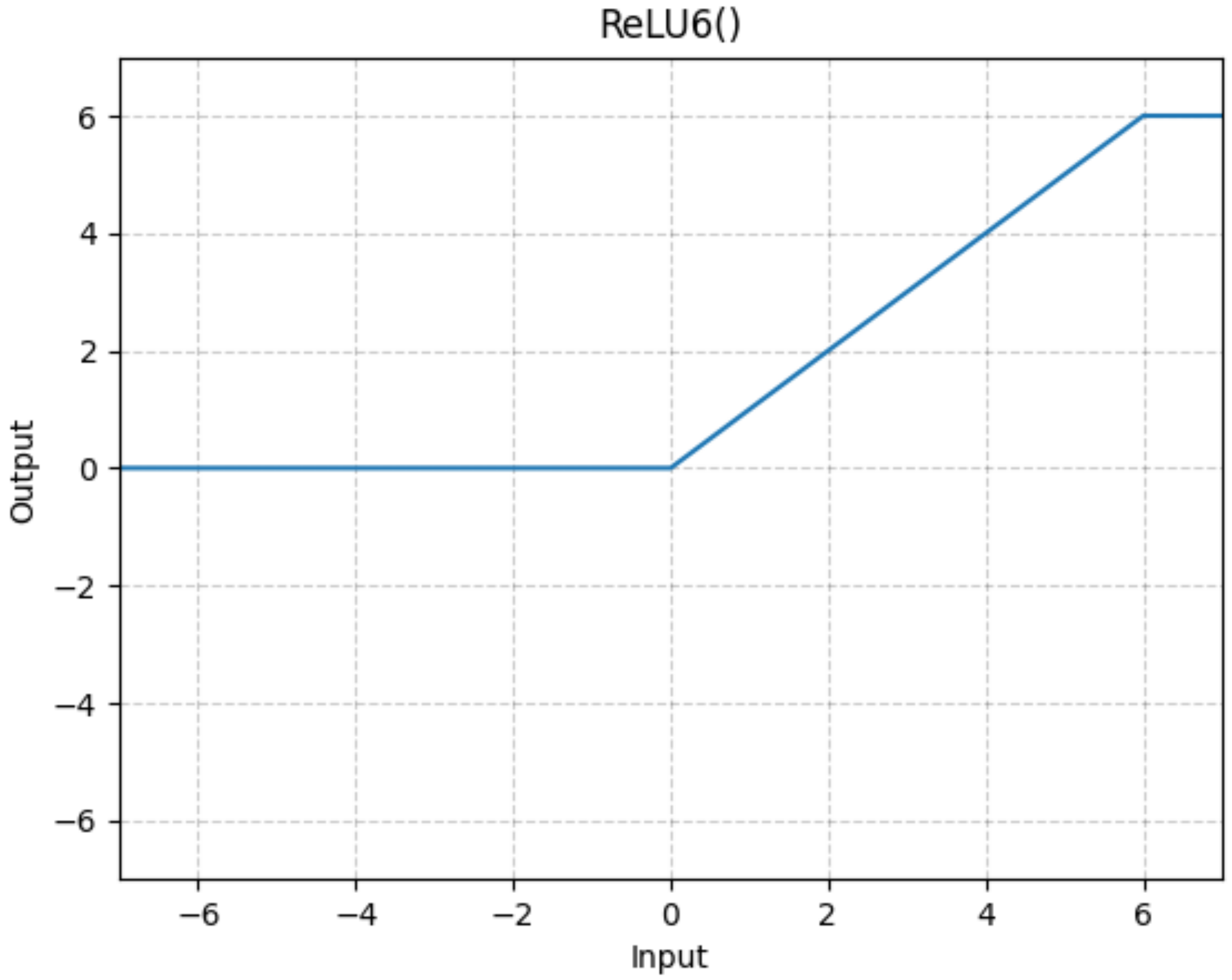
- **Problem.** Some neurons never activate!
    - Suppose that we have a ReLU neuron $\sigma(\mathbf{w}^\top \mathbf{x} + b)$
        - Gradient for ith connection. $\mathbf{1}[\mathbf{w}^\top \mathbf{x} + b \geq 0] \cdot x_i$
            - Zero if $\mathbf{w}^\top \mathbf{x} < -b$ for most $\mathbf{x}$
              (e.g., most weights are negative and $\mathbf{x}$ is a ReLU output)
- **Solution.**
    - Initialize the bias as a small-but-positive value.
    - Use "leaky" ReLU / ELU / ...



Modern choices

- Practitioners training giant models love GeLU / Swish / ...
- Quantization people love ReLU6
- **Recommendation.** Try ReLU as a default, and try these for squeezing out max performance.





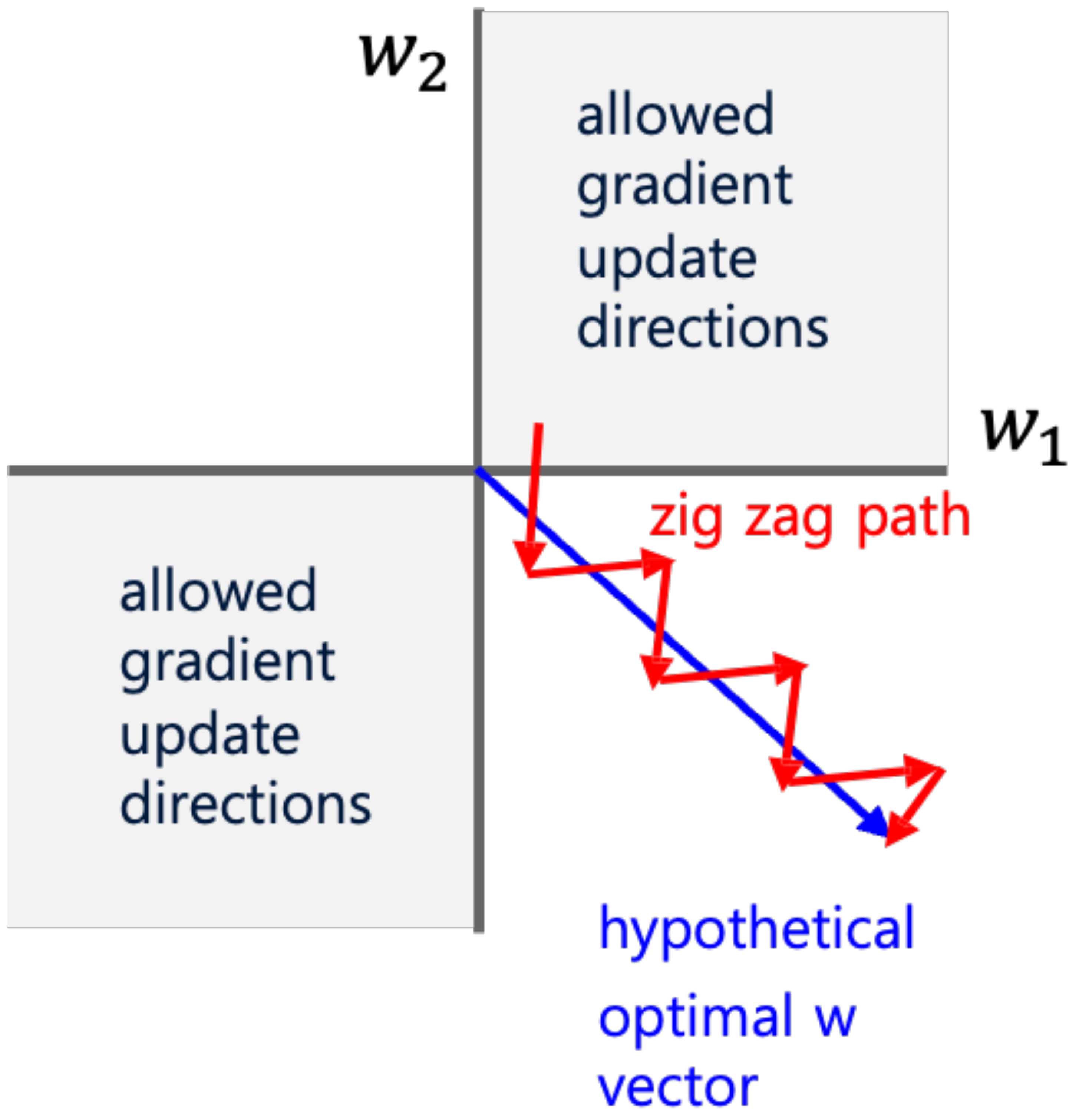
Data preprocessing

Data preprocessing

- Recall that zig zag path happened when the neuron input is all-positive
  - Gradient for ith weight. $\nabla_{w_i} f(\mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot x_i$

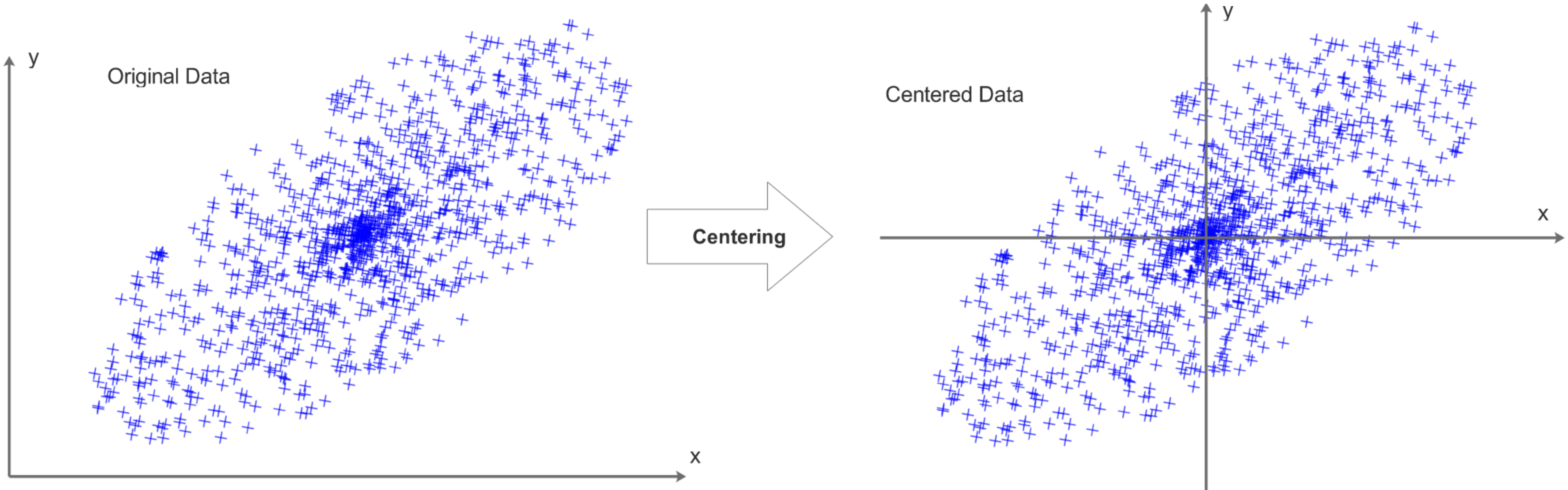
positive

positive, if also sigmoid outputs



Data preprocessing

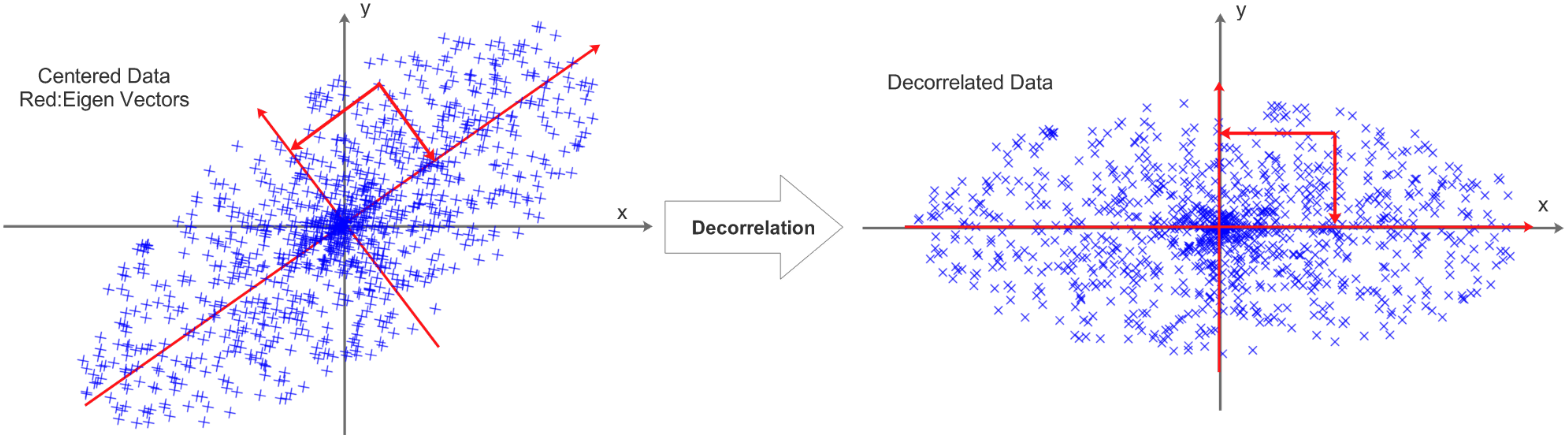
- Recall that zig zag path happened when the neuron input is all-positive
  - Gradient for ith weight. $\nabla_{w_i} f(\mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot x_i$
- **Idea.** Force data to have different signs.
  - Centering. Makes the data to have a zero-mean.



Data preprocessing

- **Also common.** For classic ML, it is typical to do:

  - Decorrelation. Make the axes have no correlation
    (not an idea that works for all data, e.g., image)

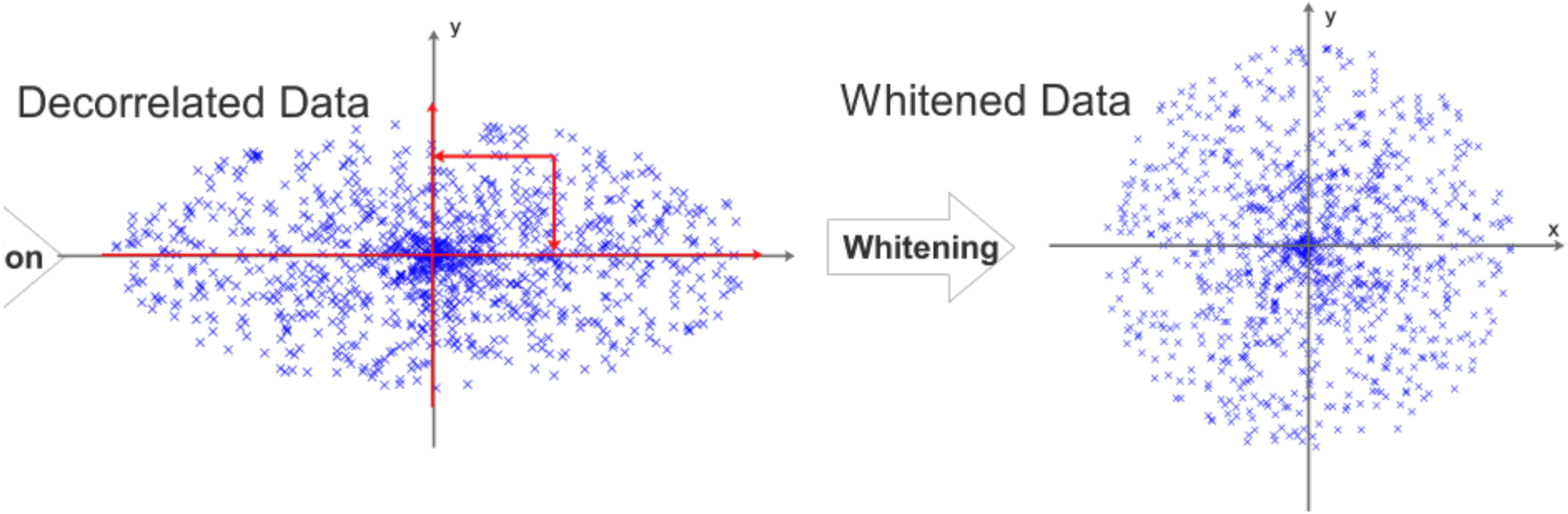


Data preprocessing

- **Also common.** For classic ML, it is typical to do:

  - <u>Decorrelation</u>. Make the axes have no correlation
    (not an idea that works for all data, e.g., image)

  - <u>Whitening</u>. Make each dim. have unit variance or range; avoids being biased by data scale
    (advanced: provably better convergence of GD)



Remarks

- In some cases, we also perform dimensionality reduction (e.g., PCA)
    - Not a good idea in general for DL

Remarks

- In some cases, we also perform dimensionality reduction (e.g., PCA)
    - Not a good idea in general for DL
- In many practical cases (e.g., images), we only perform the centering operation
    - For CIFAR-10 data:
        - AlexNet: Subtract the mean image ([32,32,3] tensor)
        - VGG: Subtract the mean along RGB channels (i.e., 3-dimensional value)

Remarks

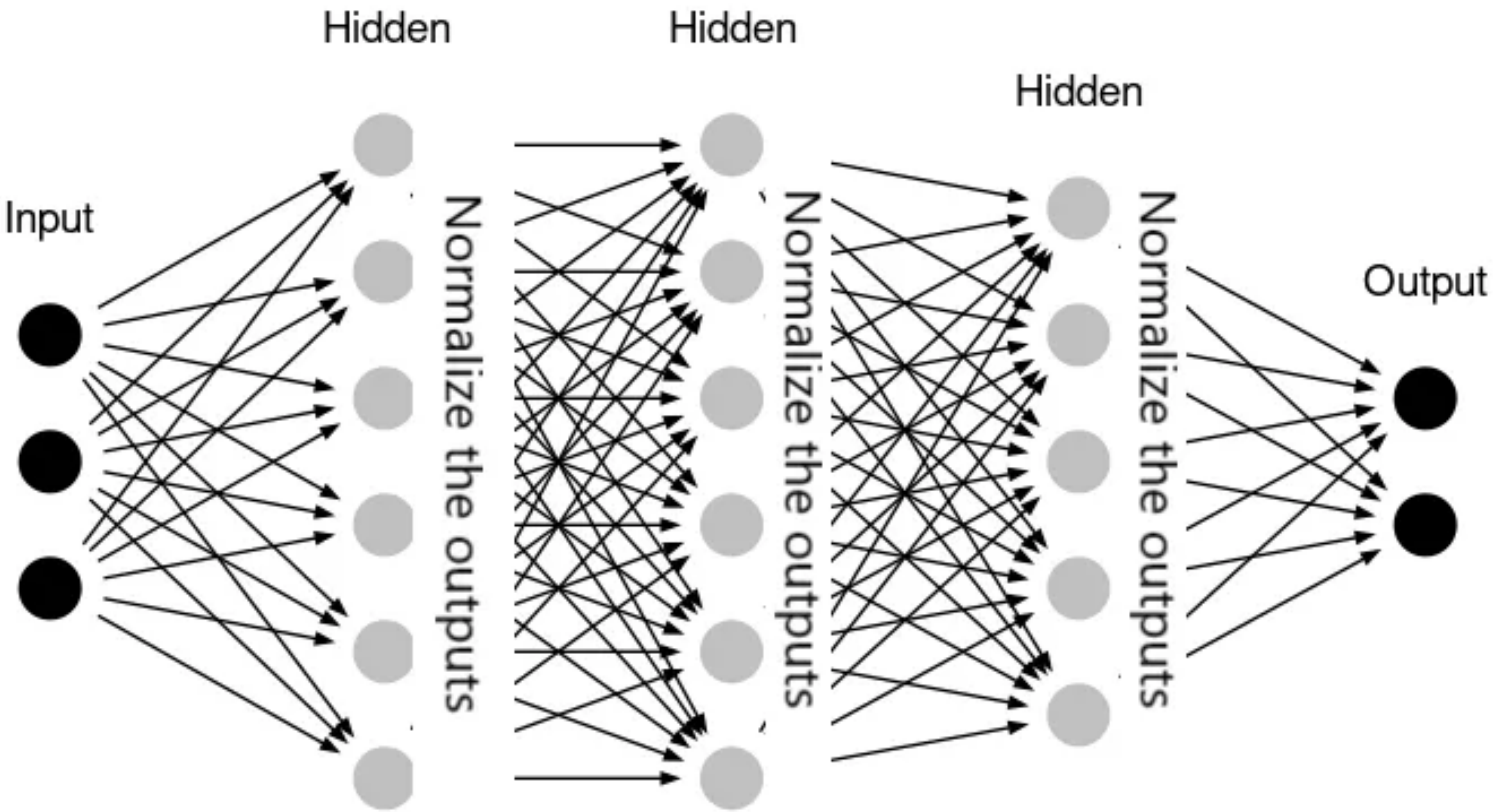
- In some cases, we also perform dimensionality reduction (e.g., PCA)
  - Not a good idea in general for DL
- In many practical cases (e.g., images), we only perform the centering operation
  - For CIFAR-10 data:
    - AlexNet: Subtract the mean image ([32,32,3] tensor)
    - VGG: Subtract the mean along RGB channels (i.e., 3-dimensional value)
- It is common to perform centering & whitening occationally in hidden layers
  - Many different ways to do it; Batch / Layer

Normalization layers

Normalization layers

- **Idea.** Perform centering + scaling in intermediate layers

zero-mean unit variance



Normalization layers

- **Idea.** Perform centering + scaling in intermediate layers

- Batch Norm. Consider a batch of activations at some layer $\mathbf{z}_1, \ldots, \mathbf{z}_B$

    - Here, B is the batch size

    - Each activation has d channels.

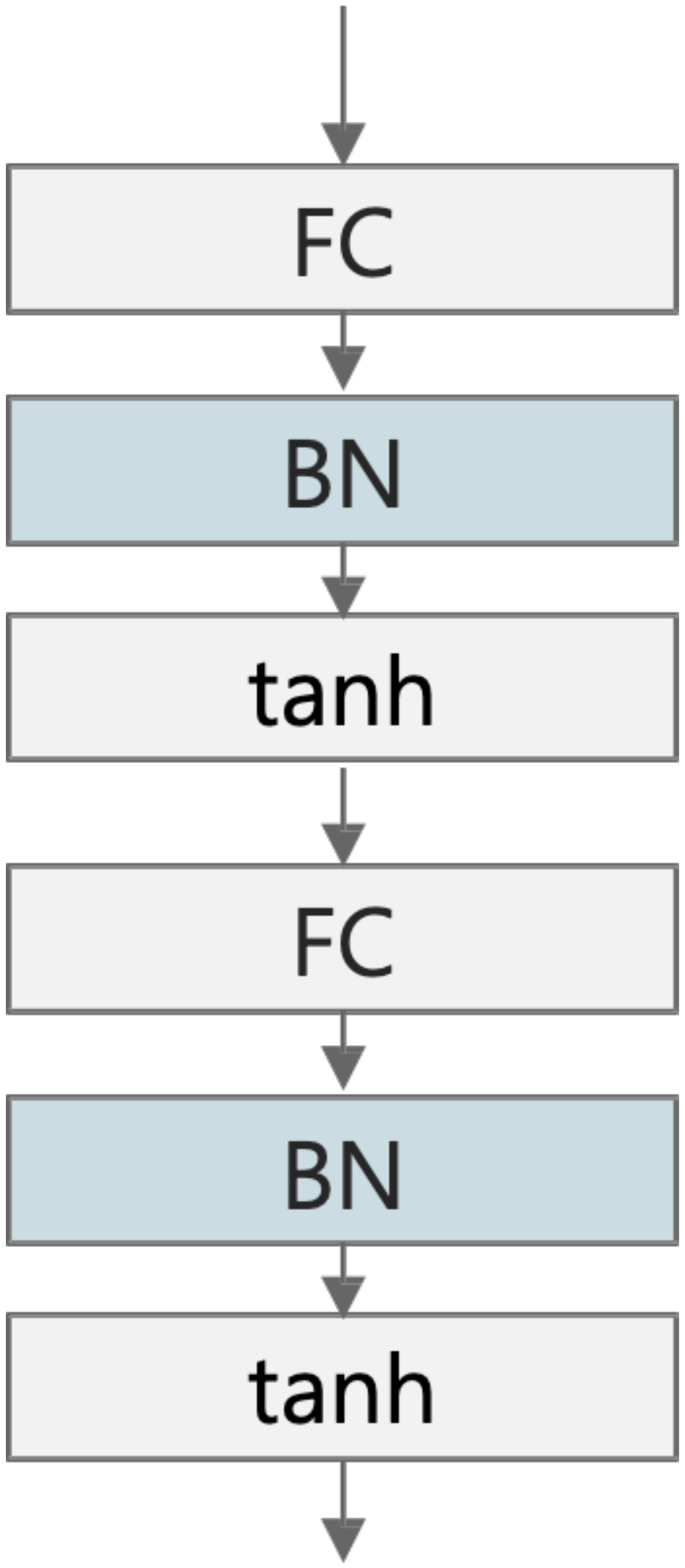
  - For each dimension, apply

$$\hat{\mathbf{z}}^{(j)} = \frac{\hat{\mathbf{z}}^{(j)} - \mathbb{E}[\mathbf{z}^{(j)}]}{\sqrt{\mathbf{Var}(\mathbf{z}^{(j)})}}$$

  - This is a differentiable function, so we can have it as a module in neural network

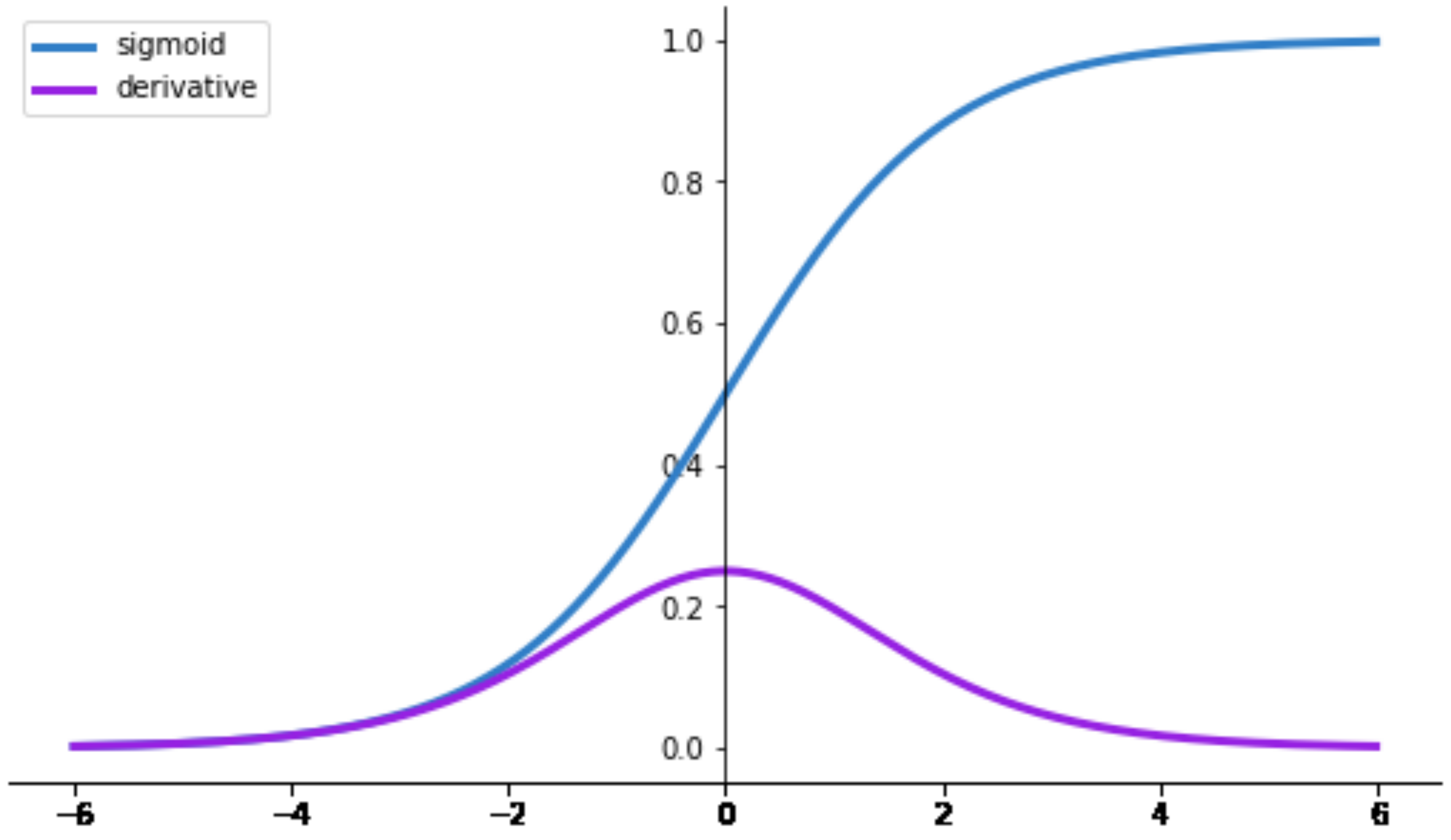
Considerations

- **Where to add?** Mostly placed at…
  - After each linear operations, e.g., linear / convolution
  - Before activation function



Considerations

- Mostly placed at...
  - After each linear operations, e.g., linear / convolution
  - Before activation function
- **Problem.** It may be harmful to place BN at all layers
  - Normalizing pre-sigmoids puts it in a linear region



Considerations

- Mostly placed at...
  - After each linear operations, e.g., linear / convolution
  - Before activation function
- **Caution.** It may be harmful to place BN at all layers
  - Normalizing pre-sigmoids puts it in a linear region
  - Solution. Add a trainable linear layer:

$$\hat{\mathbf{y}}^{(j)} = \gamma^{(j)}\hat{\mathbf{x}}^{(j)} + \beta^{(j)}, \qquad j \in [d]$$

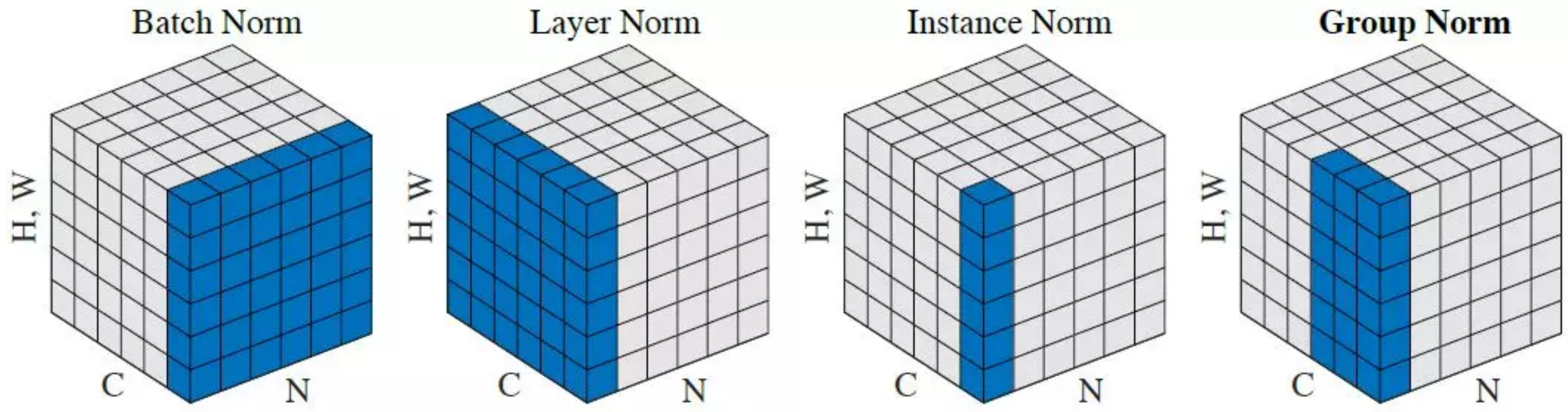
    - Allows us to scale back to negate the batch norm, whenever needed.

Considerations

- **Problem.** At the test time (i.e., inference phase), we don't take data as a batch!
  - Solution. Take a running average of the mean & variance during training.
    Use these values at the test time!
    - These can be merged into linear layers for a speedup.

Considerations

- **Problem.** At the test time (i.e., inference phase), we don't take data as a batch!
  - Solution. Take a running average of the mean & variance during training.
    Use these values at the test time!
    - These can be merged into linear layers for a speedup.
- **Problem.** Often, there are undesired side effects
  - e.g., training instability, sensitive to distribution shift & batch size…
  - Solution. Many variants



Advantages

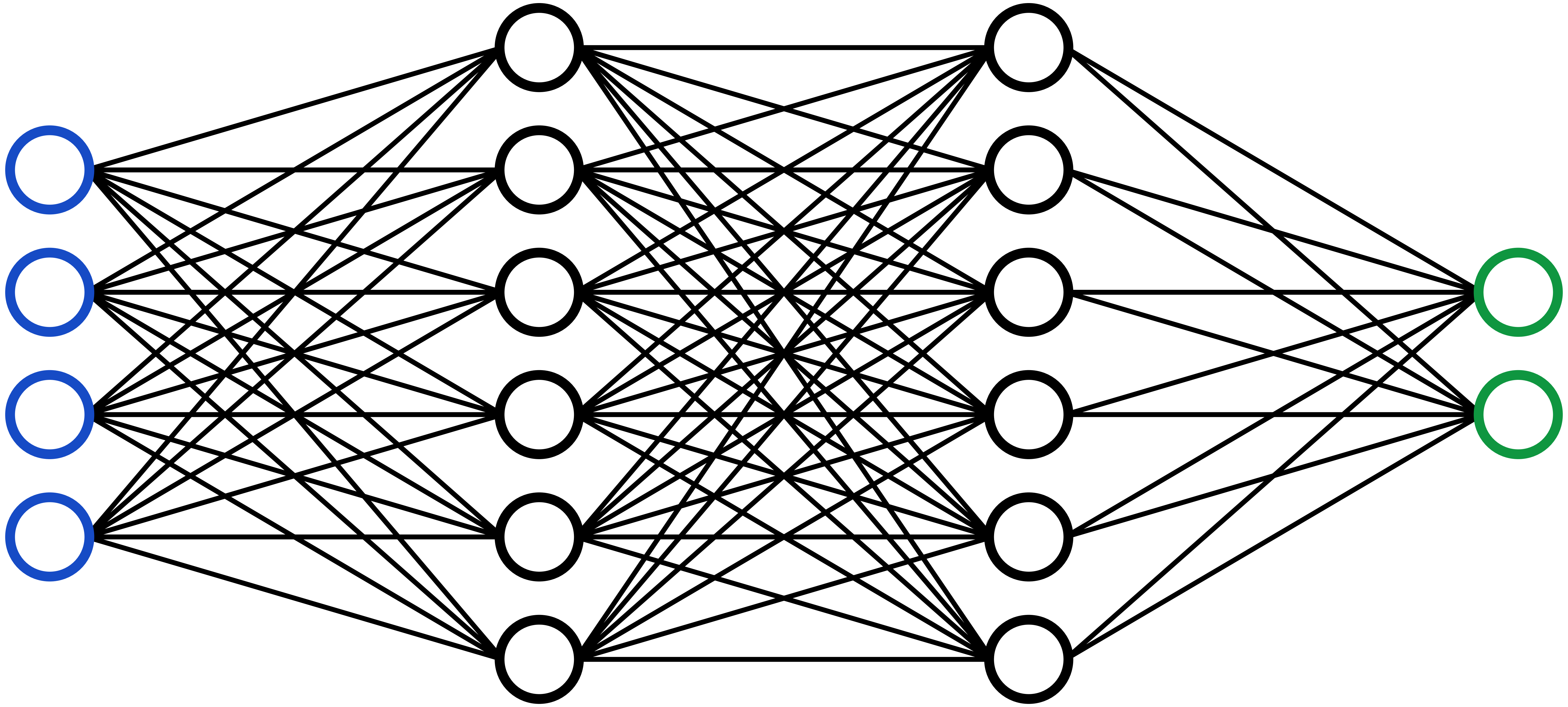
- In most cases, the pros outweigh the cons
    - Improves the gradient flow during training
    - Allows higher learning rate
    - Reduces the initialization sensitivity…

Parameter initializations

Initializing the weight parameters

- SGD-based optimization of NN parameters is also sensitive to initializations.
  (similar to most iterative optimizations)

- **Question.** What happens if all weights are initialized as the same constant?

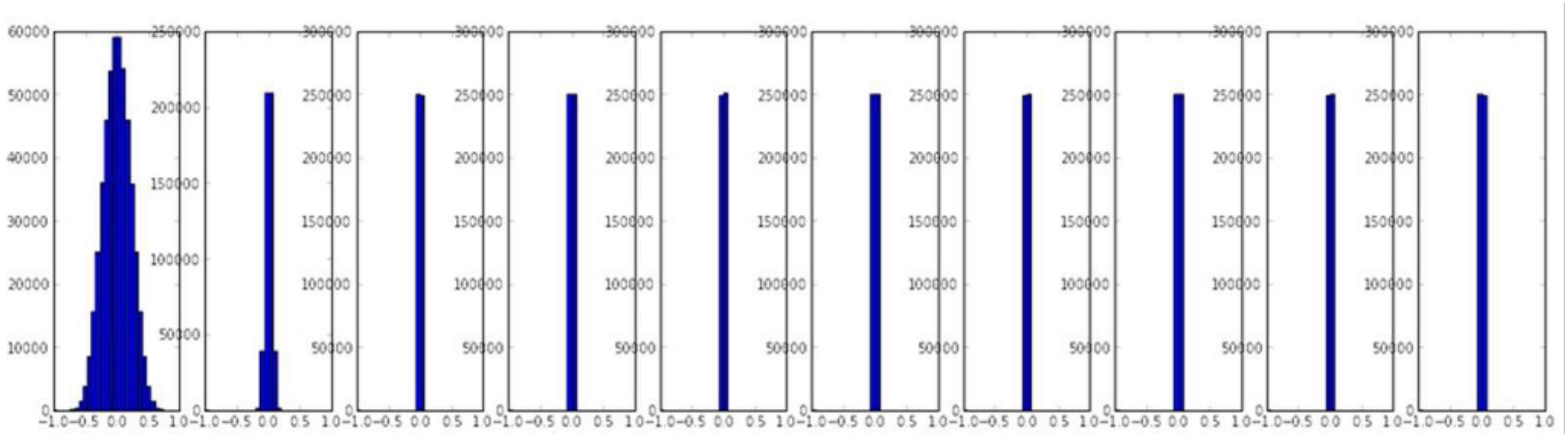


Random initialization

- **Idea.** Randomly initialize all weights, with $w \sim N(0, a^2)$
  - Works reasonably well for shallow nets.

Random initialization

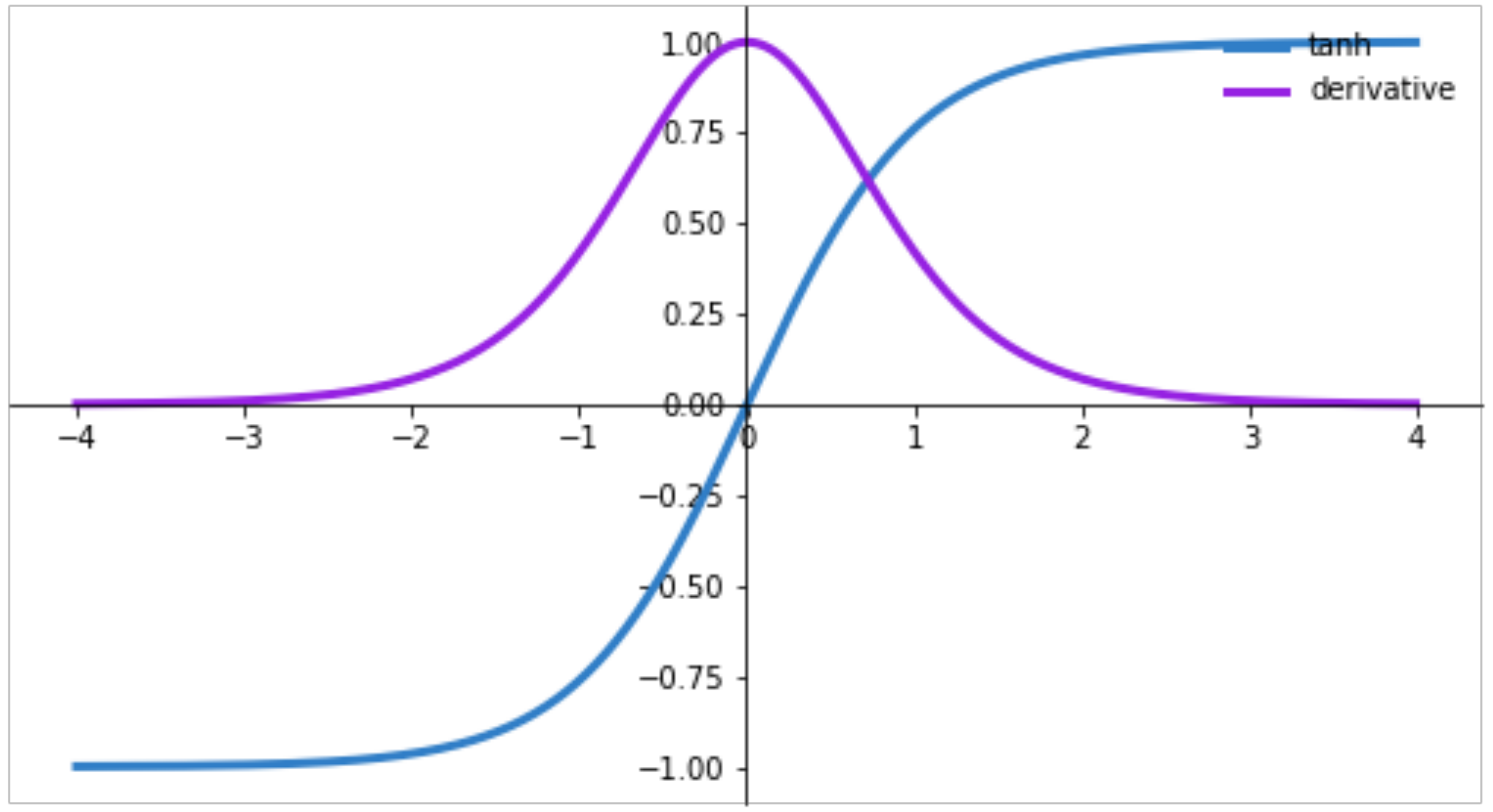
- **Idea.** Randomly initialize all weights, with $w \sim N(0,\sigma^2)$
  - Works reasonably well for shallow nets.
- **Problem.** For deep nets, very sensitive to the scale a
  - Small a. The activation $\sigma(\mathbf{w}^\top \mathbf{x})$ becomes very small for deep layers.
    - Thus, gradient become small in deeper layers: $\nabla_w \sigma(\mathbf{w}^\top \mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot \mathbf{x}$

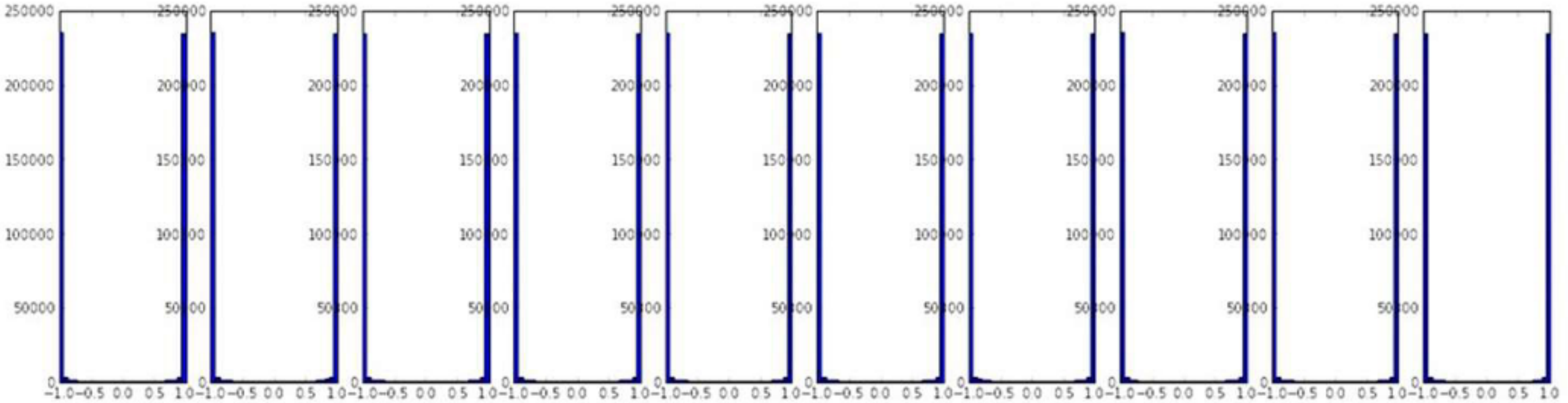


Random initialization

- <u>Large a</u>. Depends on the choice of activation functions
  - Tanh: Gradients become very small.
    - <u>Function output</u> $\sigma(a \cdot \mathbf{w}^\top \mathbf{x}) \to \{\pm 1\}$
    - <u>Gradients</u> $\nabla_w \sigma(\mathbf{w}^\top \mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot \mathbf{x}$

small if a is large





Random initialization

- Large a. Depends on the choice of activation functions
  - Tanh: Gradients become very small.
    - Function output $\sigma(a \cdot \mathbf{w}^\top \mathbf{x}) \to \{\pm 1\}$
    - Gradients $\nabla_w \sigma(\mathbf{w}^\top \mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot \mathbf{x}$
  - ReLU: Exploding gradients as the layer gets deeper
    - Function output $\sigma(a \cdot \mathbf{w}^\top \mathbf{x}) = a \cdot \sigma(\mathbf{w}^\top \mathbf{x})$
    - Gradients $\nabla_w \sigma(\mathbf{w}^\top \mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot \mathbf{x}$

accumulates the scaling factor a

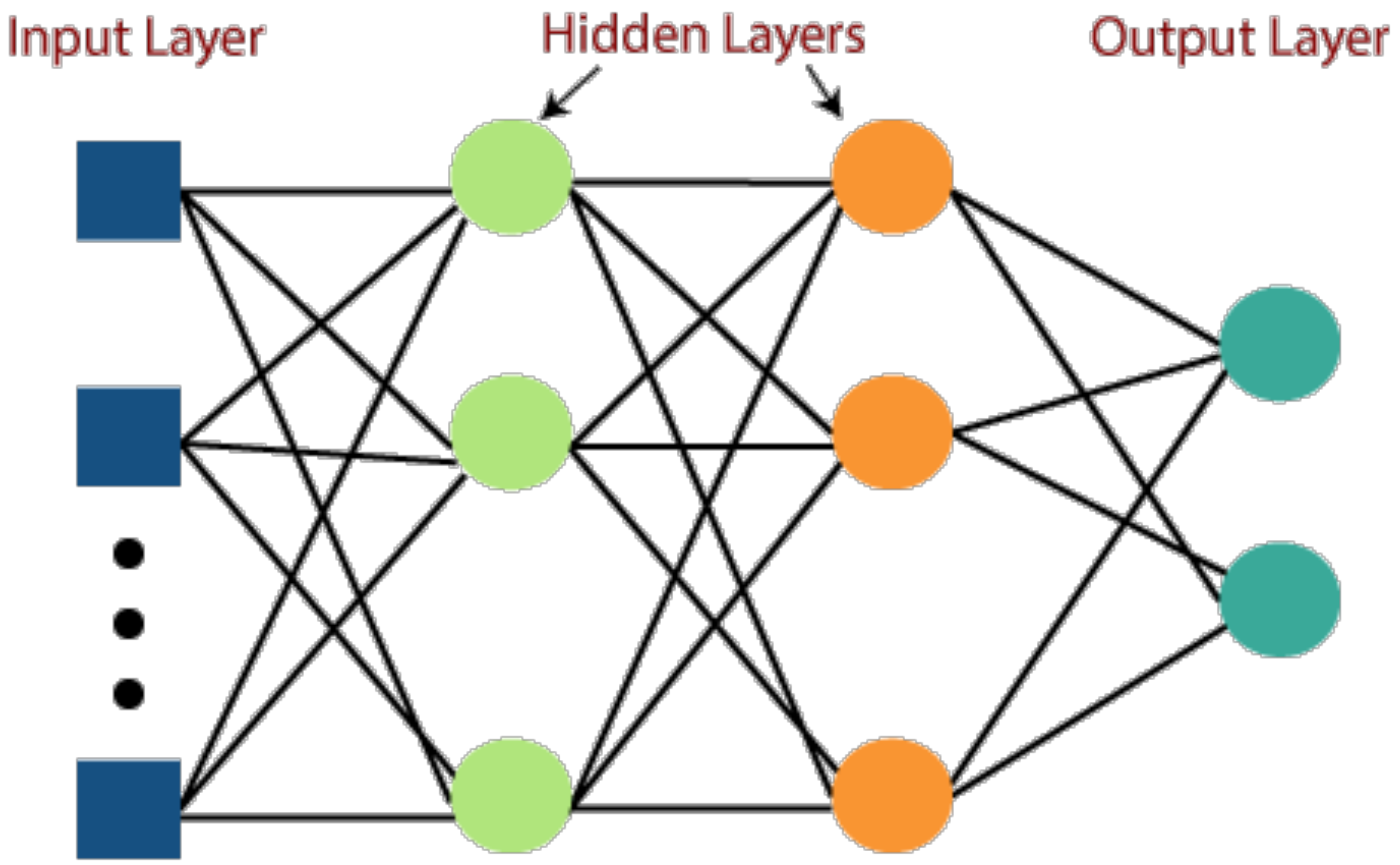
Weight-scaled initialization

- **Idea.** Choose a so that the activation variance remains constant over the layers
  - Xavier initialization (2010). Use the scaling factor

$$a = \sqrt{\frac{2}{(\text{input dim}) + (\text{output dim})}}$$

  - He initialization (2015). Use the scaling factor

$$a = \sqrt{\frac{2}{(\text{input dim})}}$$



- **Question.** Why $1/\sqrt{\text{neurons}}$?

Weight scales

- Suppose that we have a layer with d input & output neurons.
  - The input activation is $\mathbf{x} = (x_1, \ldots, x_d)$
  - The weight that connects ith input neuron to jth output neuron is w_{ij}
    - Drawn independently from $w_{ij} \sim N(0, a^2)$
- Goal. Make the activation scale similar: $\|\mathbf{x}\|^2 \approx \mathbb{E}\|\mathbf{W}\mathbf{x}\|^2$

Weight scales

- Suppose that we have a layer with d input & output neurons.
  - The input activation is $\mathbf{x} = (x_1, \dots, x_d)$
  - The weight that connects ith input neuron to jth output neuron is w_{ij}
    - Drawn independently from $w_{ij} \sim N(0, a^2)$
- Goal. Make the activation scale similar: $\|\mathbf{x}\|^2 \approx \mathbb{E}\|\mathbf{W}\mathbf{x}\|^2$
- Inspecting the weights connected to jth output neuron, we have

$$\mathbf{w}_j^\top \mathbf{x} \sim N\left(0, a^2 \|x\|^2\right)$$

- Then, we have

$$\mathbb{E}\|\mathbf{W}\mathbf{x}\|^2 = \sum_{j=1}^{d} \mathbb{E}(\mathbf{w}_j^\top \mathbf{x})^2 = d \cdot a^2 \cdot \|\mathbf{x}\|^2$$

should be 1, thus we have what we want

Remarks

- There are many research on how to initialize
  - Has been mostly okay with BNs, but BNs are getting faded away…
  - Many unmentioned works:
    - Orthogonal initialization
    - Identity initialization
    - Zero initialization

Cheers