Training neural networks - 1

Recap

- What deep neural networks are
- How to train deep neural networks
  - Algorithm: Stochastic Gradient Descent (SGD)

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

  - Trick: Backpropagation for efficient gradient evaluation
    - **Forward.** Compute intermediate activations & store in memory

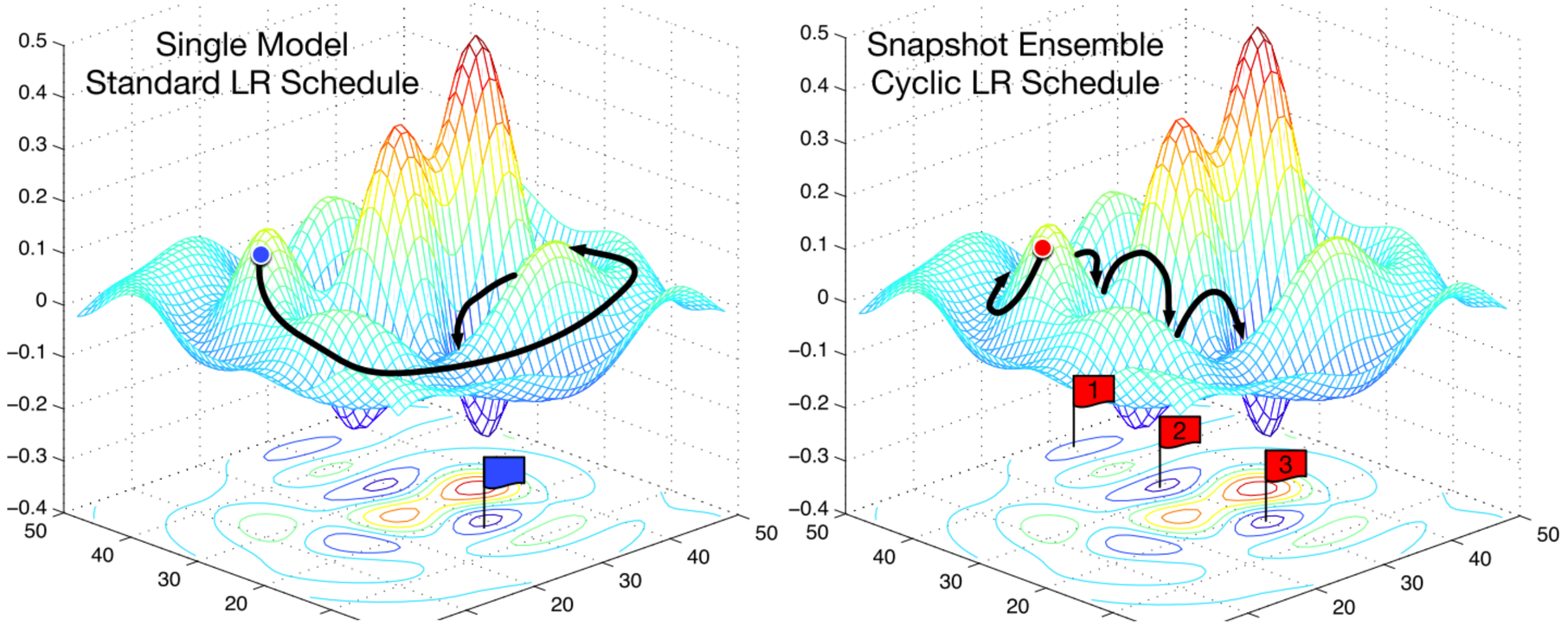
$$\mathbf{z} = f_1(\mathbf{x}; \mathbf{W}_1), \qquad f(\mathbf{x}) = f_2(\mathbf{z}; \mathbf{W}_2)$$

    - **Backward.** Combine modular derivative to compute the gradient

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

This week

- Neural net training is actually quite difficult:
  - Fail to converge to a solution with low training loss
  - Fail to converge to a well-generalizing solution
  - Even when converging, excessive time & computation needed



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

- Who is this document for?
- Why a tuning playbook?
- Guide for starting a new project
  - Choosing the model architecture
  - Choosing the optimizer

This week

- Fortunately, people tend to agree on basic principles

- **Today.** Setting up the training
  - Activation functions
  - Data Pre-processing
  - Normalization layers
  - Parameter initialization

- **Next class.** Tuning the optimization
  - Learning rate, Batch size, Regularizers, Optimizers, HP tuning, …

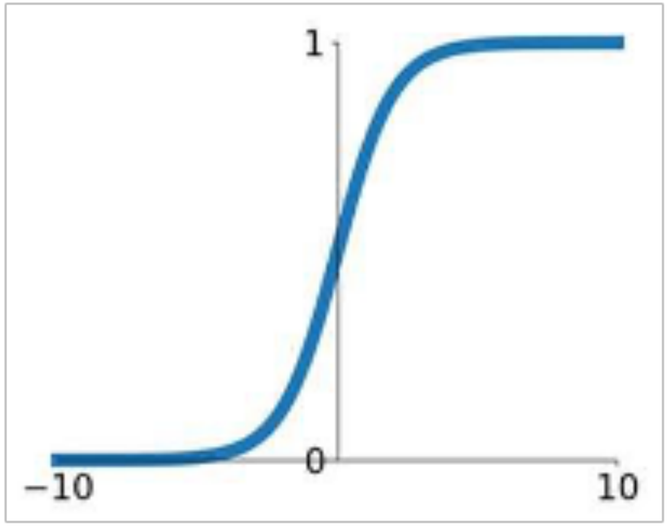
Activation functions

Fall of sigmoids

- In the past, sigmoidal activation functions were quite popular
  - Similar to $\mathbf{1}\{\cdot\}$ — quite accurate surrogate
  - Biological interpretation — firing rate of neurons
  - Easy to compute the gradient — $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$

Sigmoid

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



Fall of sigmoids

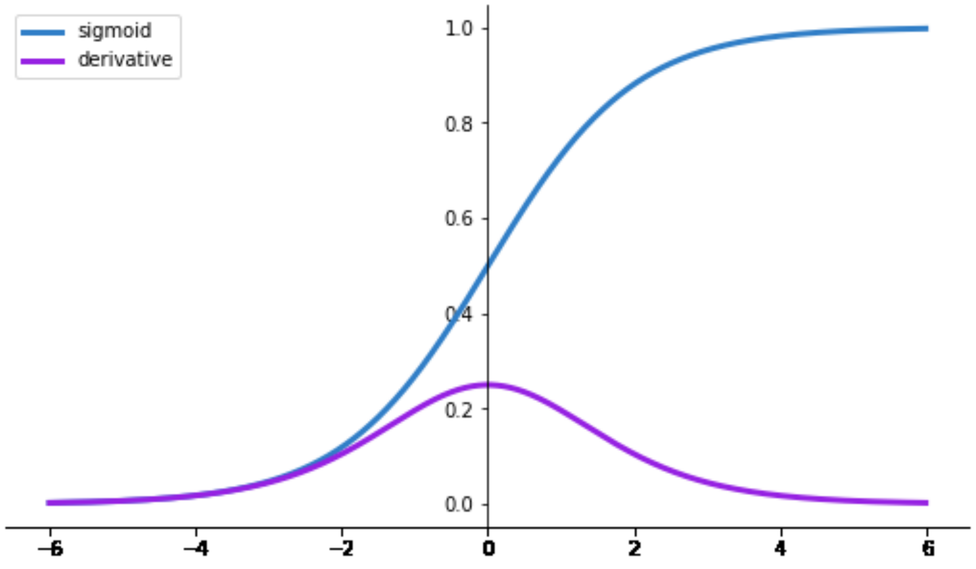
- However, these became less popular for deep learning
  - Vanishing gradient problem
  - Not zero-centered
  - Computational inefficiency

- Let’s take a look at each issue more carefully…

1. Vanishing gradients

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

- To see this, first consider a 1-layer net: $f(x) = \sigma(wx)$
  - The gradient will be $\nabla_w f(x) = \sigma'(wx) \cdot x$
    - The maximum scale will be $x/4$



1. Vanishing gradients

- What about deep nets?

$$f(x) = \sigma(w_L \cdot \sigma(\cdots \sigma(w_1 \cdot x) \cdots))$$

- The 1st layer gradient will be:

$$\nabla_{w_1} f(x) = \sigma'(w_L \cdot z_L) \cdot \sigma'(w_{L-1} z_{L-1}) \cdot \cdots \cdot \sigma'(w_1 \cdot x) \cdot x$$

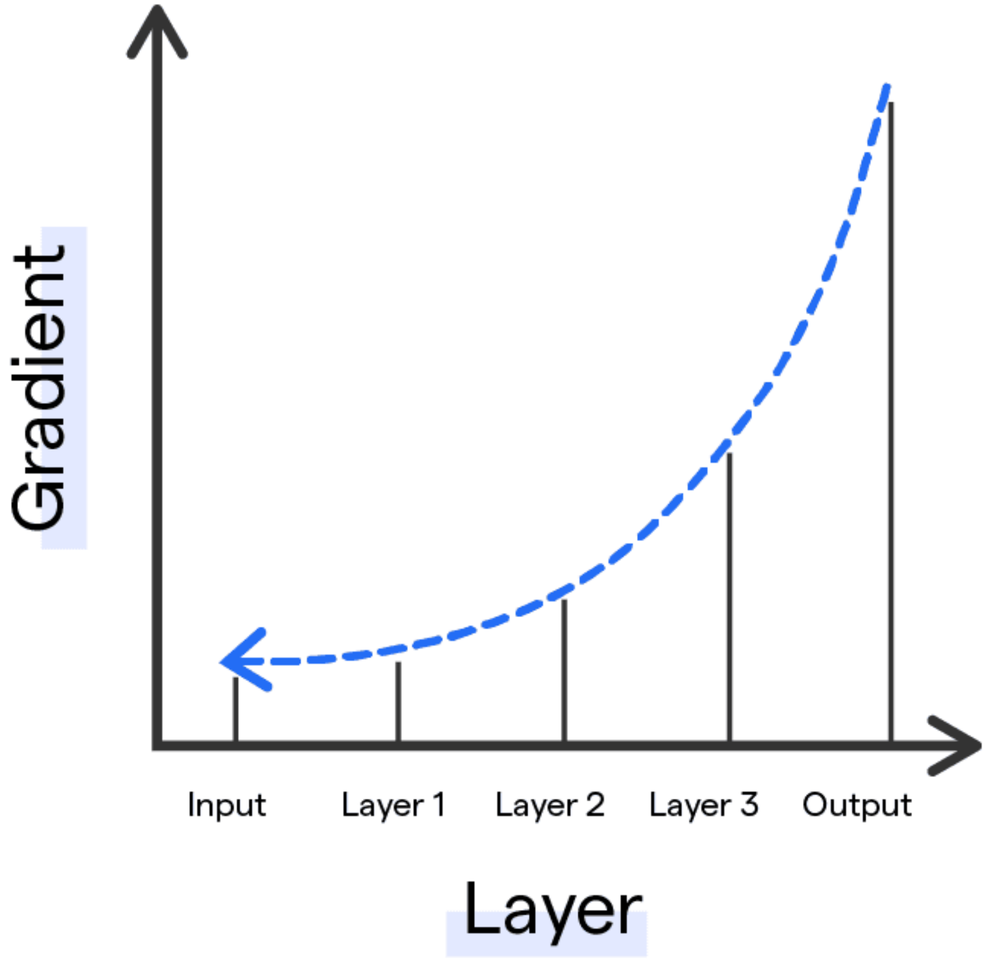
- Thus the maximum scale will be: $x/4^L$
  - If the network is deep, the 1st layer gradient is almost zero!

- On the other hand, the L-th layer gradient is:

$$\nabla_{w_L} f(x) = \sigma'(w_L \cdot z_L) \cdot z_L$$

1. Vanishing gradients

- This results in a severe imbalance in the layerwise gradients
  - Parameters in the early layers are not utilized well

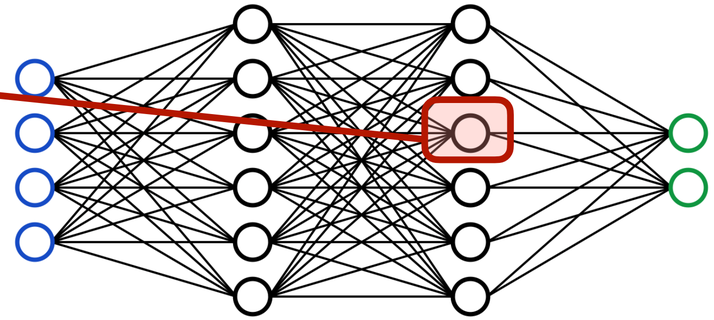


2. Not zero-centered

- **Problem.** Gradients of sigmoid nets are either all-positive or all-negative

- To see this, consider a sigmoid neuron in a deep network

$$f(x) = \sigma\left(\mathbf{w}^\top \mathbf{x}\right)$$



  - Gradient for the i-th weight will be:

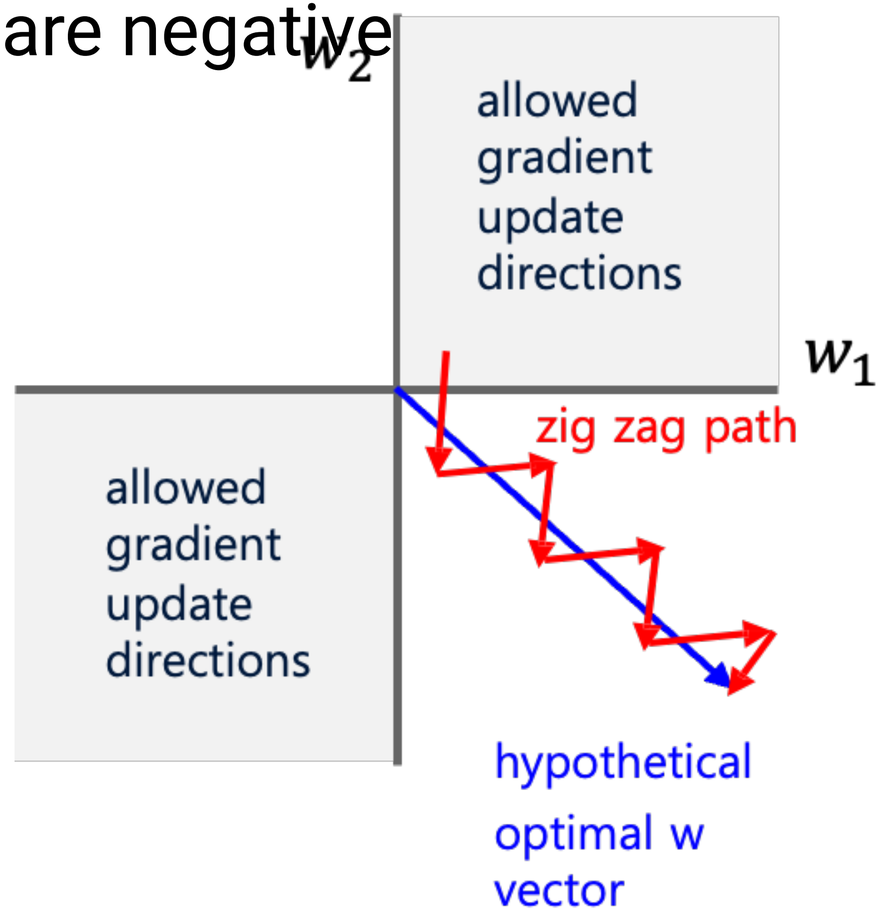
$$\nabla_{w_i} f(\mathbf{x}) = \underbrace{\sigma'(\mathbf{w}^\top \mathbf{x})}_{\text{positive}} \cdot \underbrace{x_i}_{\text{positive, if also sigmoid outputs}}$$

2. Not zero-centered

$$\nabla_{w_i} f(\mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot x_i$$

- 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 $\{x_i\}$ were zero-centered (not all-positive)



3. Efficiency

- **Problem.** Sigmoid is computationally tricky!
  - *Inference.* Need to compute the function $\sigma(t) = 1/(1 + \exp(-t))$
  - Complicated to implement with hardwares
    - Divide into intervals and approximate with polynomials
    - Use lookup tables

  - *Training*. Need to compute the derivative $\sigma'(t) = \sigma(t)(1 - \sigma(t))$
  - Requires an extra floating point multiplications

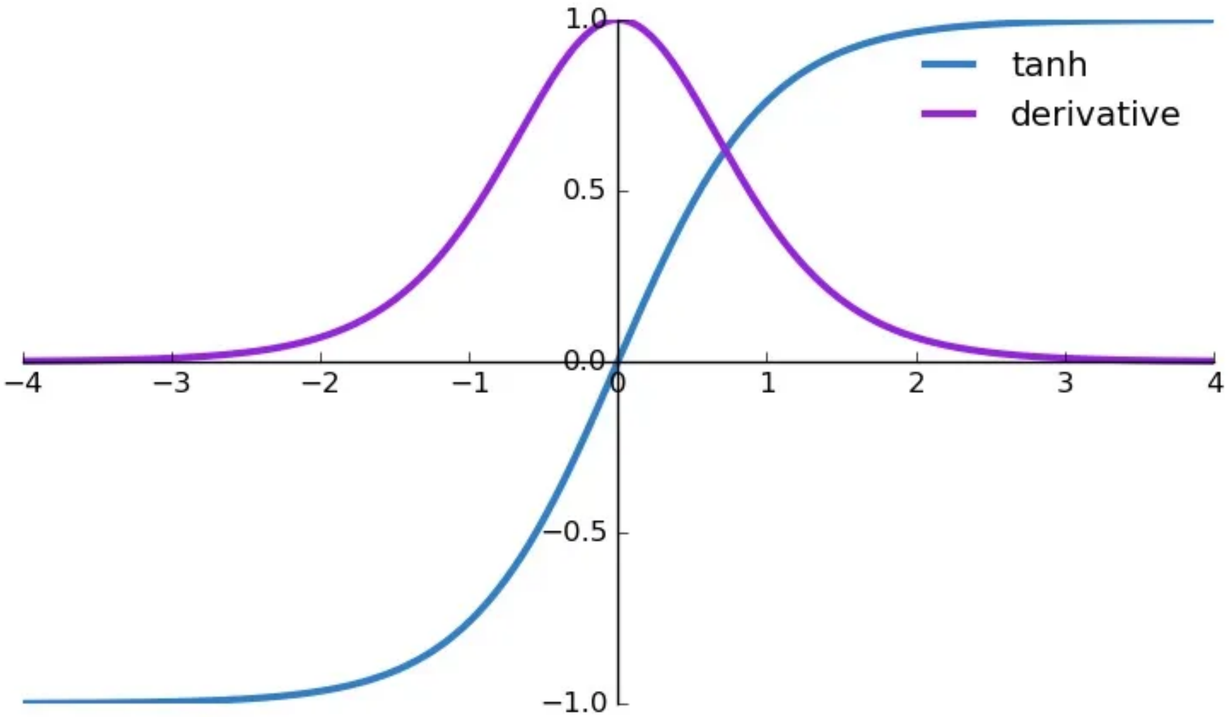
Better activations

- Noticing these problems, alternative activations have been used

- **Tanh.**

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

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



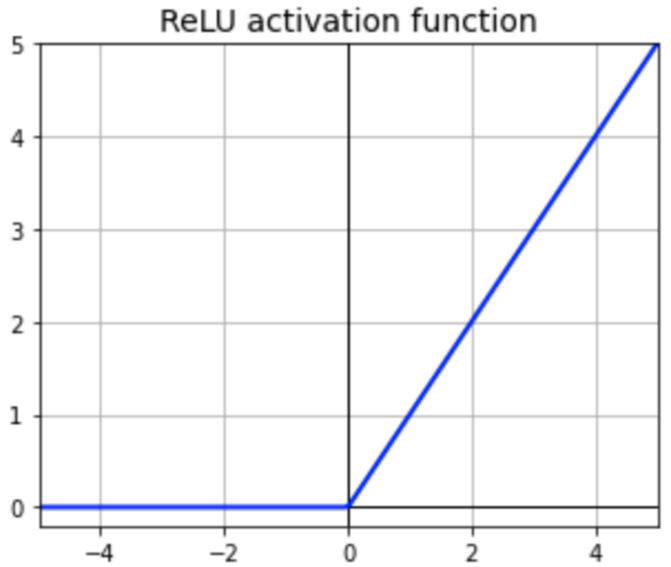
Better activations

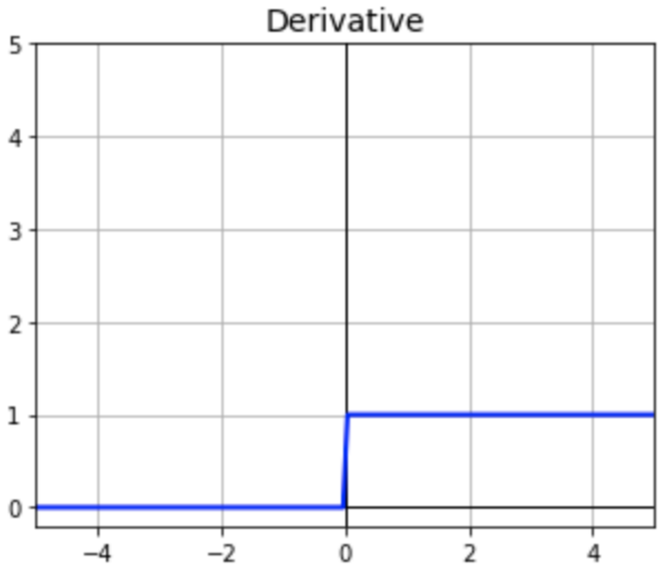
- **ReLU.**

$$f(x) = \max\{0, x\}$$

  - Zero-centered **X/V**
  - Non-vanishing gradient **X/V**
  - Computational efficiency **V**

  - Converges faster, empirically
  - Can be made zero-centered (later)
  - With careful initialization, can avoid vanishing gradients (later)





Dying ReLU

- Sadly, ReLU experiences dying ReLU problem
  - Some neurons never activate (i.e., have non-zero output)

- Suppose that we have a ReLU neuron

$$f(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

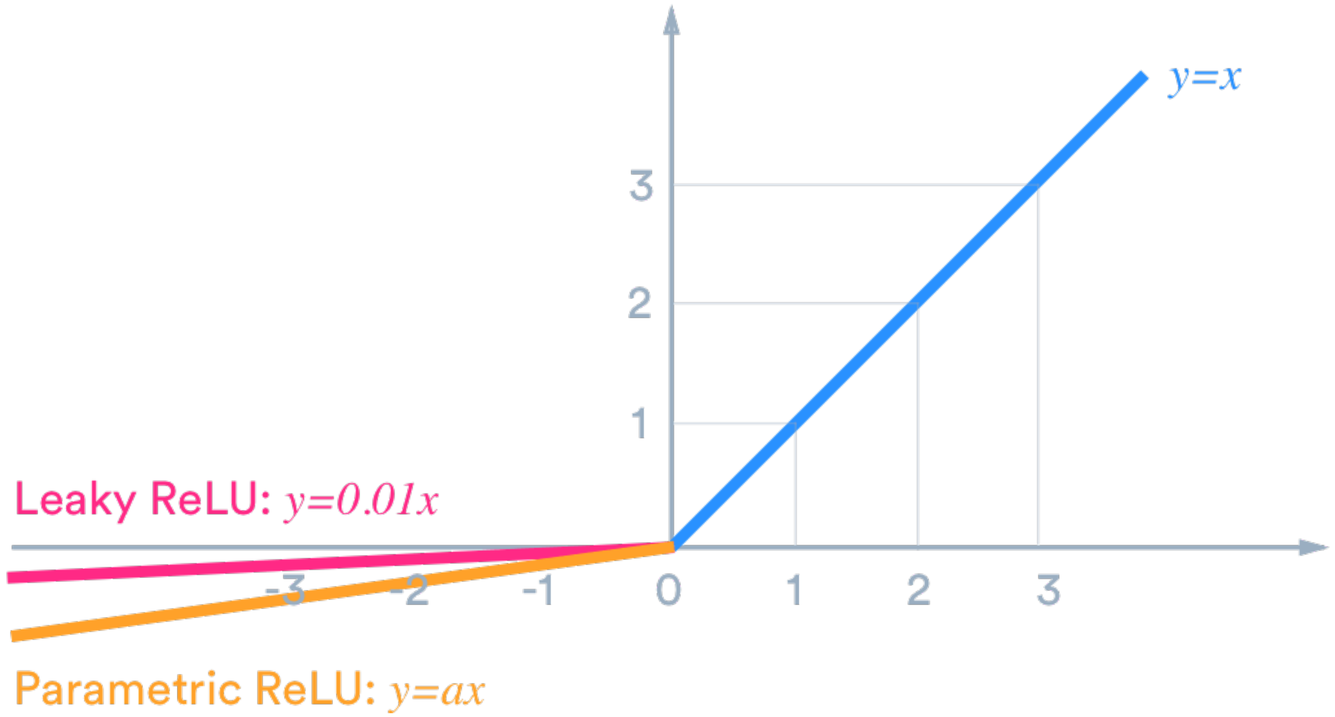
  - Gradient for i-th connection:

$$\mathbf{1}[\mathbf{w}^\top \mathbf{x} + b \geq 0] \cdot x_i$$

- **Problem.** Some neurons have $\mathbf{w}^\top \mathbf{x} < -b$ for most $\mathbf{x}$, thus outputting 0
  - e.g., most weights are negative and $\mathbf{x}$ is also a ReLU output

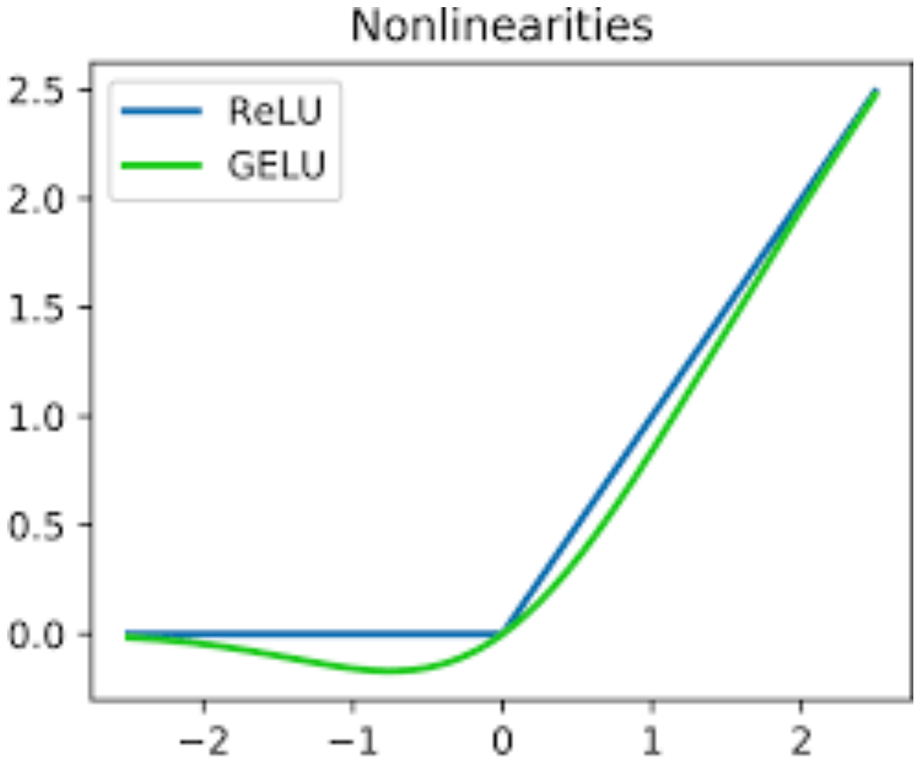
Dying ReLU

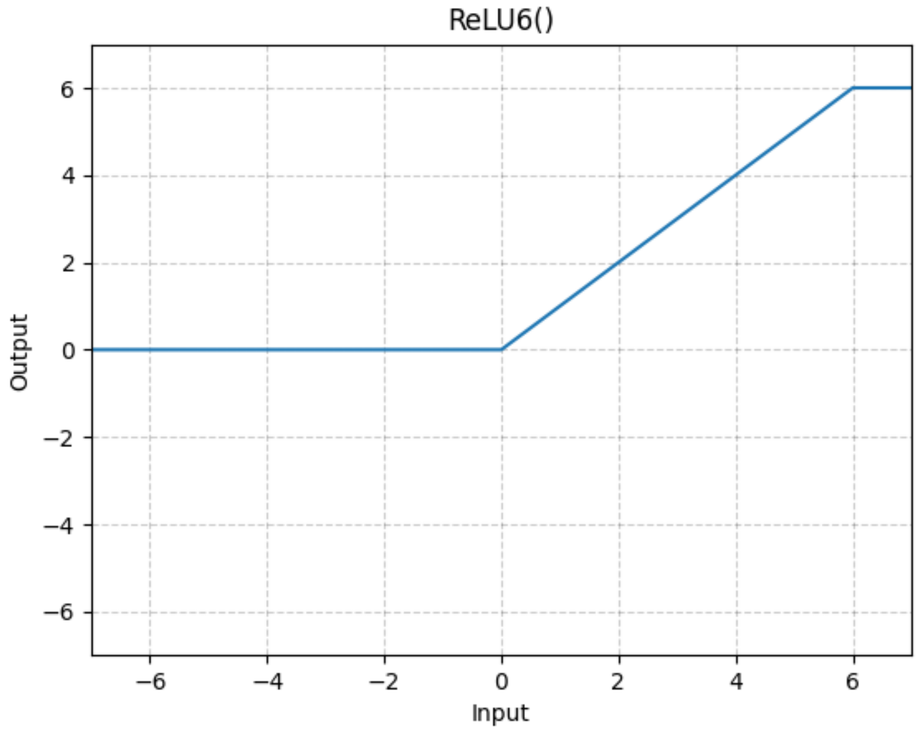
- **Solution.**
  - Initialize the bias b as a small-but-positive value
  - Use variants
    - “leaky” ReLU / ELU / …



Modern choices

- Practitioners training giant models love GeLU / Swish / …
  - Quantization people love ReLU6
- **Recommendation.** Try ReLU as a default
  - Then try variants for squeezing out maximum performance





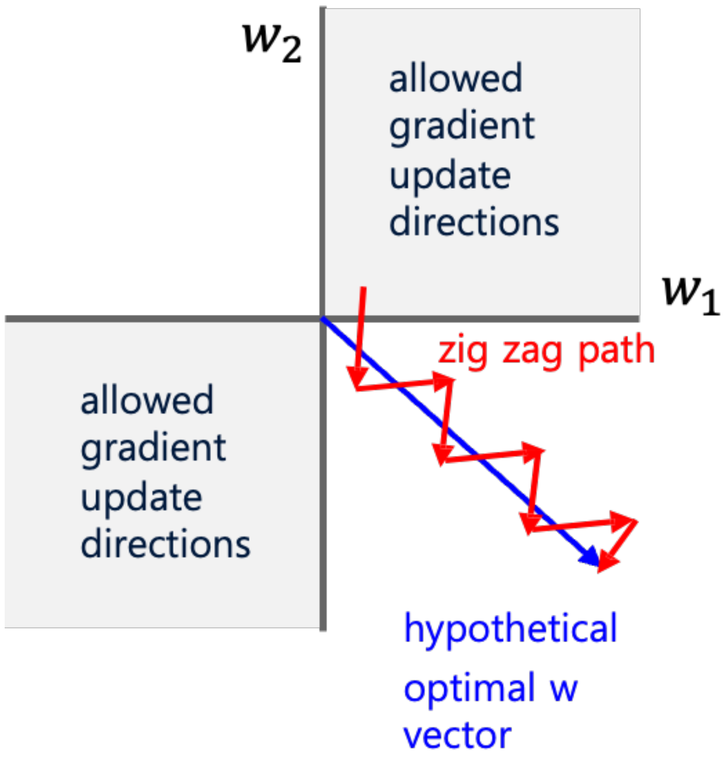
Data preprocessing

Data preprocessing

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

positive

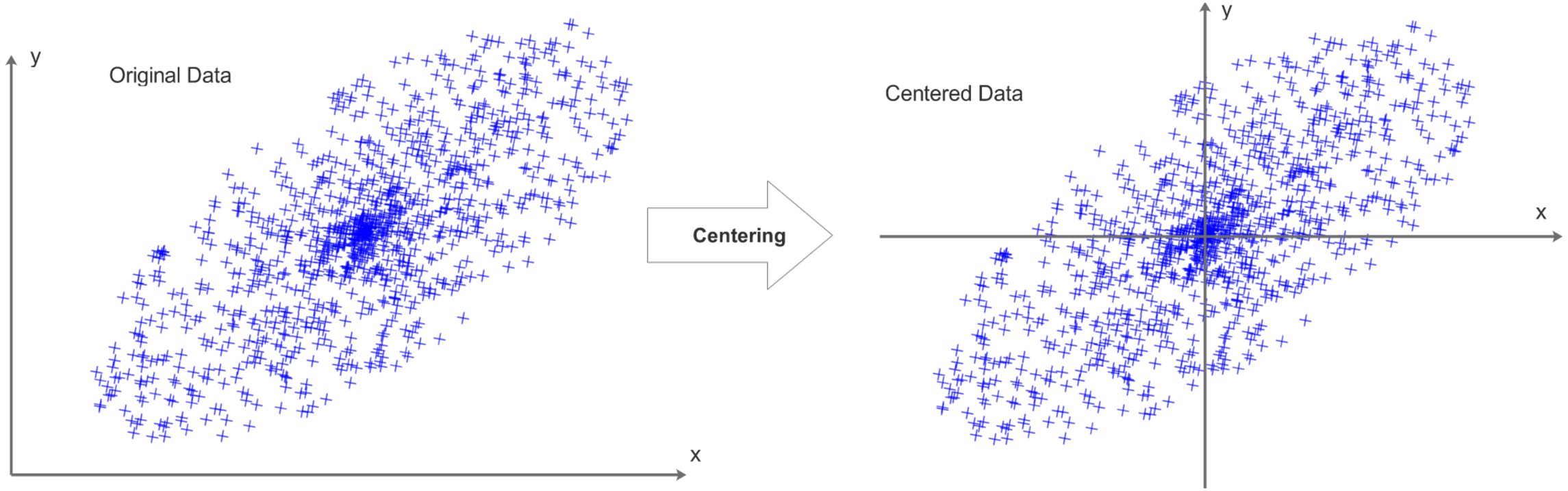
positive?



Data preprocessing

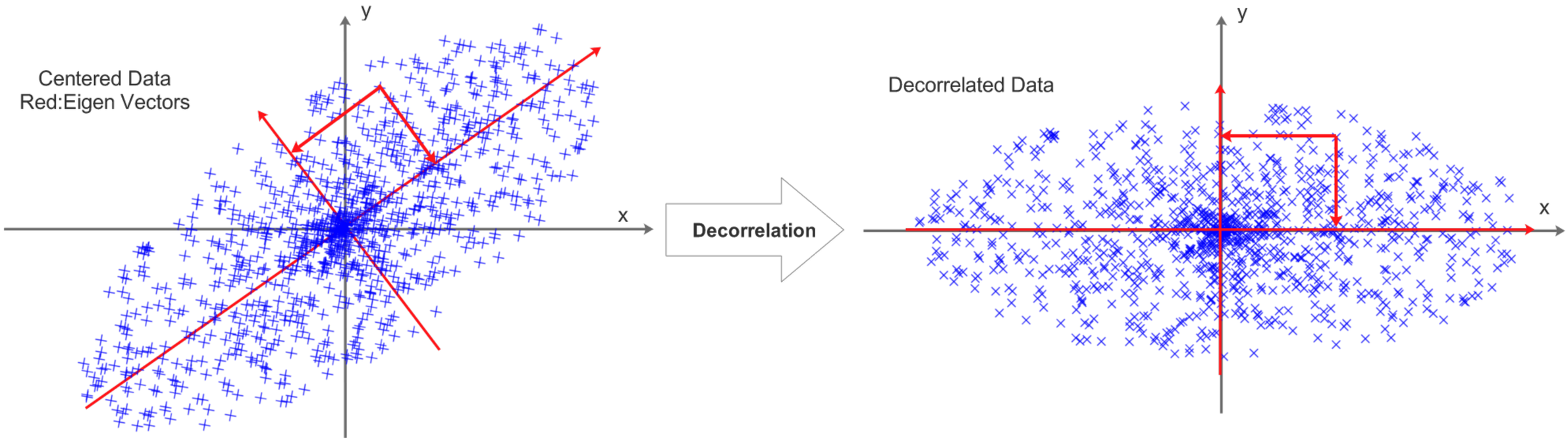
- **Idea.** Force the data to have different signs

- *Centering*. Shift the data to have a zero-mean
  - subtract the mean



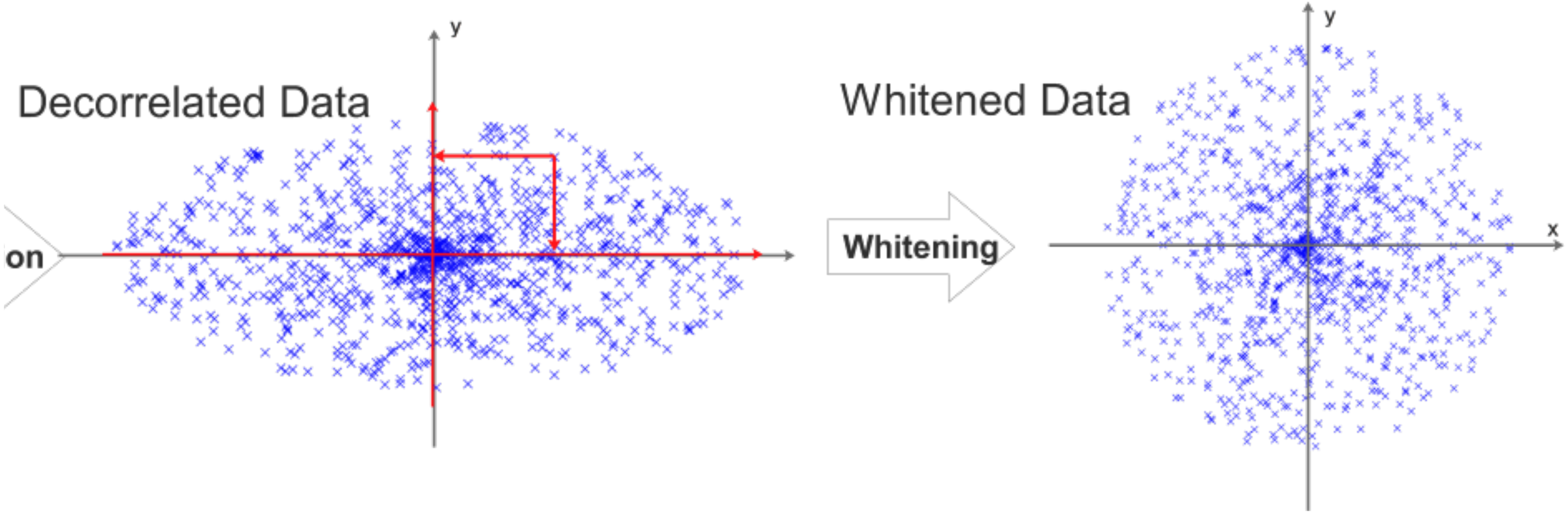
Data preprocessing

- For classic ML, it is also typical to do:
- *Decorrelation*. Make the axes have no correlation
  - Find the principal component, subtract, and repeat, …



Data preprocessing

- For classic ML, it is also typical to do:
- *Standardization*. Make each dimension have unit variance or range
  - Avoids being biased by the data scale
  - (advanced; provably better convergence of GD)



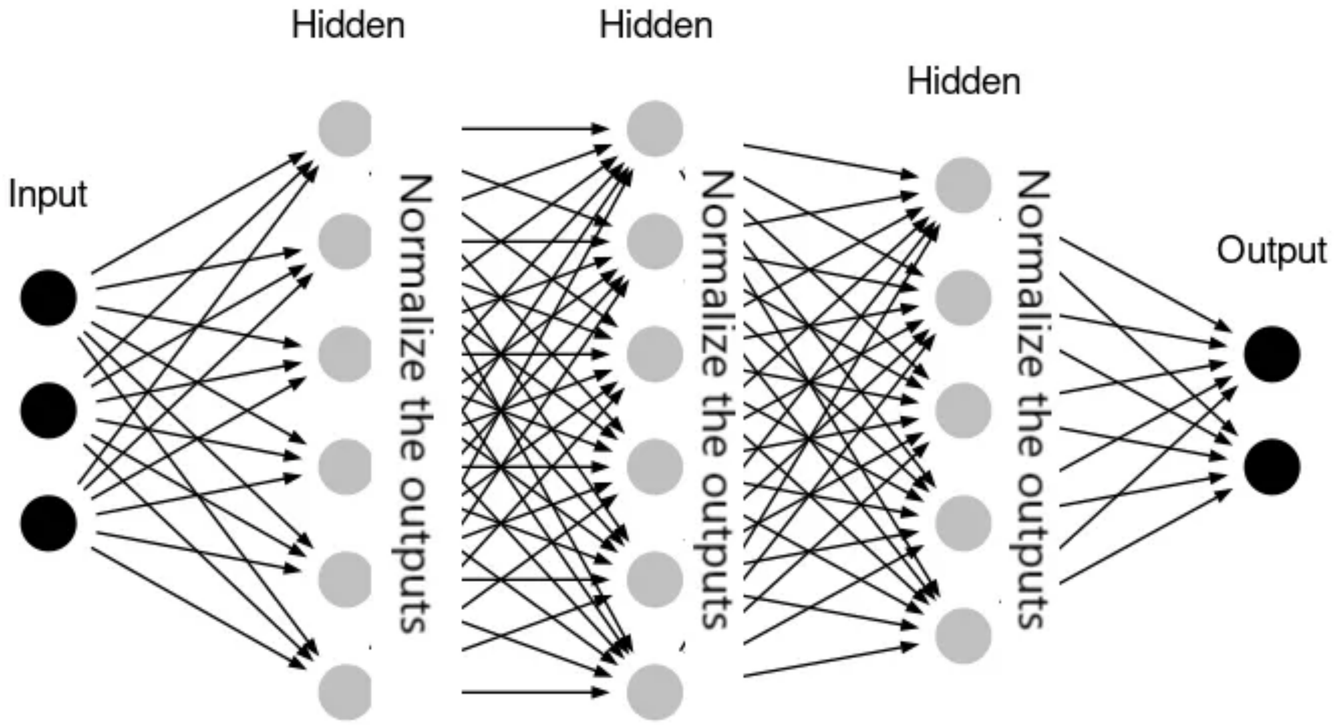
Data preprocessing

- In some cases, we perform dimensionality reduction
  - e.g., PCA, information-theoretic tests, …
- Many of these are not effective for some data, such as images
  - Don’t make too much sense to decorrelate the pixels
  - For CIFAR-10:
    - AlexNet: Simply subtract the mean image ([32,32,3] tensor)
    - VGG: Subtract the mean along RGB channels (3-dim values)

Normalization layers

Normalization layers

- **Idea.** Perform centering + scaling in intermediate layers
  - zero-mean
  - unit variance
  - Many different styles



Batch Normalization

- Consider a batch of activations at some layer:

$$\mathbf{z}_1, \ldots, \mathbf{z}_B \in \mathbb{R}^d$$

  - Here, B is the batch size
  - Each activation has d channels

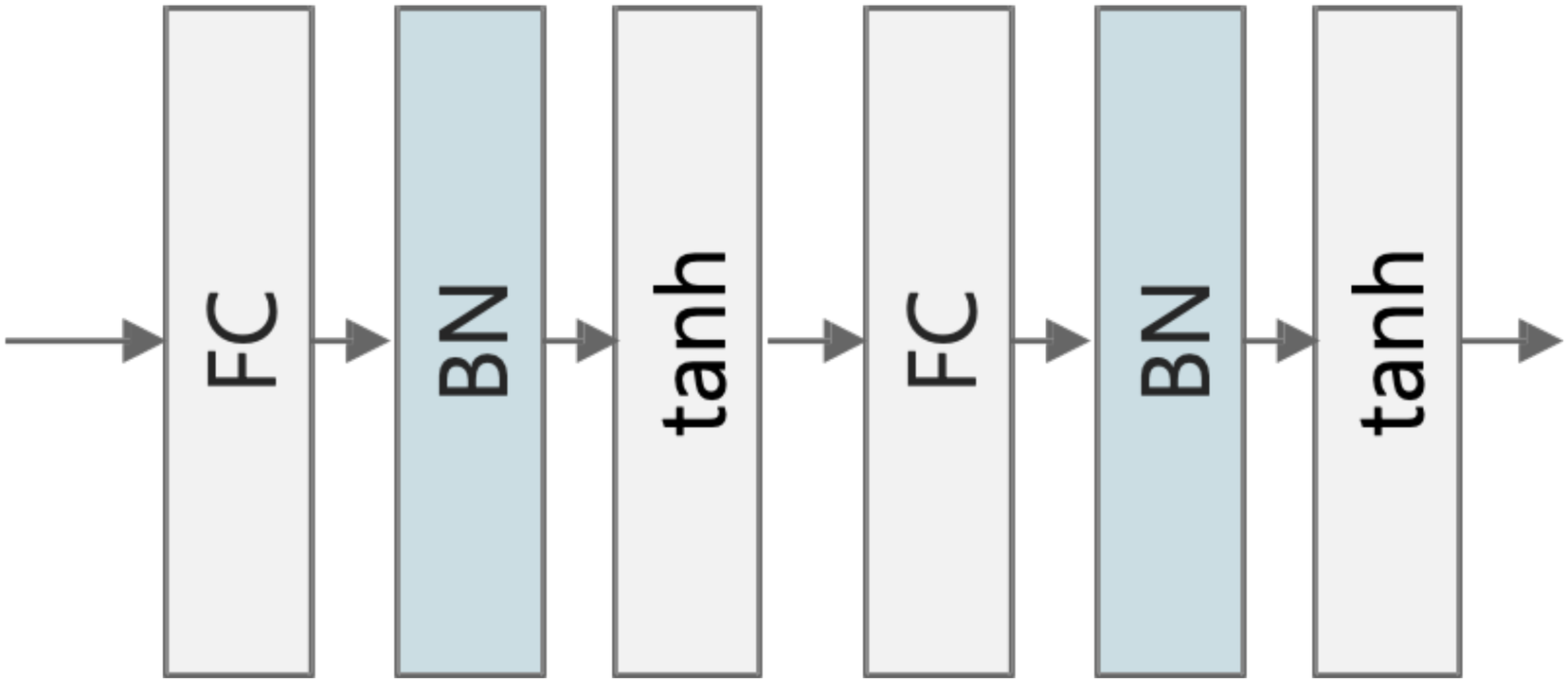
- **BatchNorm.** For each dimension, apply

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

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

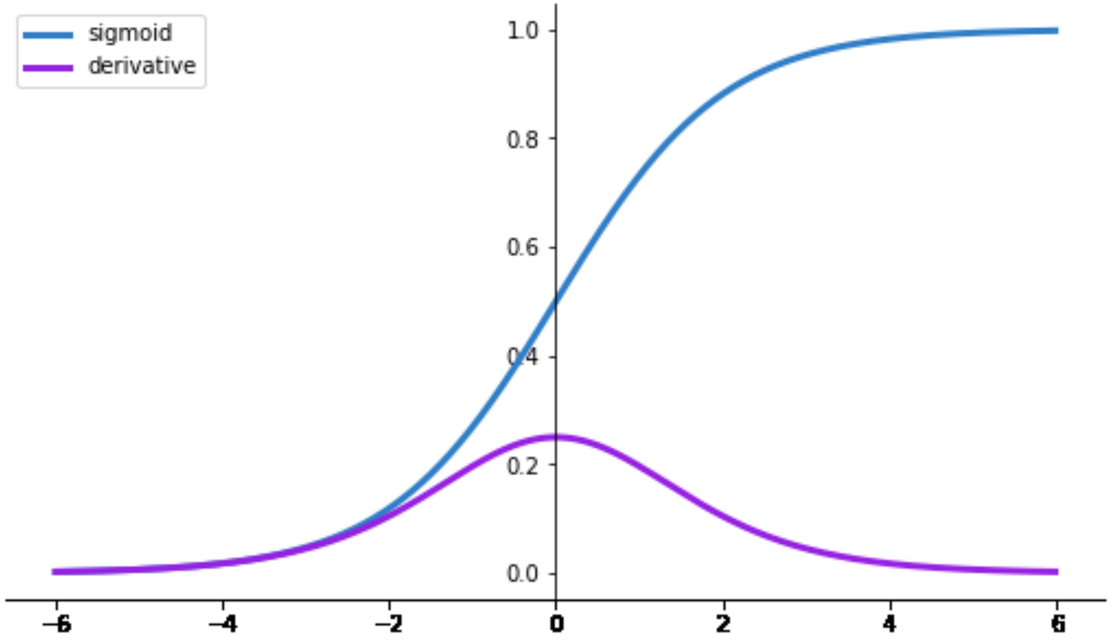
Where to add BatchNorms?

- Mostly placed at…
  - After each linear operation, e.g., linear / convolution
  - Before activation function



Where to add BatchNorms?

- **Problem.** May be harmful to put BN after all layers
  - Normalizing pre-sigmoids makes it behave like linear functions



- **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 and negate the BN whenever needed

Inference phase

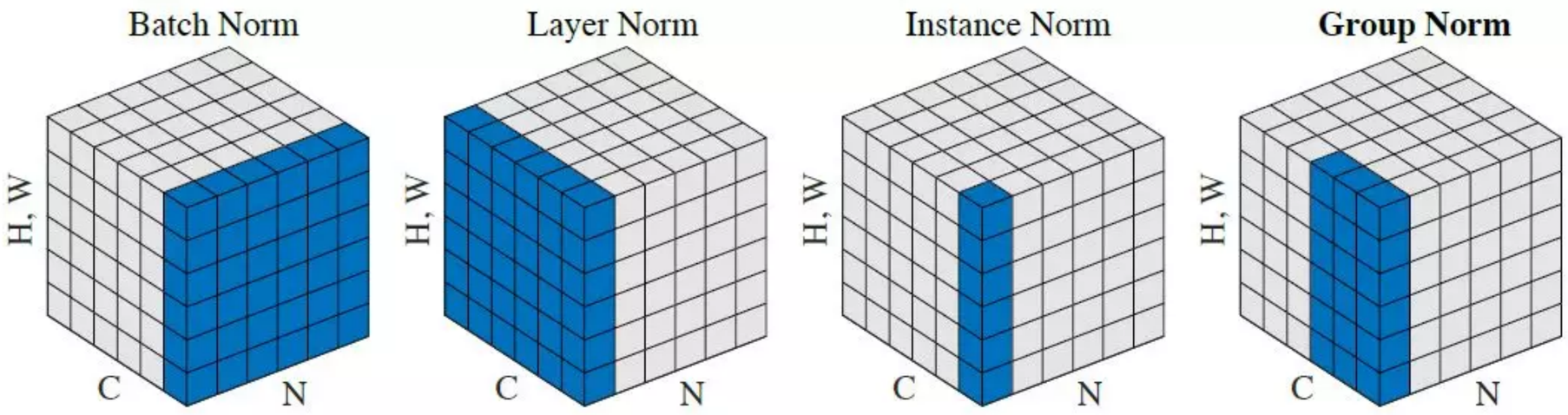
- **Problem.** At the test time (i.e., inference phase), data does not come in as a batch

- **Solution.** Take a running average of the mean & variance during the training
  - Use these values to normalize at the test time!
  - Can be merged into linear layers for a speedup

Considerations

- **Problem.** Often, there are undesired side effects
  - Training instability
  - Sensitive to distribution shifts & batch sizes

- **Solution.** Many variants
  - LayerNorm, RMSNorm, …



Advantages

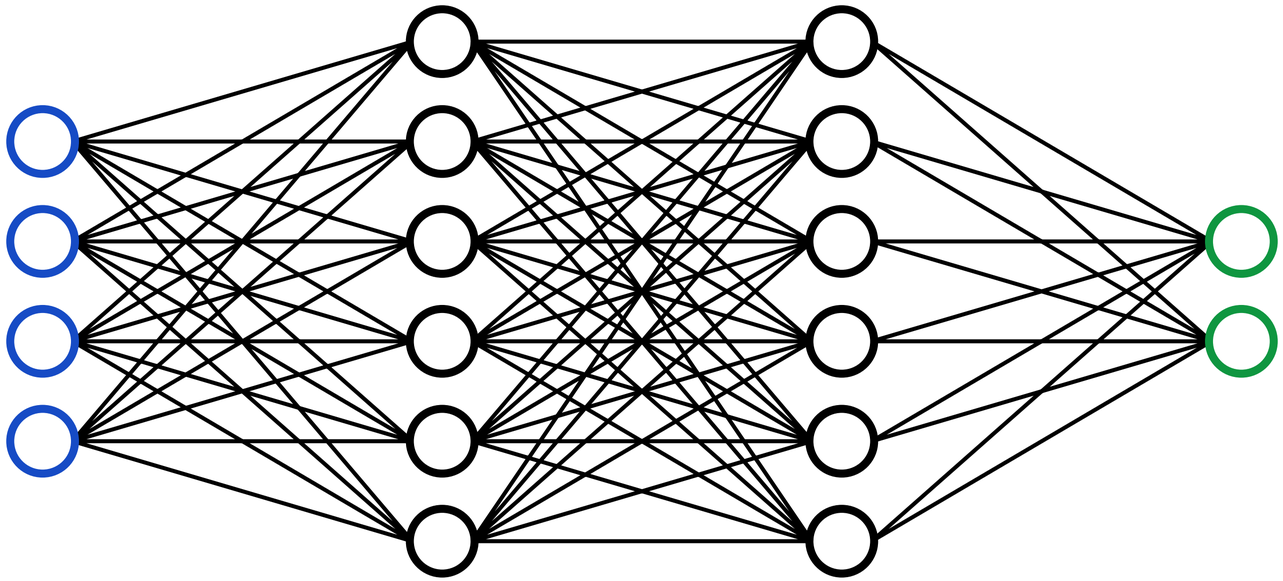
- However, in most cases, the pros outweigh the cons
  - Improve the gradient flow during training
  - Allows higher learning rate
  - Reduces the initialization sensitivity

Weight Initializations

Initializing the weights

- Similar to most iterative optimizations (e.g., K-Means),
  SGD-based optimization of neural net is also sensitive to initialization

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



Initializing the weights

- **Idea.** Randomly initialize all weights:

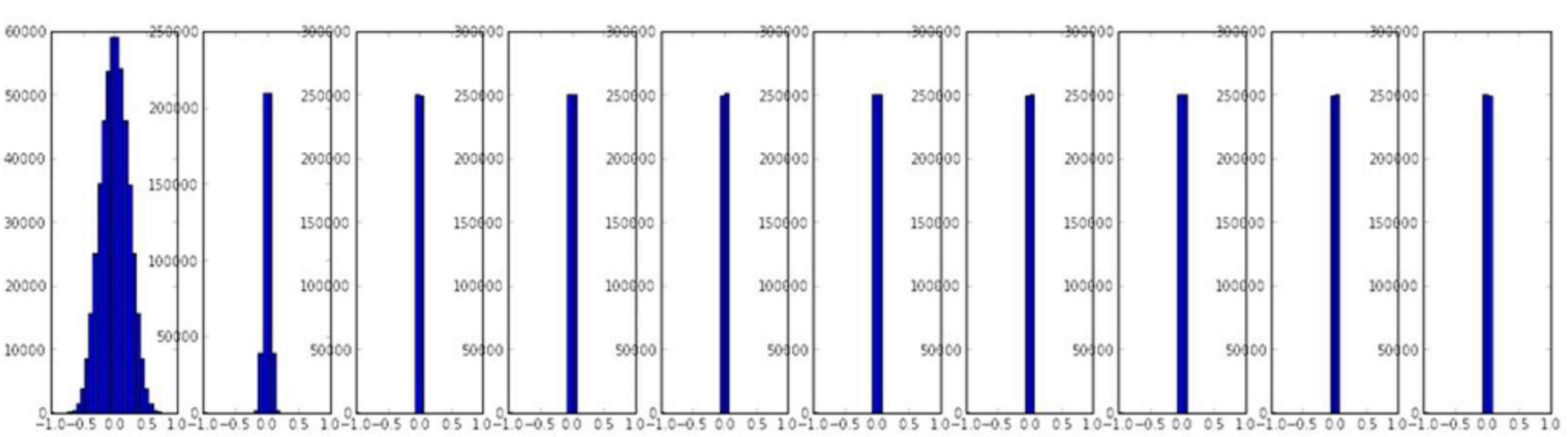
$$w \sim N(0, a^2), \qquad \text{for some } a > 0$$

  - *Shallow nets*. Any $a > 0$ works reasonably well
  - *Deep nets*. Very sensitive to the choice of a

Initializing the weights

- **Small** a**.** The activation $\sigma(\mathbf{w}^\top \mathbf{x})$ becomes very small for deep layers
  - Thus, the gradient also becomes very small:

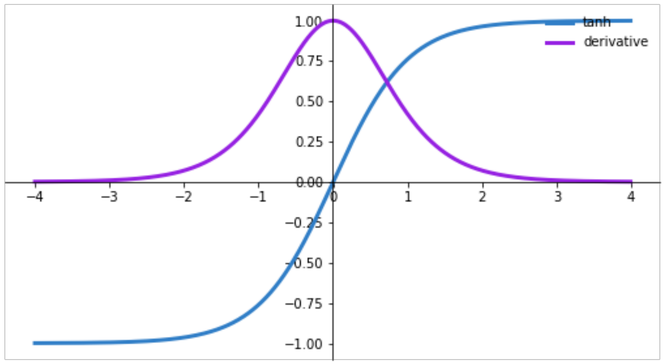
$$\nabla_w \sigma(\mathbf{w}^\top \mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot \mathbf{x}$$

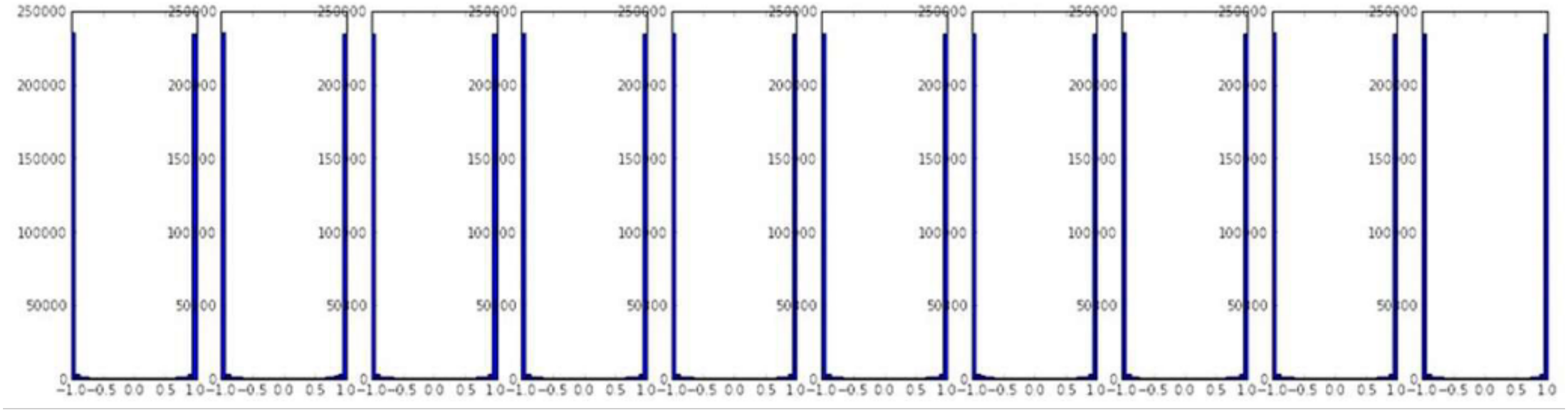


Initializing the weights

- **Large** a**.** Depends on the activation function
  - $\tanh$: Gradients becomes very small
    - Function output: $\sigma(a \cdot \mathbf{w}^\top \mathbf{x}) \rightarrow \{\pm 1\}$
    - Gradients $\nabla_w \sigma(\mathbf{w}^\top \mathbf{x}) = \sigma'(\mathbf{w}^\top \mathbf{x}) \cdot \mathbf{x}$

small if a is large





Initializing the weights

- **Large** a**.** Depends on the activation function
  - 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

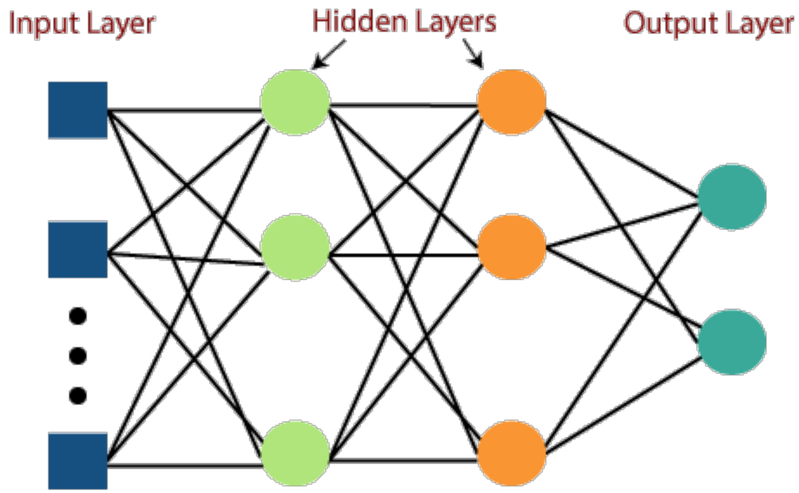
- **Idea.** Choose a so that the activation scale is similar over the layers
  - *Glorot init (2010)*. Use the scaling factor

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

  - *He init (2015)*. Use the scaling factor

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

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



Weight-scaled initialization

- Suppose that we have a layer with d inputs and output neurons.
  - The input activation is $\mathbf{x} = (x_1, \ldots, x_d)$
  - The weight that connects the i-th input neuron to j-th output neuron is denoted by w_{ij}
    - Drawn independently from $N(0, a^2)$

- **Goal.** Make the activation scale similar, i.e.,

$$\|\mathbf{x}\|^2 \approx \mathbb{E}\|\mathbf{W}\mathbf{x}\|^2$$

Weight-scaled initialization

$$\|\mathbf{x}\|^2 \approx \mathbb{E}\|\mathbf{W}\mathbf{x}\|^2$$

- Inspecting the weights connected to j-th output neuron, we have:

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

- 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 need to let $a = 1/\sqrt{d}$

Remarks

- There are many research on how to initialize:
  - Orthogonal initialization
  - Identity initialization
  - Zero initialization

- Has been mostly okay with BNs
  - But BNs are being less used nowadays

</lecture 13>