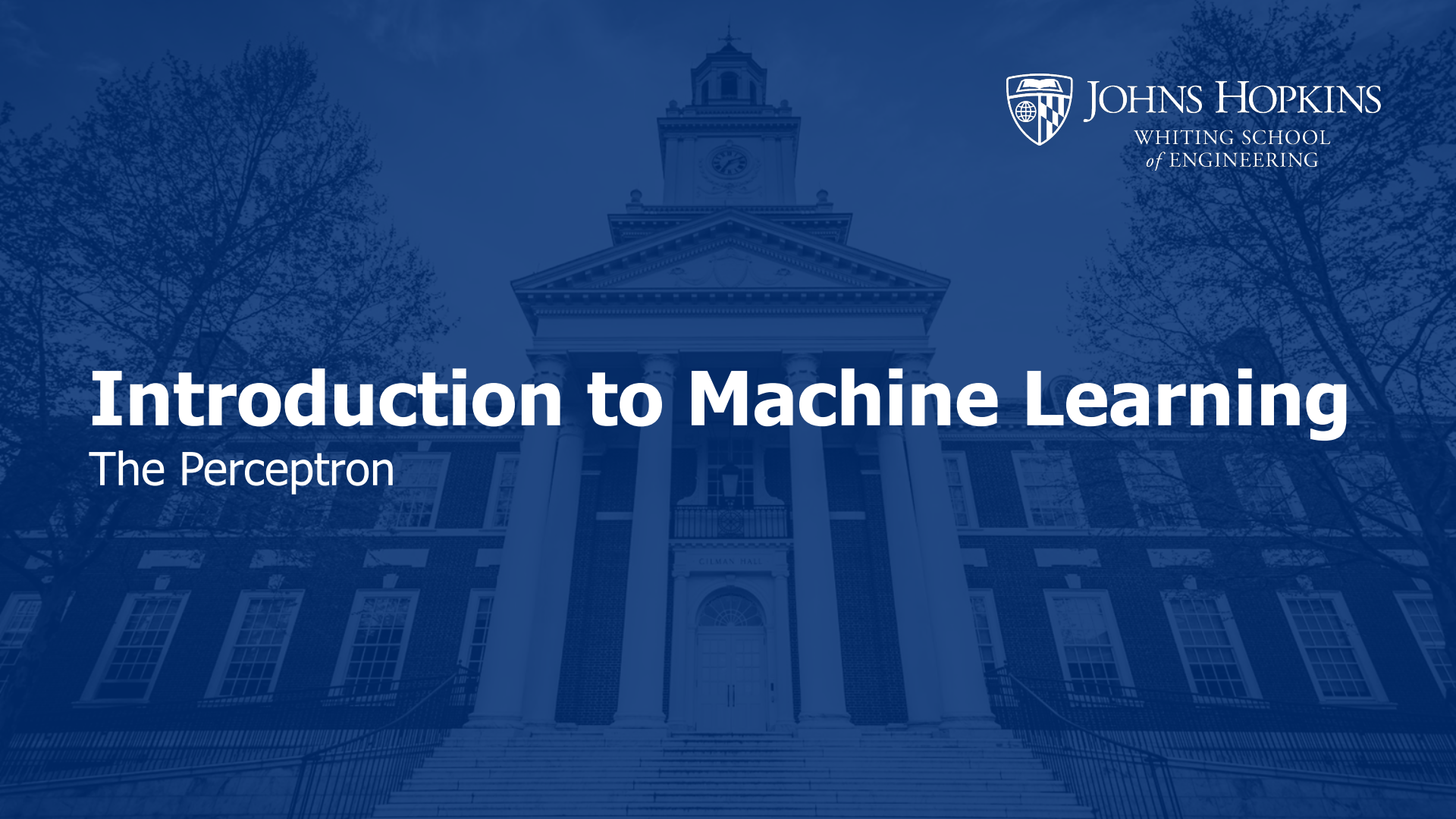


Introduction to Machine Learning

The Perceptron

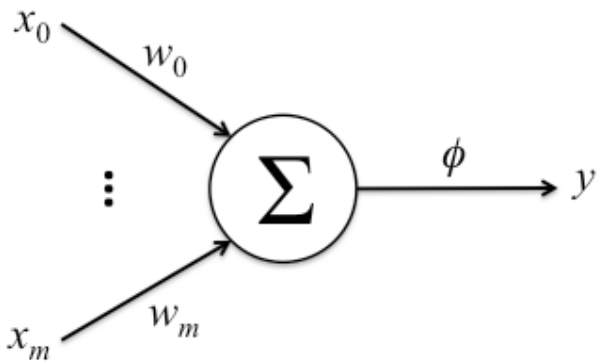
The Perceptron

- Recall notion of function approximation:

$$f : \mathbf{X} \subset \mathbb{R}^m \rightarrow \mathbf{Y} \subset \mathbb{R}^n$$
$$\mathcal{D} = \left\{ \left(\mathbf{x}^t, \mathbf{y}^t \right) \right\}_{t=1}^{|\mathcal{D}|}$$
$$\mathbf{y}^t = f\left(\mathbf{x}^t \right) \text{ indicates the "target"}$$

- The model representation is a network (artificial neural network):
  - Neurons
  - Nodes
- Activation function

$$y_k = \phi \left(\sum_{i=1}^{m} w_{ki} x_i \right)$$



The Loss Function

- The perceptron is generally used for classification.
- Even so, the loss function used is mean squared error.
- The network encodes a function $\hat{f}_{\mathbf{w}}(\mathbf{x}) \approx f(\mathbf{x})$
- Minimize

$$\mathrm{MSE}(\mathbf{w}) = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} \left(f(\mathbf{x}) - \hat{f}_{\mathbf{w}}(\mathbf{x}) \right)^2$$

- We find $\mathbf{w}^* = \mathrm{argmin}_{\mathbf{w}} \, \mathrm{MSE}(\mathbf{w})$

The Perceptron Network

- Two layers of neurons/nodes.
  - Distributes the inputs
  - Apply activation function to weighted sum

$$\phi(y) = \phi\left(\sum_{i=1}^{d} w_i x_i - \theta\right) = \operatorname{sgn}\left(\sum_{i=1}^{d} w_i x_i - \theta\right)$$

where

$$\operatorname{sgn}(x) = \begin{cases} +1 & x > 0 \\ -1 & x \leq 0 \end{cases}$$

- Activation is a step function (aka, hard limiter or Heaviside function)

Perceptron Training

- Let

$$\mathbf{w} = [w_1, \ldots, w_d]^\top$$
$$\mathbf{x} = [x_1, \ldots, x_d]^\top$$
$$z = \text{sgn}\left(\mathbf{w}^\top \mathbf{x}\right)$$
$$\mathbf{F} = \mathbf{F}^+ \cup \mathbf{F}^-$$

- **F** is the training set

  $\mathbf{F}^+$ denotes positive examples

  $\mathbf{F}^-$ denotes negative examples

1. Choose initial weight vector $\mathbf{w}$ at random
2. Choose some example $\mathbf{x} \in \mathbf{F}$
3. Consider the $\text{class}(\mathbf{x})$
   - If $\mathbf{x} \in \mathbf{F}^+$ and $\text{sgn}\left(\mathbf{w}^\top \mathbf{x}\right) \leq 0$ then $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}$
   - If $\mathbf{X} \in \mathbf{F}^-$ and $\text{sgn}\left(\mathbf{w}^\top \mathbf{x}\right) > 0$ then $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}$
   - Otherwise, do nothing (class is correct)
4. Repeat steps 2 and 3 until convergence

Example – Logical OR

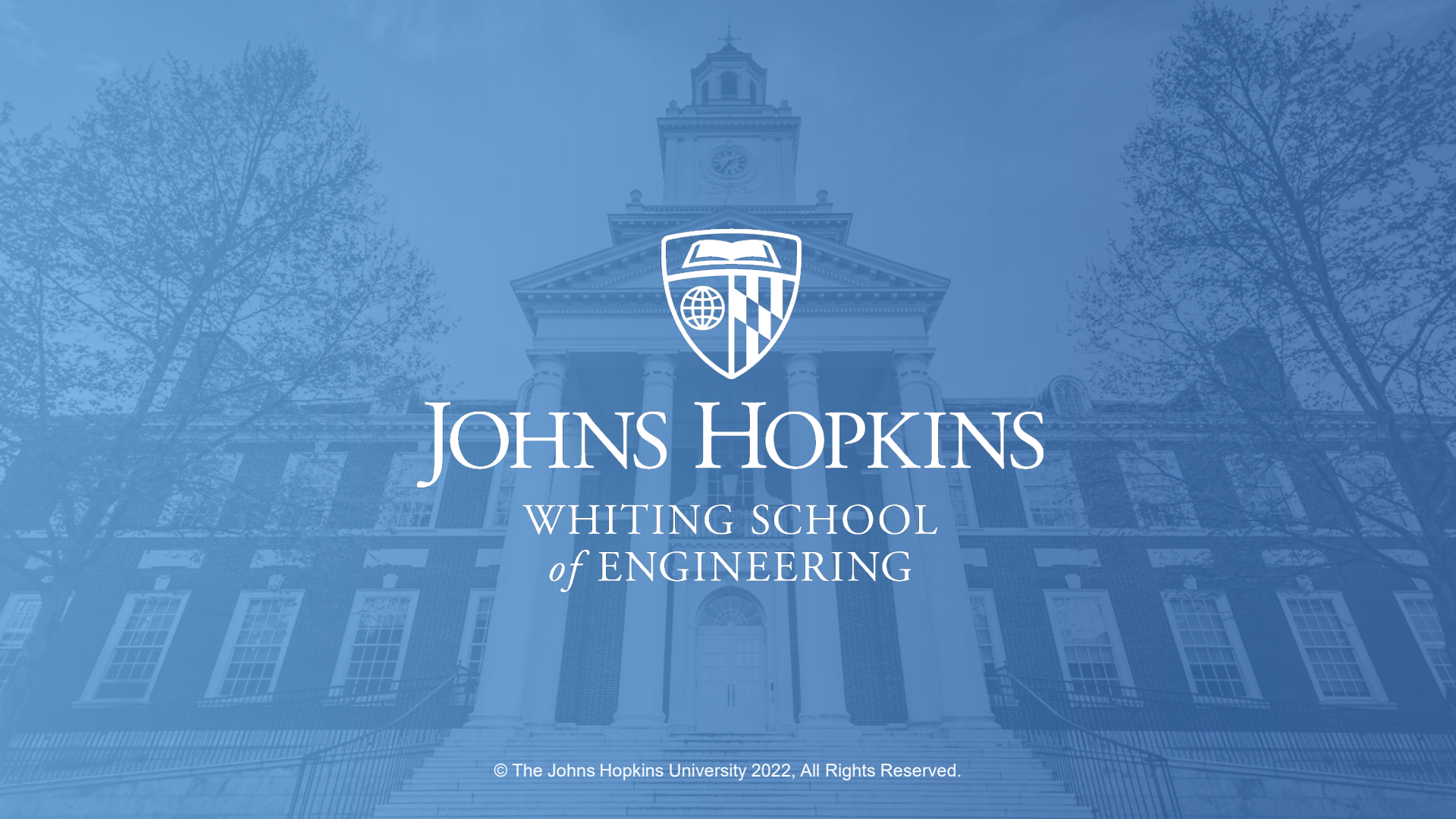
x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

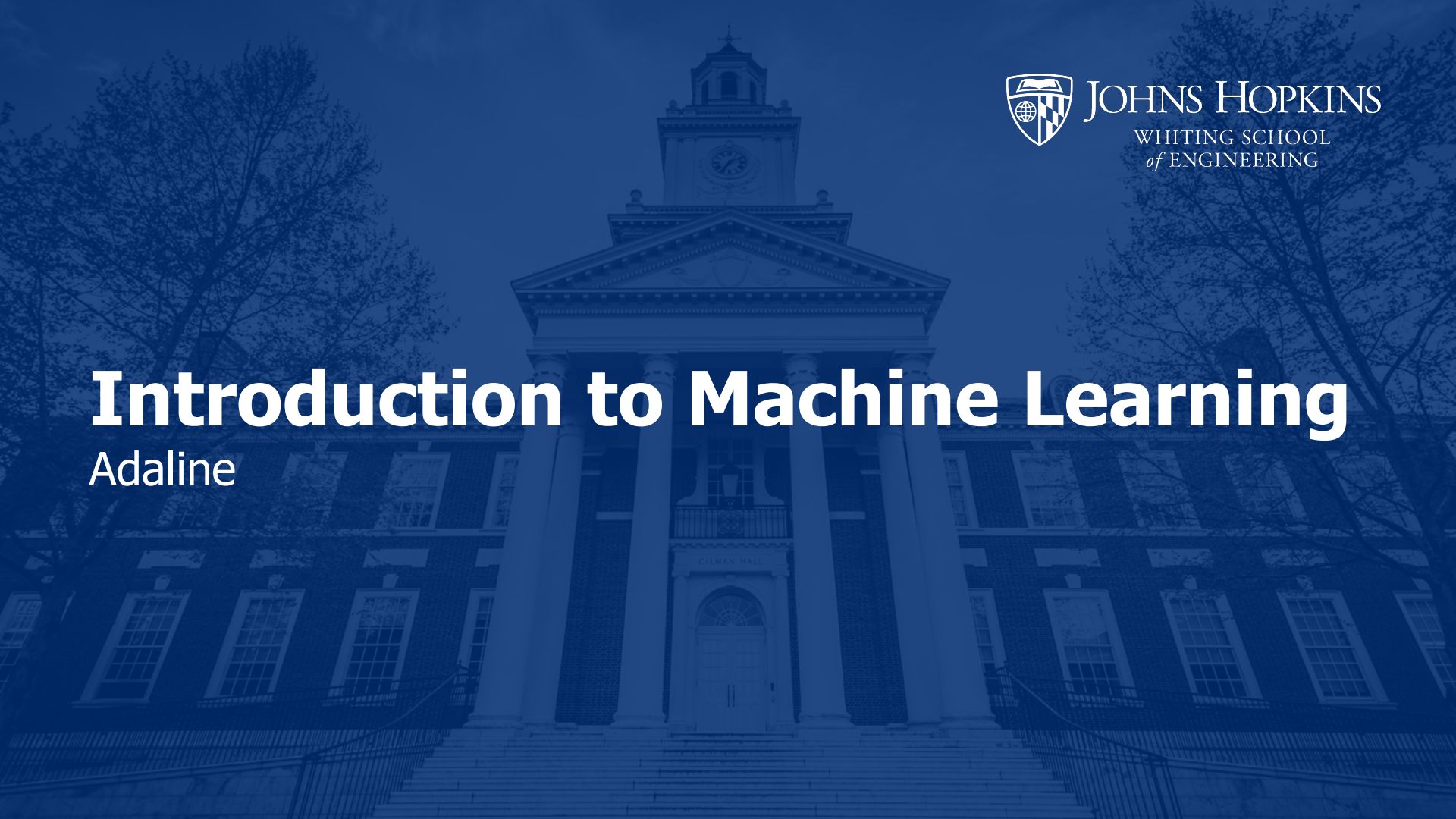
Train

- Initial weight vector $\mathbf{w} = [0 \quad 0]^\top$
- For (0,0), sgn **w** = 0: No change in weights
- For (0,1), sgn **w** = 0: Update $\mathbf{w} = [0 \quad 1]^\top$
- For (1,0), sgn **w** = 0: Update $\mathbf{w} = [1 \quad 1]^\top$
- For (1,1), sgn **w** = 1: No change in weights

Try again

- For (0,0), sgn **w** = 0: No change in weights
- For (0,1), sgn **w** = 1: No change in weights
- For (1,0), sgn **w** = 1: No change in weights
- For (1,1), sgn **w** = 1: No change in weights





Introduction to Machine Learning

Adaline

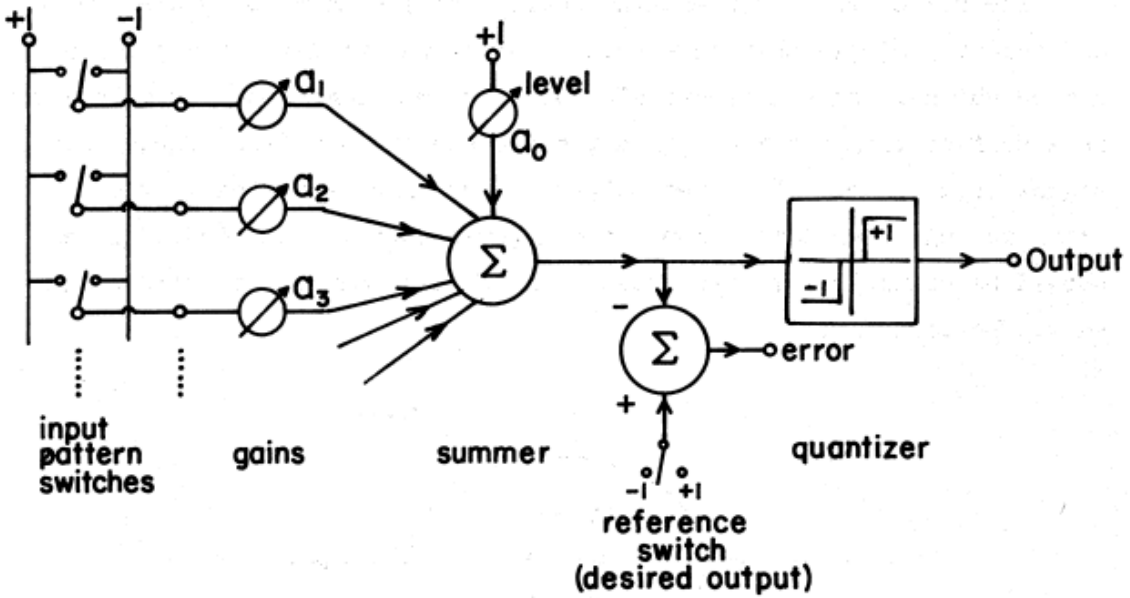
Adaline

- What is Adaline?
  - **Ada**ptive **Li**near **Ne**uron
  - **Ada**ptive **Lin**ear **E**lement
- Similar to the perceptron,
  - Let $\mathbf{w} = [w_1, \ldots, w_d]^\top$ be a weight vector
  - Let $\mathbf{x} = [x_1, \ldots, x_d]^\top$ be an input vector
  - Let Θ be a threshold
- Compute

$$y = \sum_{i=1}^{d} w_i x_i - \theta$$

- Note, $x_0 = -1$ and $w_0 = \theta$

Adaline (cont.)



B. Widrow, *An Adaptive "Adaline" Neuron Using Chemical "Memristors"* Technical Report No. 1553-2, Stanford University, October 17, 1960.

Adaline Training

- Let
  - *y*: network output = $\hat{f}_{\mathbf{w}}(\mathbf{x})$
  - *d*: desired output = $f(\mathbf{x})$
- Minimize squared error: $Err = (d - y)^2$
- Apply a learning rate $\eta \in (0, 1)$
- Then the weight update rule is

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \eta(d - y)\mathbf{x}$$

Widrow-Hoff Rule

- The function we are learning: $\hat{f}_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$
- Mean squared error:

$$\mathrm{MSE}(\mathbf{w}) = \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \left(f\left(\mathbf{x}_i\right) - \mathbf{w}^\top \mathbf{x}_i \right)^2 \Bigg) = \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \left(d_i - y_i\right)^2$$

- Calculating the gradient:

$$\nabla_{\mathbf{w}} \mathrm{MSE}(\mathbf{w}) = -\frac{2}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \left(f\left(\mathbf{x}_i\right) - \mathbf{w}^\top \mathbf{x}_i \right) \mathbf{x}_i$$

- Weight update:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \eta \nabla_{\mathbf{w}} \mathrm{MSE}\left(\mathbf{w}_t\right) = \mathbf{w}_t - \eta \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} f\left(\mathbf{x}_i\right) \left(\mathbf{w}_t^\top \mathbf{x}_i \right) \mathbf{x}_i$$

Adaline vs Perceptron

Adaline:

- Batch updating

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta \left(f\left(\mathbf{x}_i\right) - \mathbf{w}_t^\top \mathbf{x}_i \right) \mathbf{x}_i$$

Perceptron:

- Incremental updating:

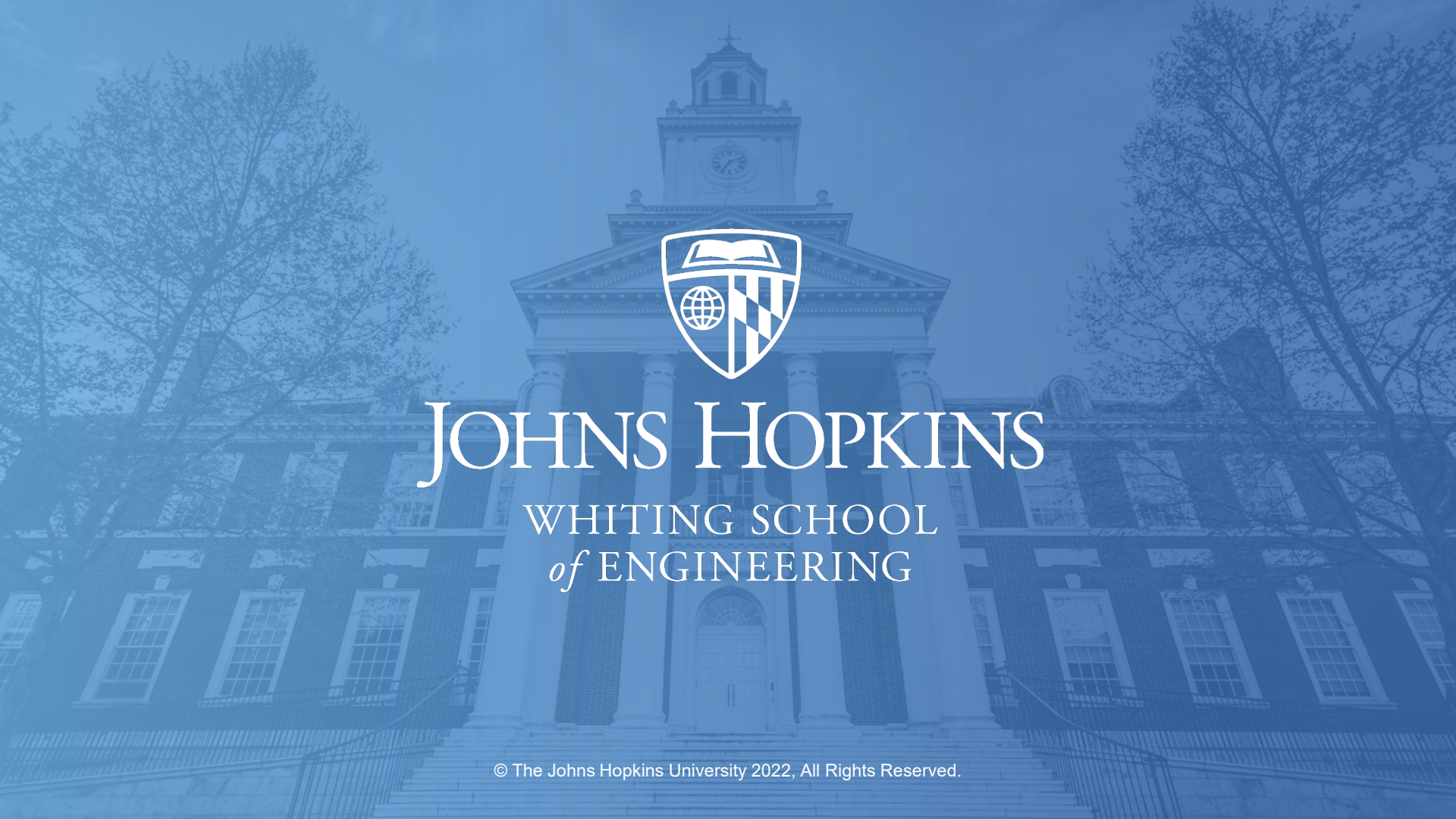
$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t \pm \mathbf{x}$$

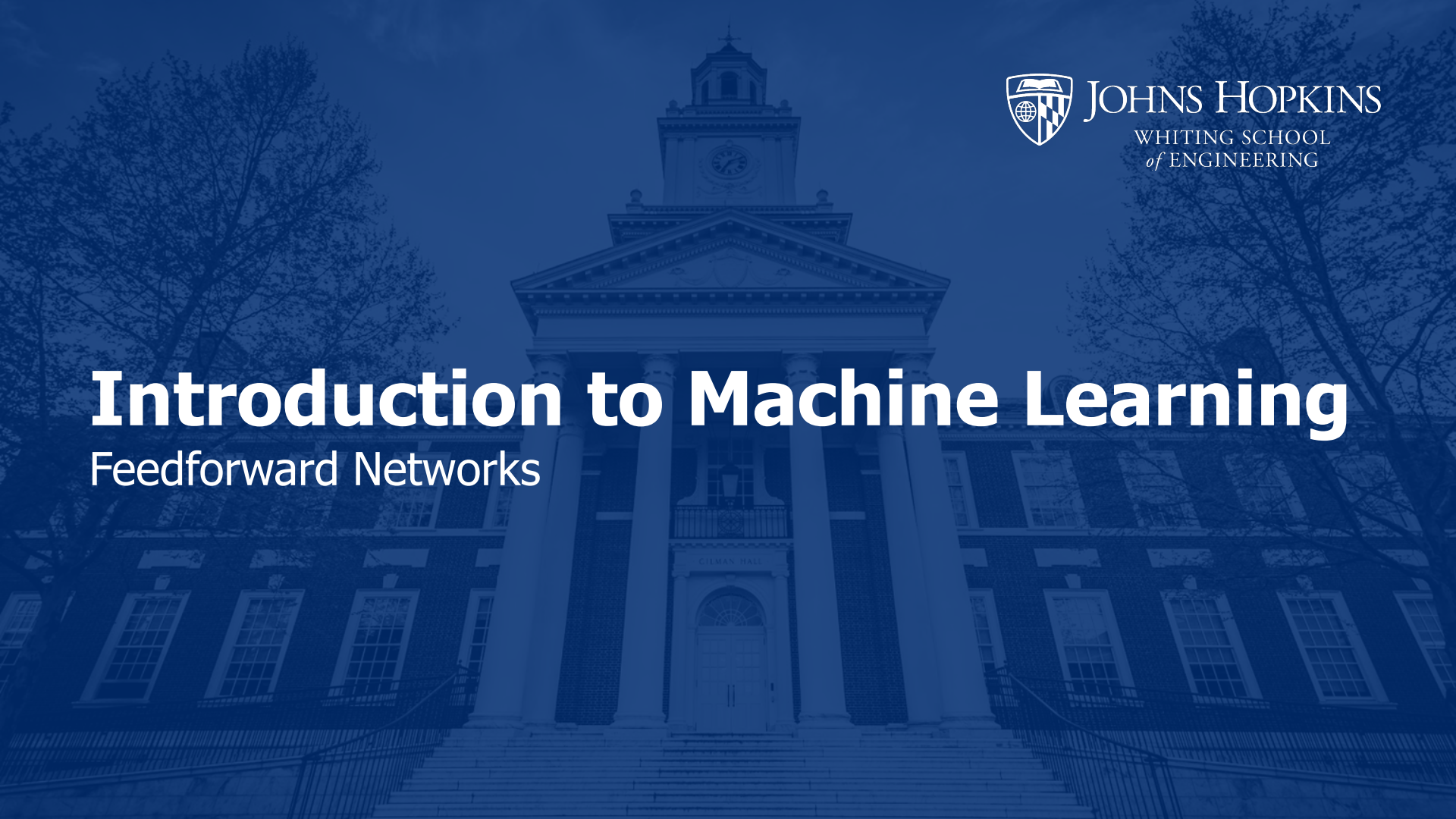
When correct

- Positive: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \eta(+1 - (+1))\mathbf{x}_i = \mathbf{w}_t$
- Negative: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \eta(-1 - (-1))\mathbf{x}_i = \mathbf{w}_t$

When incorrect

- Should be Positive: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \frac{1}{2}(+1 - (-1))\mathbf{x}_i = \mathbf{w}_t + \mathbf{x}_i$
- Should be Negative: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \frac{1}{2}(-1 - (+1))\mathbf{x}_i = \mathbf{w}_t - \mathbf{x}_i$



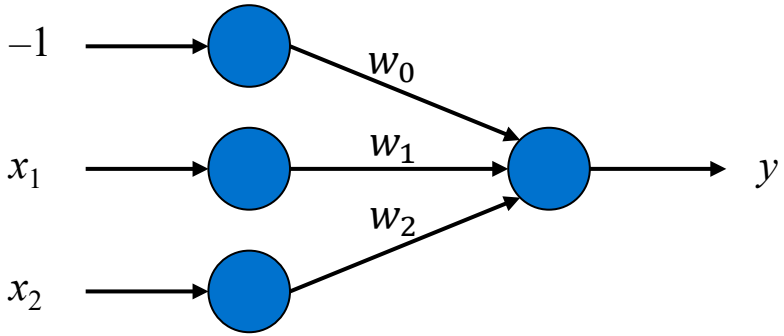


Introduction to Machine Learning

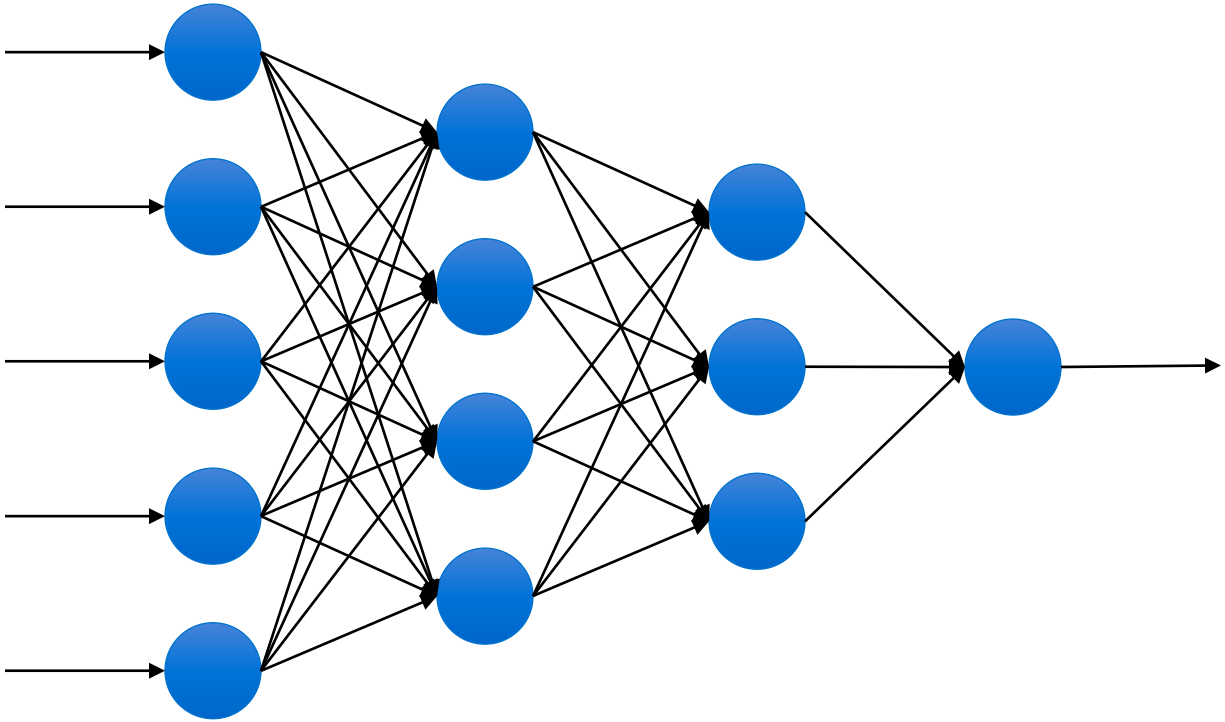
Feedforward Networks

Perceptrons

- Previously, we examined the Perceptron and Adaline, which are linear networks
- Fundamentally, they are the same network.
- They are also examples of what we call "feedforward" networks.
- "Feedforward" refers to the flow of information along the directed edges of the network.



Multilayer Perceptrons



Example Feedforward Networks

- Multilayer Perceptrons
- Radial Basis Function Networks
- (Stacked) Autoencoders
- Convolutional Neural Networks
- Competitive Learning Networks
- Self-Organizing Maps

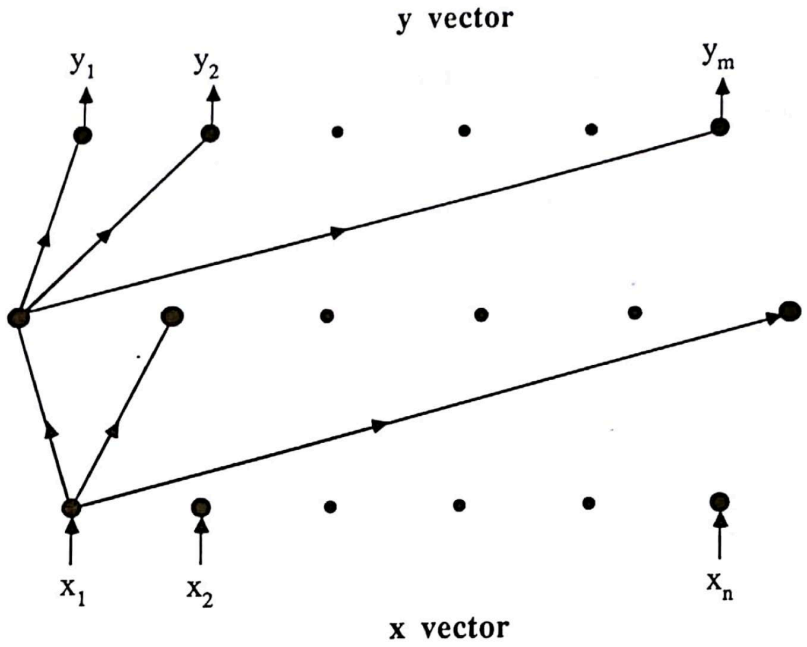
Kolmogorov (Feedforward) Networks

Theorem: Given any *continuous* function $\phi : I^n \to \mathbb{R}^m, \phi(\mathbf{x}) = \mathbf{y}$ where I is the closed unit interval [0,1] (and therefore, I^n is the *n*-dimensional unit cube), φ can be implemented *exactly* by a three-layer neural network having *n* processing elements in the first (*x*– input) layer, (2*n* + 1) processing elements in the middle layer, and *m* processing elements in the top (*y*– output) layer.

- The input units distribute the inputs to the hidden layer.
- The hidden units implement

$$z_k = \sum_{j=1}^{n} \lambda^k \psi\left(x_j + \epsilon k\right) + k$$

- The outputs are $y_i = \sum_{k=1}^{2n+1} g_i\left(z_k\right)$



R. Hecht-Nielsen, "Kolmogorov's Mapping Neural Network Existence Theorem," *Proc. Int. Conf. on Neural Networks*, 1987.

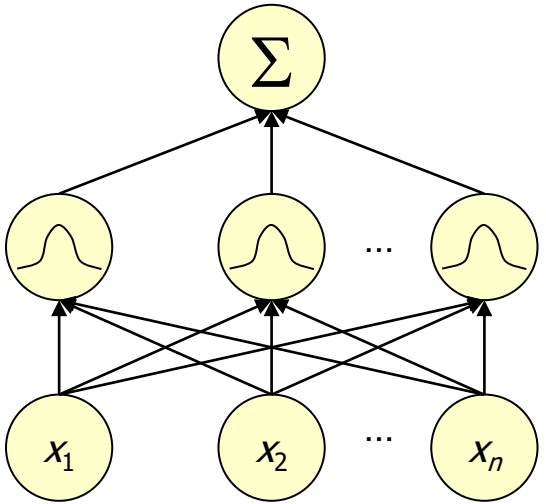
RBF Networks

- Let $g(x, a, b) = K((x - a)/b)$ be an RBF.
- Let Σ_g^ℓ denote the class of functions realized by an RBF network with $\ell - 2$ hidden layers and a linear output layer.
- Let $L^1(\mathbb{R}^d)$ denote the L-1 norm.

Theorem: If $g(x, a, b) = K((x - a)/b)$ is an RBF and *K* is integrable, then Σ_g^3 is dense in $L^1(\mathbb{R}^d)$ if and only if

$$\int_{\mathbb{R}^d} K(x)dx \neq 0$$

- Basically, this is saying $\|f - \Sigma_g^3\|_1 \leq \epsilon$ can be found for arbitrary ε.



Training Process

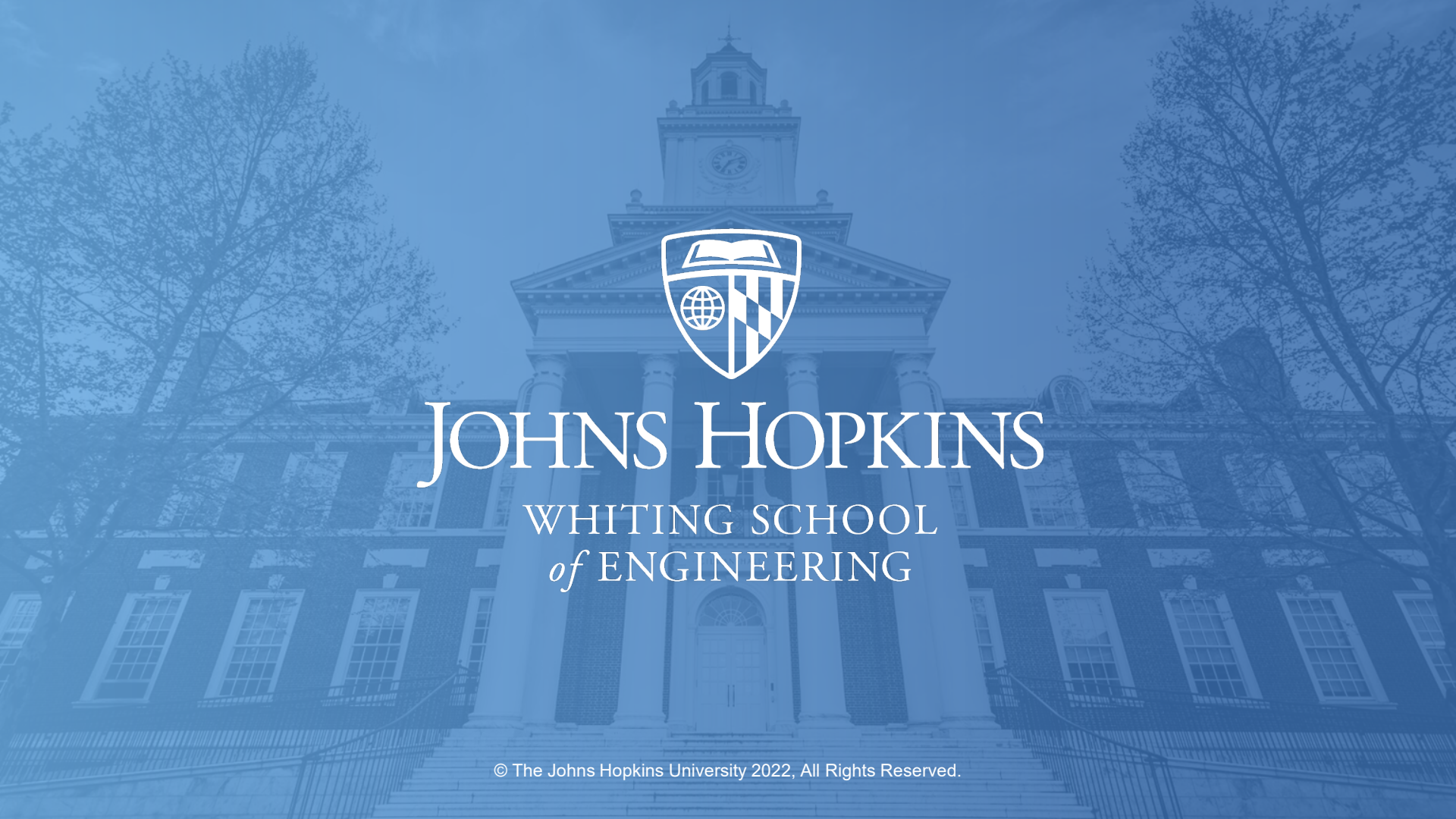
- Typically, feedforward networks are trained in two phases.
- **Phase 1:** The *forward* phase
  - Propagates an input signal forward through the network to generate an output.
  - The weights of the network are kept fixed.
- **Phase 2:** The *backward* phase
  - First calculate an error signal
  - Propagate the error signal from output back towards the input
  - Update the weights as the error signal is propagated
- Leads to what we call "backpropagation"

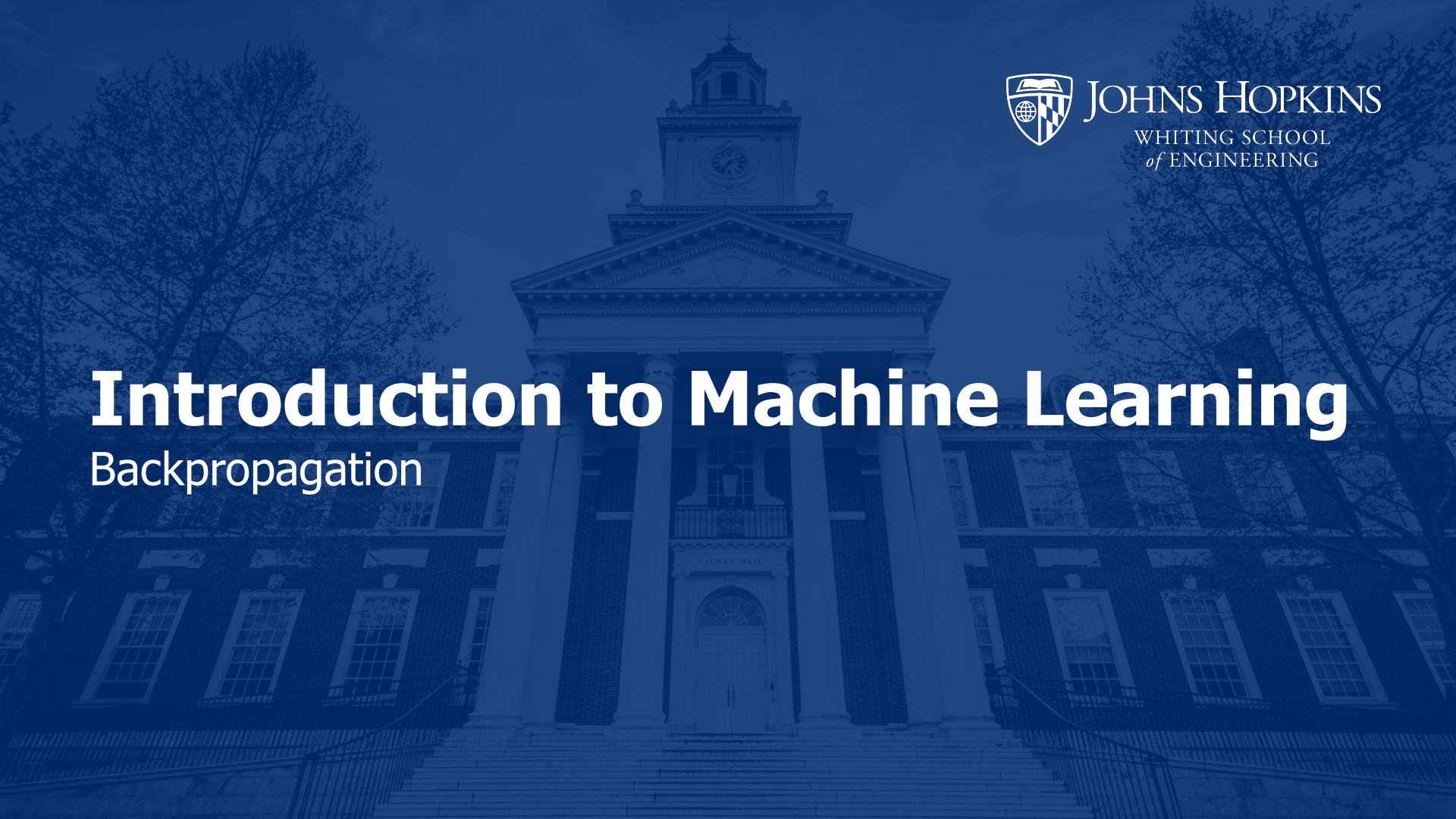
Credit Assignment

- The learning process can be described itself in two steps
  - First, associate "credit" or "blame" to each weight in the network based on impact it has on performance
  - Second, adjust the weights to reinforce credit or to mitigate blame.
  - Implements a form of "Hebbian learning."

$$w_{ji} \approx f\left(x_i x_j\right)$$

- Leads to what we call the "credit assignment problem"
- Often, associated with reinforcement learning, but the issue applies in any parameter learning process as well
- Problem is particularly difficult for weights on edges entering the hidden nodes





Introduction to Machine Learning

Backpropagation

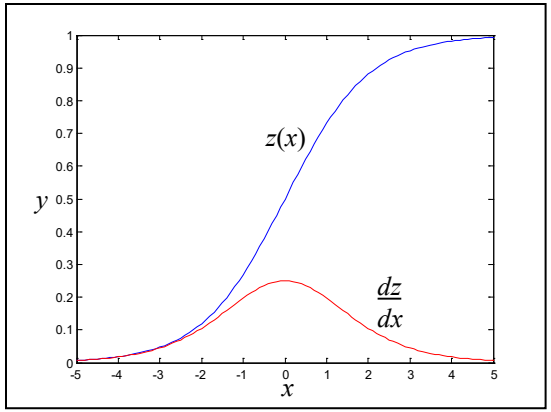
Solving Credit Assignment

- The Backpropagation Algorithm
  - Bryson and Ho, 1969
  - Werbos, 1974
  - Rumelhart, Hinton, and Williams, 1986
- Needed for networks with *nonlinear* hidden nodes
- A network of all linear nodes is equivalent to a simple Perceptron/Adaline network

Sigmoid Activation Functions

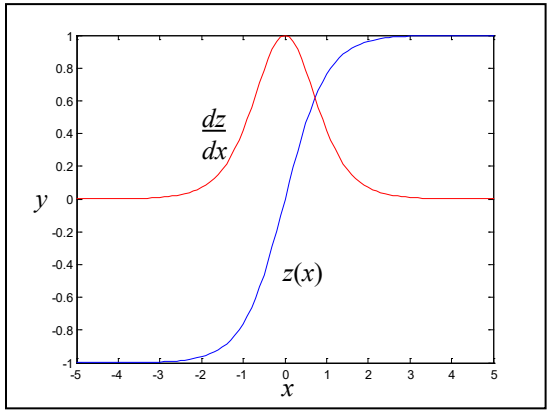
The logistic function

$$z = \frac{1}{1+e^{-x}} \Rightarrow \frac{dz}{dx} = z(1-z)$$



The hyperbolic tangent function

$$z = \tanh x = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \Rightarrow \frac{dz}{dx} = \operatorname{sech}^2 x = \frac{4}{\left(e^{x}+e^{-x}\right)^2}$$



The Algorithm

- Use the chain rule of differentiation
- Example: $\mathbf{x} \in \mathbf{X}$
- Weight update: $\Delta w_{ji} = -\eta \frac{\partial Err_{\mathbf{x}}}{\partial w_{ji}}$ where $\mathrm{Err}_{\mathbf{x}}(\mathbf{w}) = \frac{1}{2} \sum_{k \in \text{ Outputs}} (d_k - o_k)^2$
- Notation

x_{ji} : i[th] input to unit *j*

w_{ji} : weight of i[th] input to unit *j*

$\text{net}_j = \sum_j w_{ji} x_{ji}$

o_j : output of unit *j*

d_j : target output for unit *j*

$\sigma(\cdot)$: sigmoid activation (logistic)

outputs: units in final layer

downstream(j): units whose inputs are the outputs of unit j

Derivation (1)

Objective:

$$\frac{\partial Err_{\mathbf{x}}}{\partial w_{ji}} = \frac{\partial Err_{\mathbf{x}}}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}}$$

$$= \frac{\partial Err_{\mathbf{x}}}{\partial net_j} \cdot x_{ji}$$

Case 1: Outputs:

$$\frac{\partial Err_{\mathbf{x}}}{\partial net_j} = \frac{\partial Err_{\mathbf{x}}}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}$$

First Term:

$$\frac{\partial Err_{\mathbf{x}}}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in \text{ outputs}} (d_k - o_k)^2$$

$$= \frac{\partial}{\partial o_j} \frac{1}{2} (d_j - o_j)^2$$

$$= \frac{1}{2} 2 (d_j - o_j) \frac{\partial (d_j - o_j)}{\partial o_j} = -(d_j - o_j)$$

Derivation (2)

Since $o_j = \sigma(net_j)$ derivative is $\sigma(net_j)(1 - \sigma(net_j))$ for the logistic function

This yields

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(\text{net}_j)}{\partial net_j} = o_j(1 - o_j)$$

Substituting

$$\frac{\partial Err_{\mathbf{x}}}{\partial net_j} = -(d_j - o_j)o_j(1 - o_j)$$

Yielding

$$\Delta w_{ji} = -\frac{\eta \partial Err_{\mathbf{x}}}{\partial w_{ji}} = \eta(d_j - o_j)o_j(1 - o_j)x_{ji}$$

Derivation (3)

Case 2: Hidden Nodes

Let $\delta_i = -\partial Err_{\mathbf{x}} / \partial net_i$

Apply the chain rule

$$\frac{\partial Err_{\mathbf{x}}}{\partial net_j} = \sum_{k \in \text{downstream}(j)} \frac{\partial Err_{\mathbf{x}}}{\partial net_k} \cdot \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in \text{downstream}(j)} \delta_k \cdot \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in \text{downstream}(j)} \delta_k \cdot \frac{\partial net_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}$$

Derivation (4)

Continuing

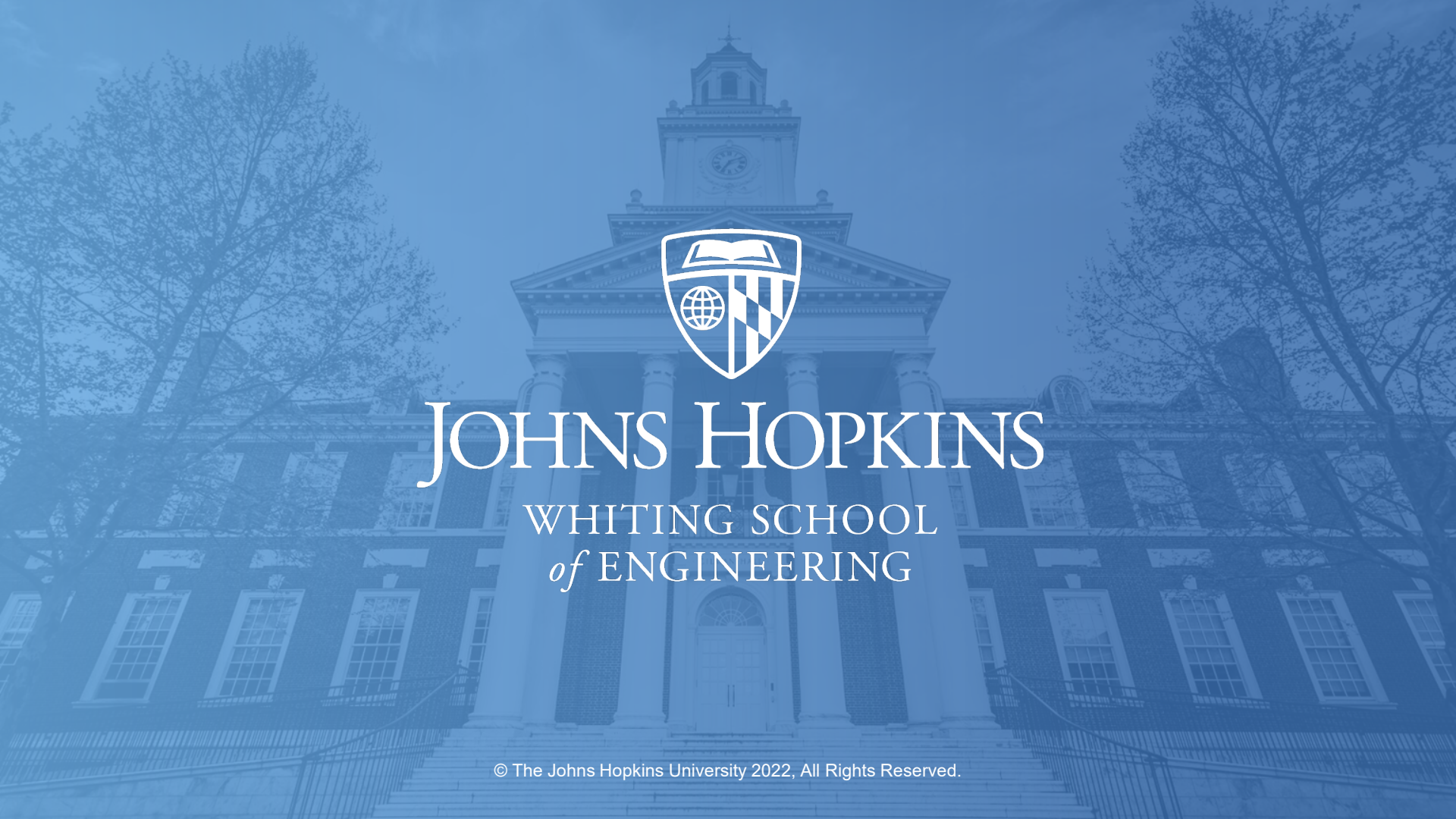
$$\frac{\partial Err_{\mathbf{x}}}{\partial net_j} = \sum_{k \in \text{ downstream } (j)} \delta_k w_{kj} \cdot \frac{\partial o_j}{\partial net_j}$$

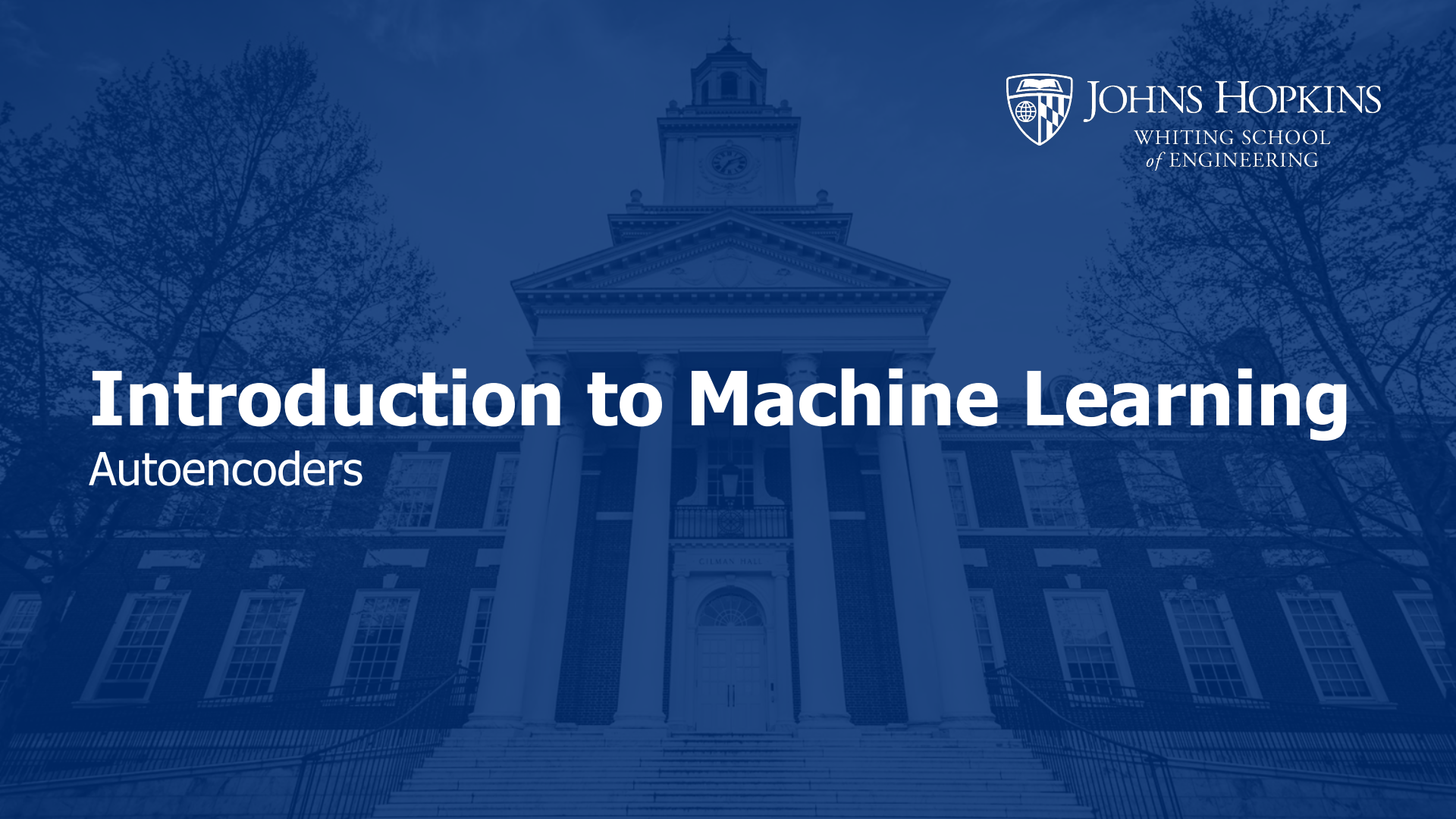
$$= \sum_{k \in \text{ downstream } (j)} \delta_k w_{kj} o_j (1 - o_j)$$

$$\delta_j = o_j (1 - o_j) \sum_{k \in \text{ downstream } (j)} \delta_k w_{kj}$$

Result: $\Delta w_{ji} = \eta \delta_j x_{ji}$

The Delta Rule





Introduction to Machine Learning

Autoencoders

Autoencoders

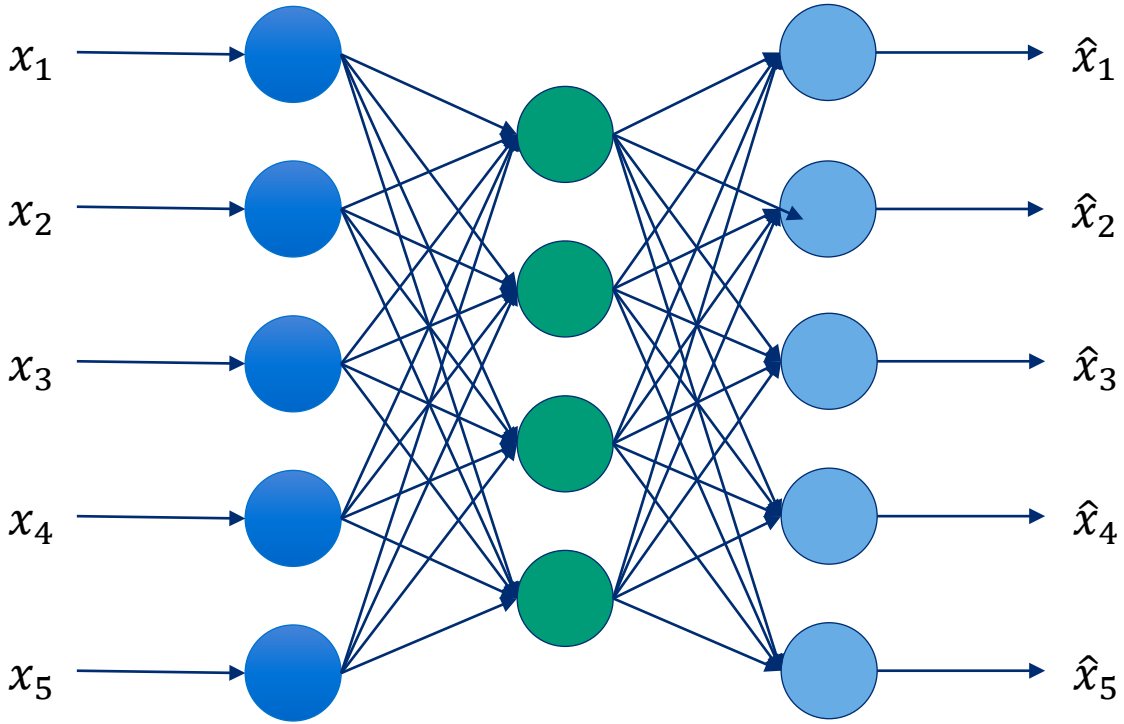
- An **autoencoder** is a feedforward network trained to reproduce the input at the output.
- Consist of two parts:
  - Encoder $\mathbf{y} = g(\mathbf{x})$ mapping the input to some embedding feature space
  - Decoder $\hat{\mathbf{x}} = f(\mathbf{y})$ mapping the embedded pattern back to the input space
- The goal is to minimize the reconstruction loss.

$$\min \mathcal{L}(\mathbf{x}, f(g(\mathbf{x})))$$

- Reconstruction loss is typically defined as

$$\mathcal{L}(\mathbf{x}, f(g(\mathbf{x}))) = \|\mathbf{x} - f(g(\mathbf{x}))\|_2^2 = \left(\sum_{i=1}^{d} |x_i - f_i(g(\mathbf{x}))|^2 \right)^{1/2}$$

Autoencoders (cont.)



Undercomplete Autoencoders

- The main goal is to learn a new feature space
- Often, want the feature space to be smaller (dimensionality reduction)

$$\text{Require } |g(\mathbf{x})| < |\mathbf{x}|$$

- If $|g(\mathbf{x})| = |\mathbf{x}|$ simplest mapping results when

$$w_{ij}^{enc} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}, \text{ and } w_{ij}^{dec} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

- Suppose the decoder is linear: Learns same subspace as PCA
- Nonlinear encoders and decoders similar to kernel PCA

Sparse Autoencoders

- The main goal is still to learn a new feature space
- No constraints on the size of the encoding layer
- Modify loss function

$$\mathcal{L}(\mathbf{x}, f(g(\mathbf{x}))) = \|\mathbf{x} - f(g(\mathbf{x}))\|_2^2 + \lambda\Omega(\text{enc}(\mathbf{W}))$$

- The function $\Omega(\text{enc}(\mathbf{W}))$ acts as a penalty on the encoder function itself
- The parameter λ sets the strength of the regularizer
- For sparseness, common to set

$$\Omega(\text{enc}(\mathbf{W})) = \sum_{\forall w_{ij} \in \text{enc}(\mathbf{W})} |w_{ij}|$$

Denoising Autoencoders

- The main goal becomes removing noise from a noisy input vector
- The idea is to train the AE on noisy data with the target being the noise-free version
- Thus the loss function becomes

$$\mathcal{L}(\mathbf{x}, f(g(\tilde{\mathbf{x}}))) = \|\mathbf{x} - f(g(\tilde{\mathbf{x}}))\|_2^2$$

  where $\tilde{\mathbf{x}}$ is the noisy input and $\mathbf{x}$ is the desired, noise-free output
- A common combination is the "sparse denoising autoencoder," which provides added freedom to learn the desired mapping by increasing the embedding space while regularizing to prevent overfitting to the noise

Orthogonal Autoencoders

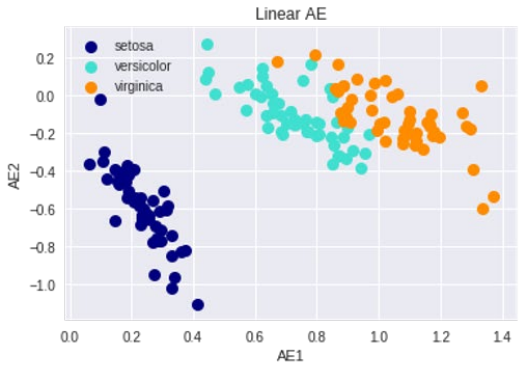
- Return to dimensionality reduction
- Suppose we wish to find an embedding where the corresponding learned features are decorrelated
- Apply a regularization term to enforce orthogonality
- Note that if $\mathbf{W}^\top \mathbf{W} = \mathbf{I}$ then $\mathbf{W}$ must be column orthogonal
- Let $\widehat{\mathbf{W}} = \mathbf{W}^\top \mathbf{W}$
- Define our regularizer as

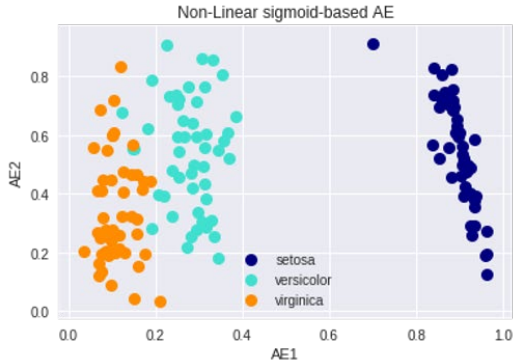
$$\Omega(\text{enc}(\mathbf{W})) = \frac{1}{\left|\mathbf{w}^k\right|^2} \sum_{i=1}^{k} \sum_{j=1}^{k} \left| I_{ij} - \widehat{w}_{ij} \right|$$

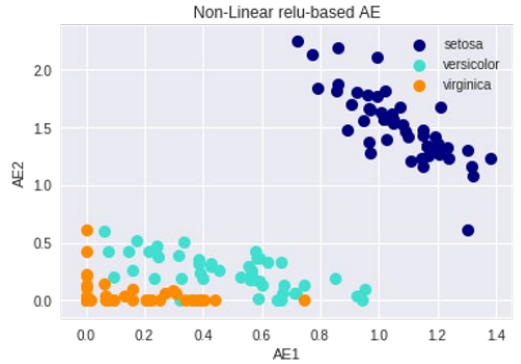
- The result should drive the weight vectors in the encoding layer to be orthogonal

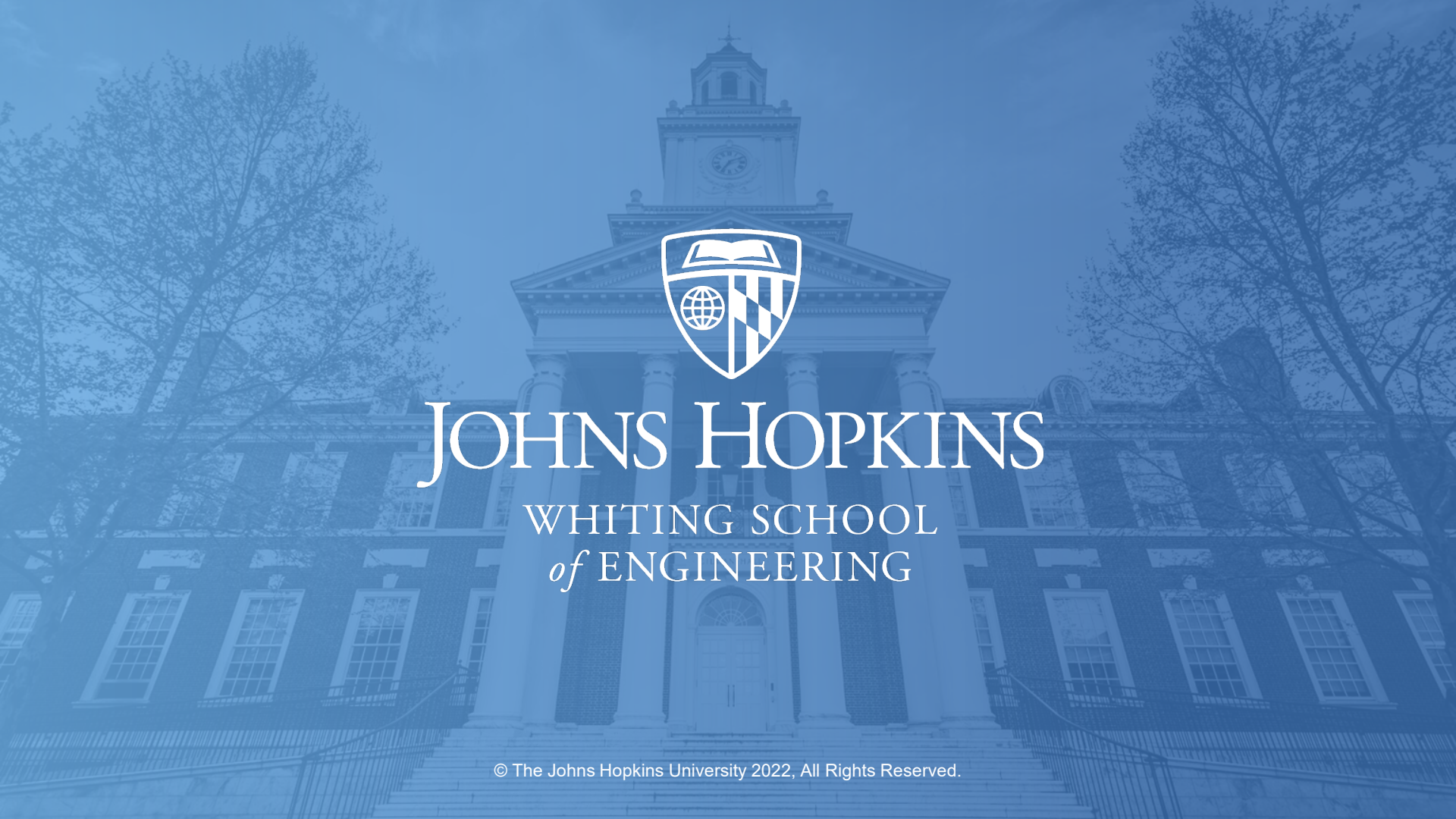
Orthogonal AEs vs PCA

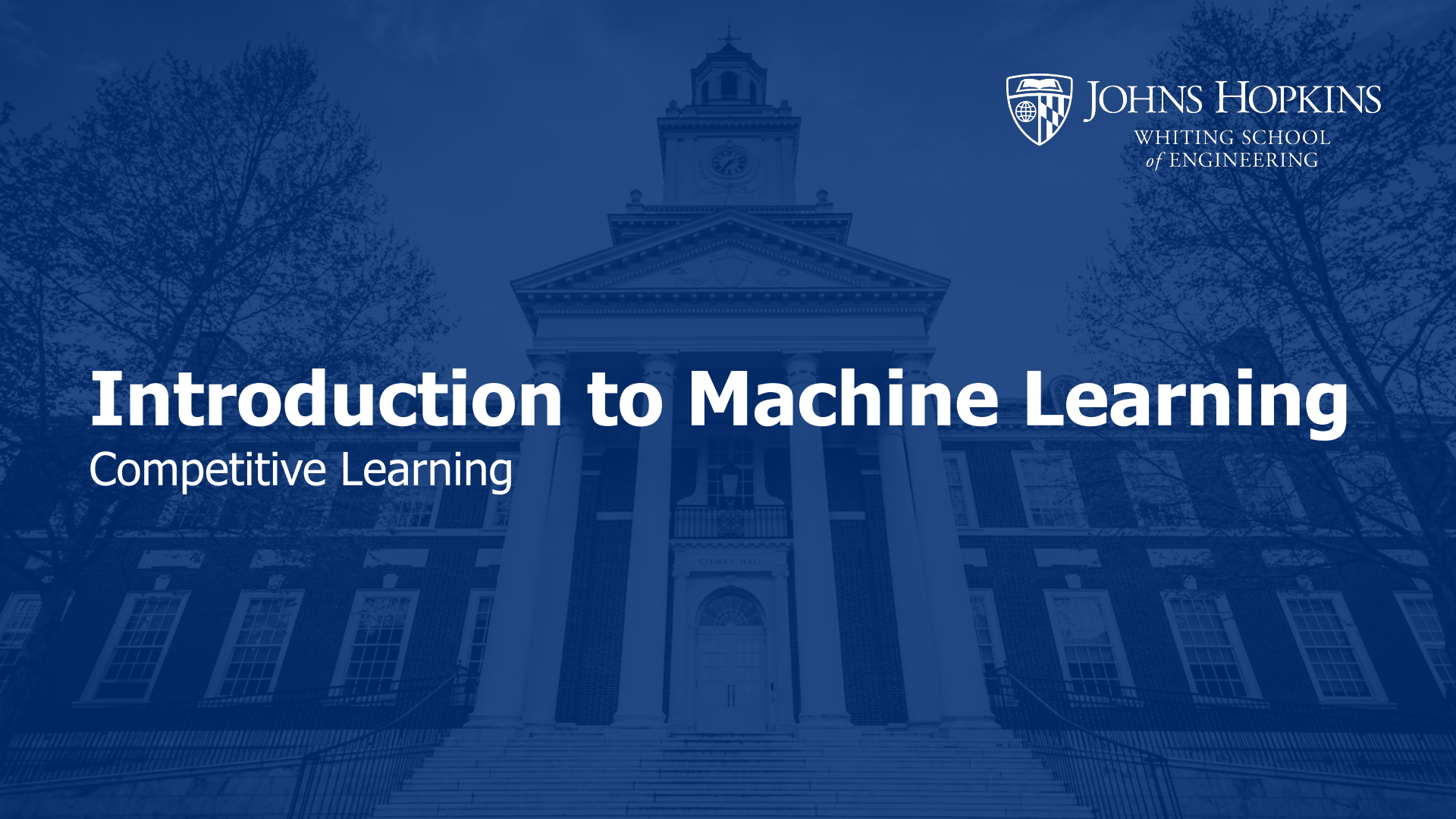
- Recall that a linear encoder covers PCA space
- The orthogonal regularizer zeroes in on principal components
- Nonlinear AEs can also find effective feature mappings
  - Sigmoid activation
  - ReLU activation











Introduction to Machine Learning

Competitive Learning

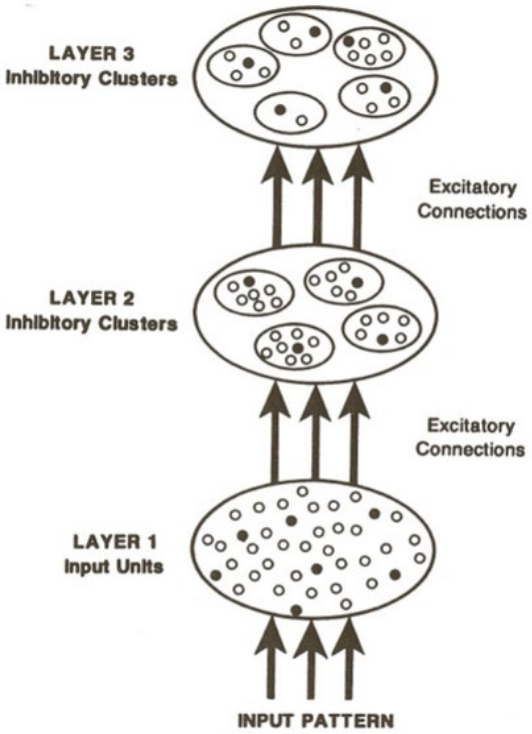
Components of Competitive Learning

- All of the units in the network are of the same type, except for random parameters values
- The "strength" of the neurons will be limited to control activation level
- Competition is introduced, introducing "inhibitory" connections among output nodes

Feature Detectors

Paradigms of Competitive Learning

1. Classification
2. Auto Association
3. Pattern Association
4. **Regularity Detection**



Rumelhart, D. E. and Zipser, D. (1985). Feature discovery by competitive learning. *Cognitive Science*, 9:75–112.

Building the Network

- Non-overlapping clusters
  - Apply “winner take all” principle (inhibitors)
  - Fire as a one (winner) or zero (inhibited)
- Complete bipartite graph between layers
  - Inputs have values of zero or one
  - Values are propagated as weighted sum
  - Competition takes place among output nodes
  - Results in nodes identifying clusters

Training the Network

- Initialize weights to positive random variables
- Normalize weights to an output node to sum to one

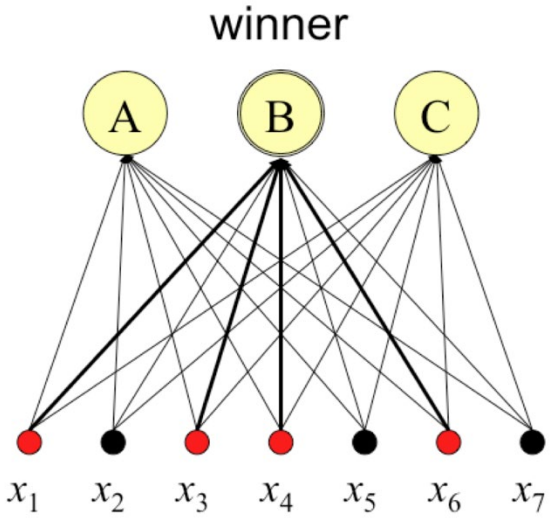
$$w'_{ik} = \frac{w_{ik}}{\sum_j w_{jk}}$$

- Update rule

$$\Delta w_{ij} = \begin{cases} 0 & \text{if unit } j \text{ loses on stimulus } k \\ g\frac{c_{ik}}{n_k} - g w_{ij} & \text{if unit } j \text{ wins on stimulus } k \end{cases}$$

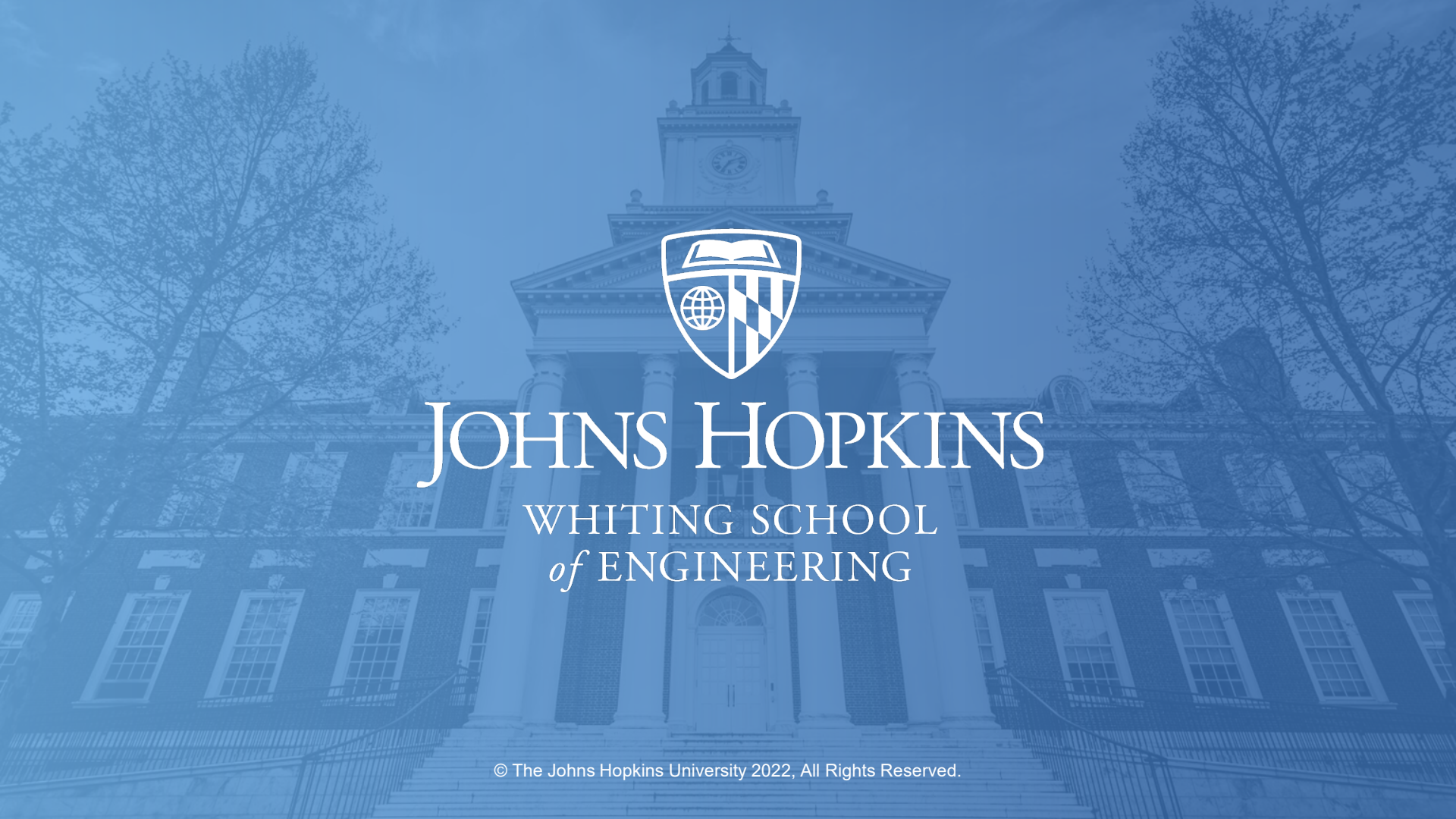
$$n_k = \sum_i c_{ik}$$

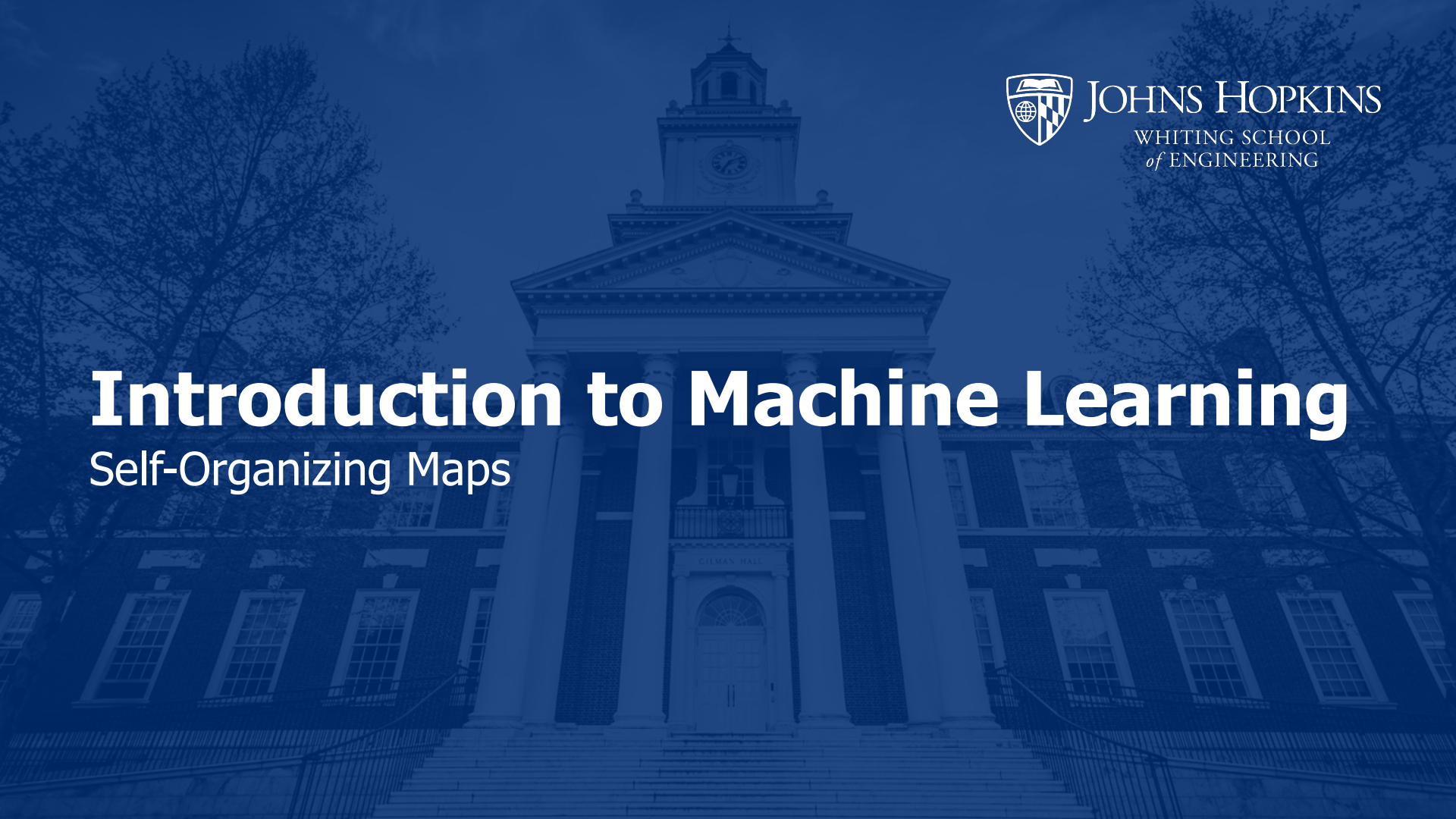
Example



$$\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\top \mathbf{b} = \sum_i a_i b_i$$

$$\cos\theta = \frac{\langle \mathbf{x}, \mathbf{w} \rangle}{\|\mathbf{x}\| \cdot \|\mathbf{w}\|}$$

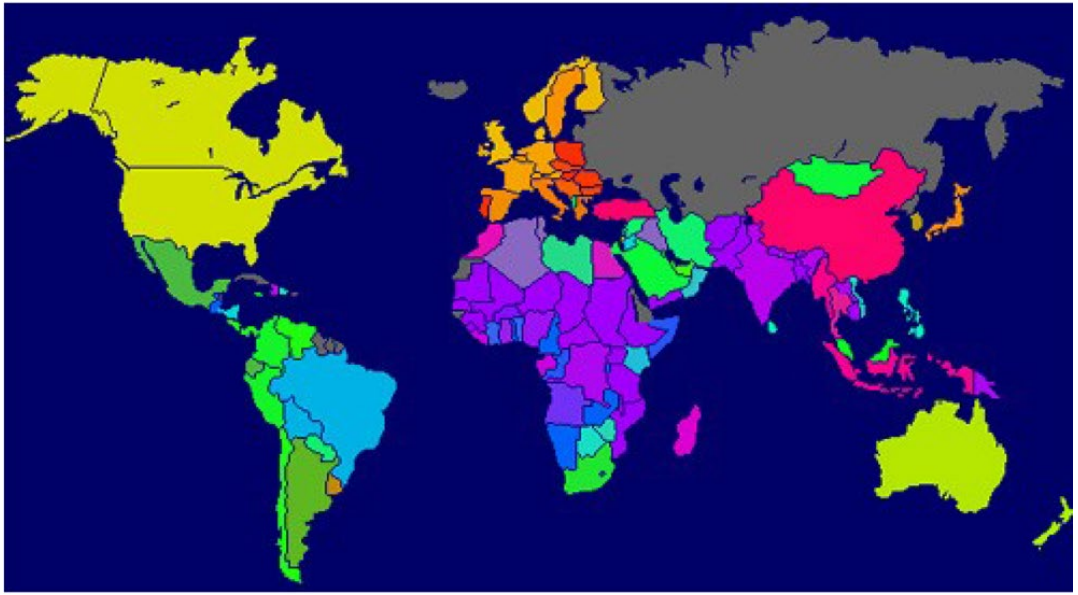




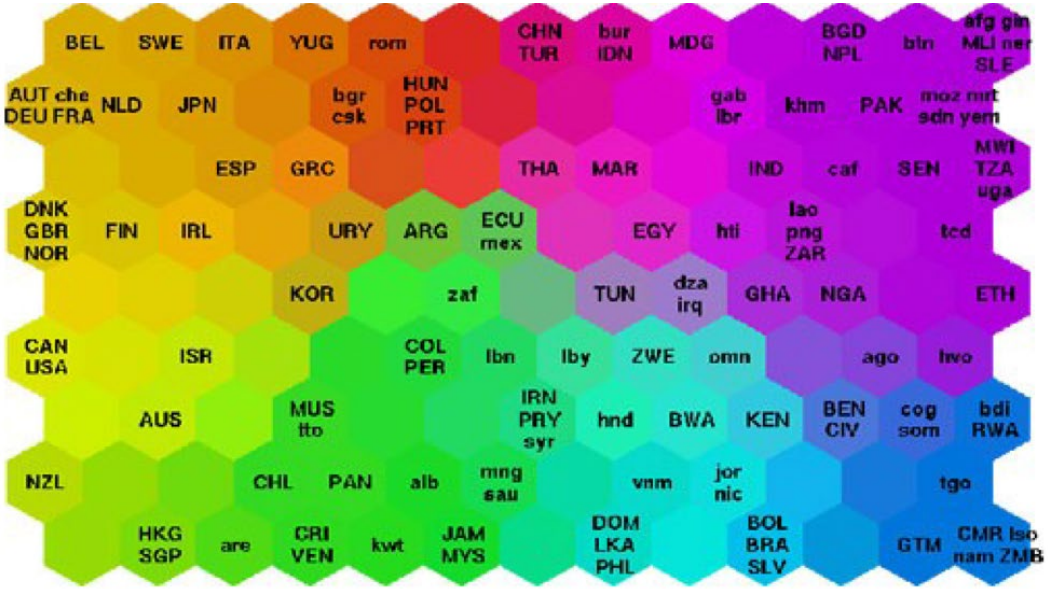
Introduction to Machine Learning

Self-Organizing Maps

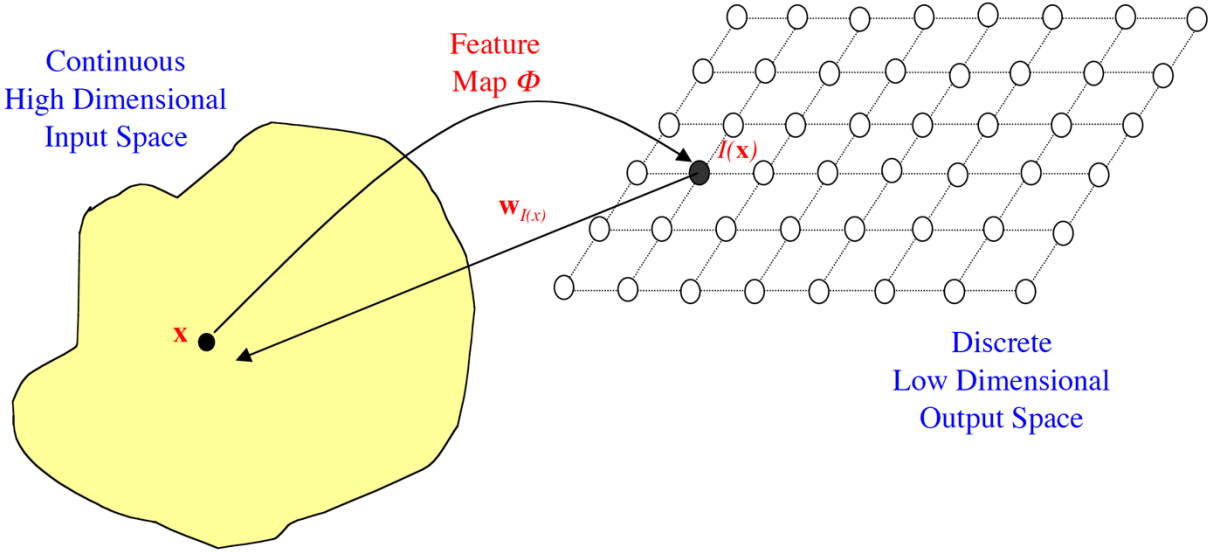
Example



Example (cont.)



Feature Map


Continuous
High Dimensional
Input Space
Feature
Map Φ
I(x)
w I(x)
x
Discrete
Low Dimensional
Output Space

SOM Characteristics

- Every input node is connected to every output node (complete bipartite graph)
- Output nodes are connected spatially – define a neighborhood
- Apply competitive learning – winner-take-all principle
- Update the weights to the winning node, and to its immediate neighbors

Training a SOM

- **Initialization:** Use random weights; need not sum to one.
- **Competition:** Compare weight vectors to input vectors.
  - Euclidean Distance
  - Inner Product
- **Cooperation:** Identify weight vectors within neighborhood based on distance of weights.
- **Adaptation:** Adjust the weights of winning node and neighbors.

Competition

- Define a discriminant function.

$$\delta_j(\mathbf{x}) = \sum_{i=1}^{d} (x_i - w_i)^2$$

- This is just Euclidean distance (without the square root).
- Maps continuous input space to discrete output space defined by the output nodes.
- Winner is node with smallest distance.

Cooperation

- Define a "topological" neighborhood.
- Let S_{ij} be the lateral distance between neurons *i* and *j*.
- Define the neighborhood as follows:

$$\mathcal{T}_{j,I(\mathbf{x})} = \exp\left(-\frac{S_{j,I(\mathbf{x})}^2}{2\sigma^2}\right)$$

where $I(\mathbf{x})$ is the winning neuron

Cooperation – Properties

- Maximal at the winning neuron
- Decays monotonically with distance
- Translation invariant
- Corresponds to a radial basis (RBF) function kernel
- Neighborhood size $\sigma(t)$

$$\sigma(t) = \sigma(0) \exp\left(-\frac{t}{\tau_\sigma}\right)$$

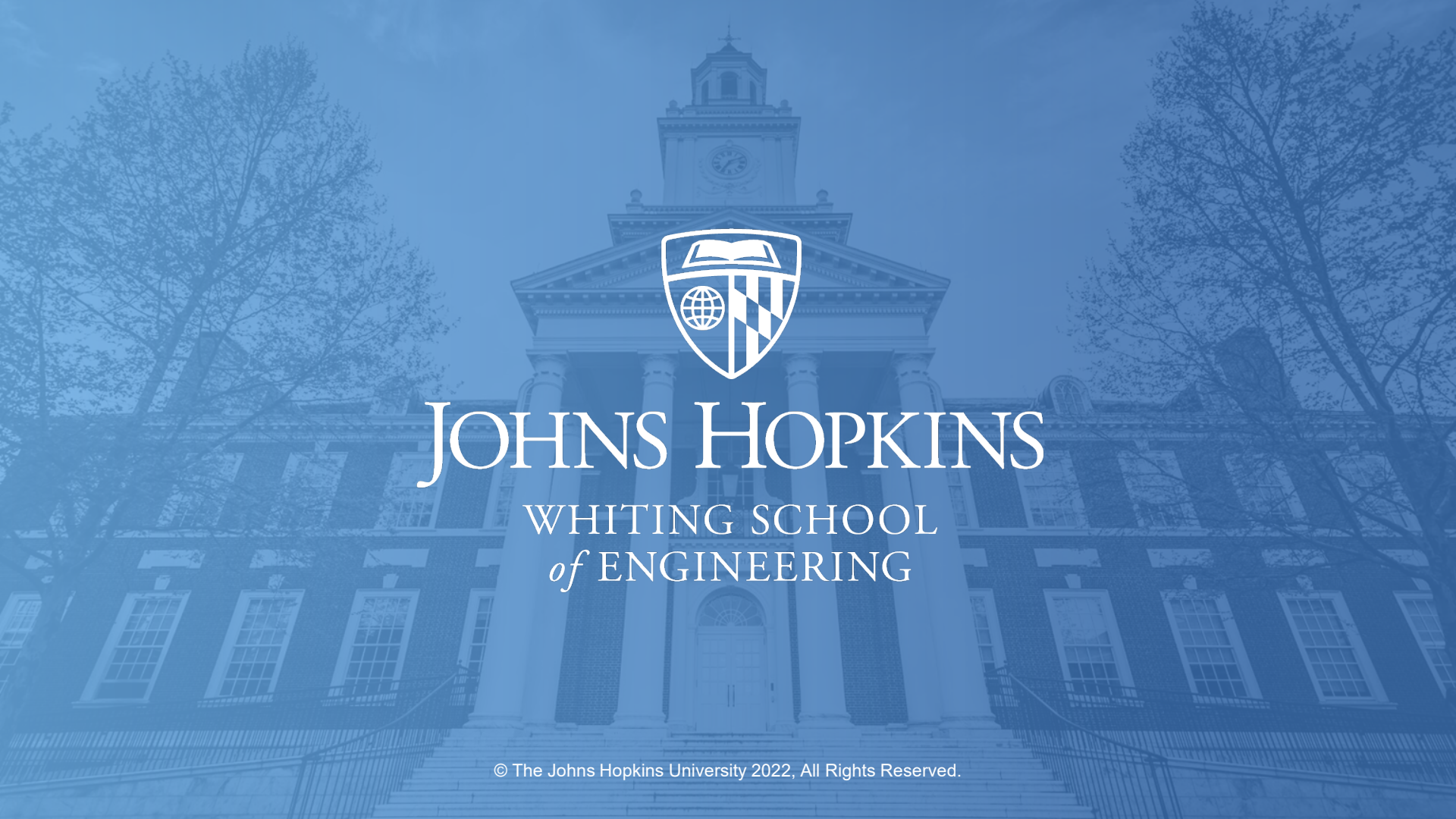
Adaptation

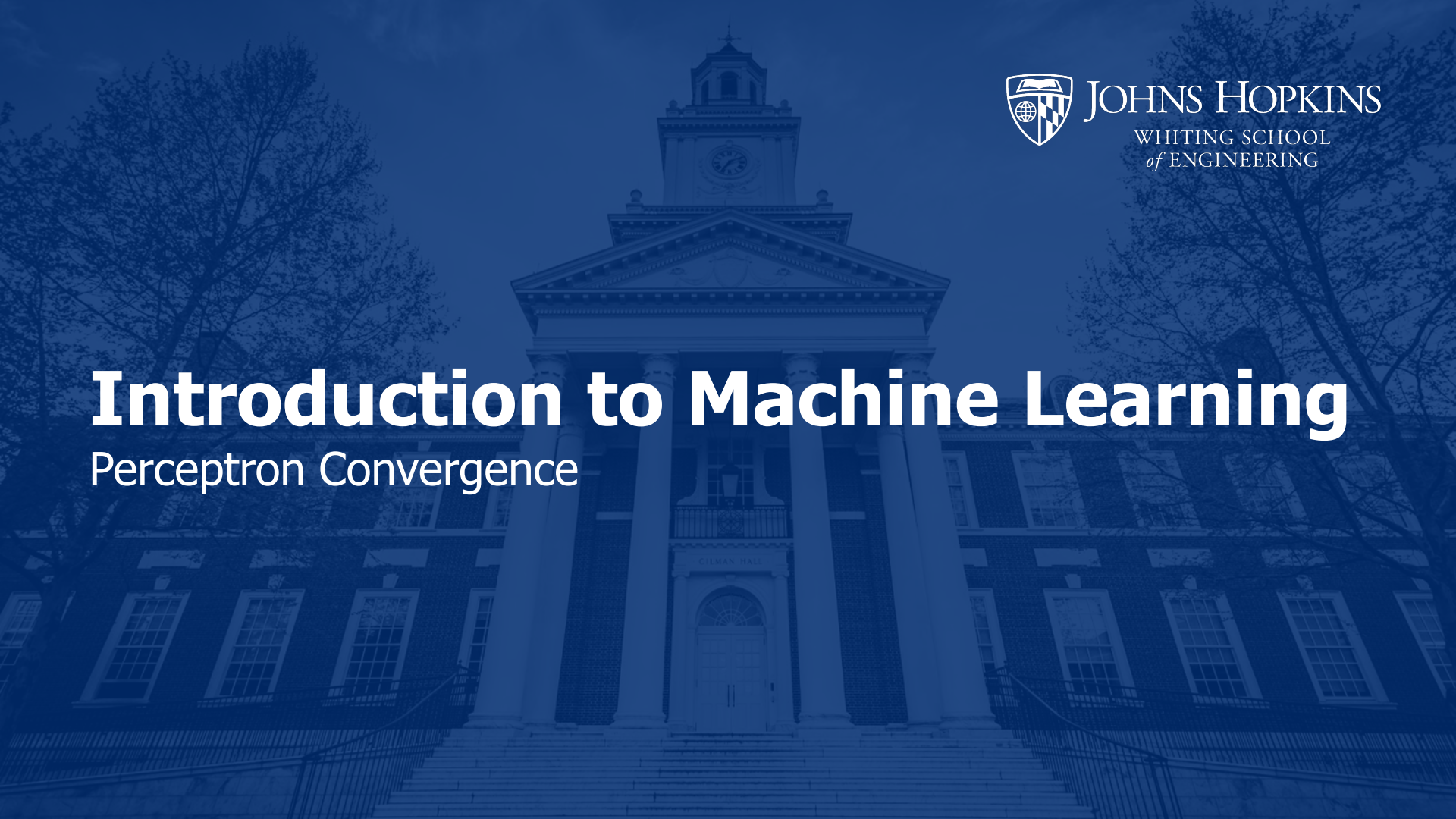
- Update winning node.
- Update neighbors of the winning node.
- Adjust the weights based on distance from winning node.

$$\Delta w_{ji} = \eta(t) \cdot \mathcal{T}_{j,I(\mathbf{x})}(t) \cdot (x_i - w_{ji})$$

- Update learning rate.

$$\eta(t) = \eta(0) \exp\left(-\frac{t}{\tau_\eta}\right)$$





Introduction to Machine Learning

Perceptron Convergence

Perceptron Training (Recap)

- Recall training set $\mathbf{F} = \mathbf{F}^+ \cup \mathbf{F}^-$
- Start by considering $\mathbf{x} \in \mathbf{F}^- : \mathbf{x} \leftarrow -\mathbf{x}$

1. Choose initial weight vector $\mathbf{w}$ at random
2. Choose some example $\mathbf{x} \in \mathbf{F}$
3. Consider the $\text{class}(\mathbf{x})$
   - If $\text{sgn}\left(\mathbf{w}^\top \mathbf{x}\right) \leq 0$ then $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}$
   - Otherwise, do nothing (class is correct)
4. Repeat steps 2 and 3 until convergence

Perceptron Convergence Theorem

Theorem: Let **F** be a set of unit-length vectors, and let **w** be an initial weight vector. If there exists a unit vector $\mathbf{w}'$ and a number $\delta > 0$ such that $\mathbf{w}'^{\top}\mathbf{x} > \delta$ for all $\mathbf{x} \in \mathbf{F}$ then the perceptron learning rule will update **w** only a finite number of times to find such a $\mathbf{w}'$

Convergence Proof (1)

- Recall that $\mathbf{w}'$ is an arbitrary weight vector.
- Since it's a unit vector, $\|\mathbf{w}'\| = 1$
- Define $g(\mathbf{w}) = \dfrac{\mathbf{w}'^{\top}\mathbf{w}}{\|\mathbf{w}\|} \leq 1$
- Assume, wlg, we have misclassified a negative example.
- Consider, first, the numerator of $g(\mathbf{w})$

$$\begin{aligned} \mathbf{w}'^{\top}\mathbf{w}_{t+1} &= \mathbf{w}'^{\top}(\mathbf{w}_t + \mathbf{x}) \\ &= \mathbf{w}'^{\top}\mathbf{w}_t + \mathbf{w}'^{\top}\mathbf{x} \end{aligned}$$

- Then $\mathbf{w}'^{\top}\mathbf{x} > \delta \Rightarrow \mathbf{w}'^{\top}\mathbf{w}_{t+1} > \mathbf{w}'^{\top}\mathbf{w}_t + \delta$
- Then for n updates, $\mathbf{w}'^{\top}\mathbf{w}_{t+1} > n\delta$

Convergence Proof (2)

- Now consider the denominator of $g(\mathbf{w})$
- We only update when $\mathbf{w}^\top \mathbf{x} \leq 0$
- Thus

$$\begin{aligned} \|\mathbf{w}_{t+1}\|^2 &= \mathbf{w}_{t+1}^\top \mathbf{w}_{t+1} \\ &= (\mathbf{w}_t + \mathbf{x})^\top (\mathbf{w}_t + \mathbf{x}) \\ &= \|\mathbf{w}_t\|^2 + 2\mathbf{w}_t^\top \mathbf{x} + \|\mathbf{x}\|^2 \end{aligned}$$

- Since $\mathbf{w}_t^\top \mathbf{x} \leq 0$ and $\|\mathbf{x}\|^2 \leq 1$ we must also have that $\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + 1$
- Again, after *n* updates, $\|\mathbf{w}_n\|^2 \leq n \Rightarrow \frac{1}{\|\mathbf{w}_n\|^2} \geq \frac{1}{n}$

Convergence Proof (3)

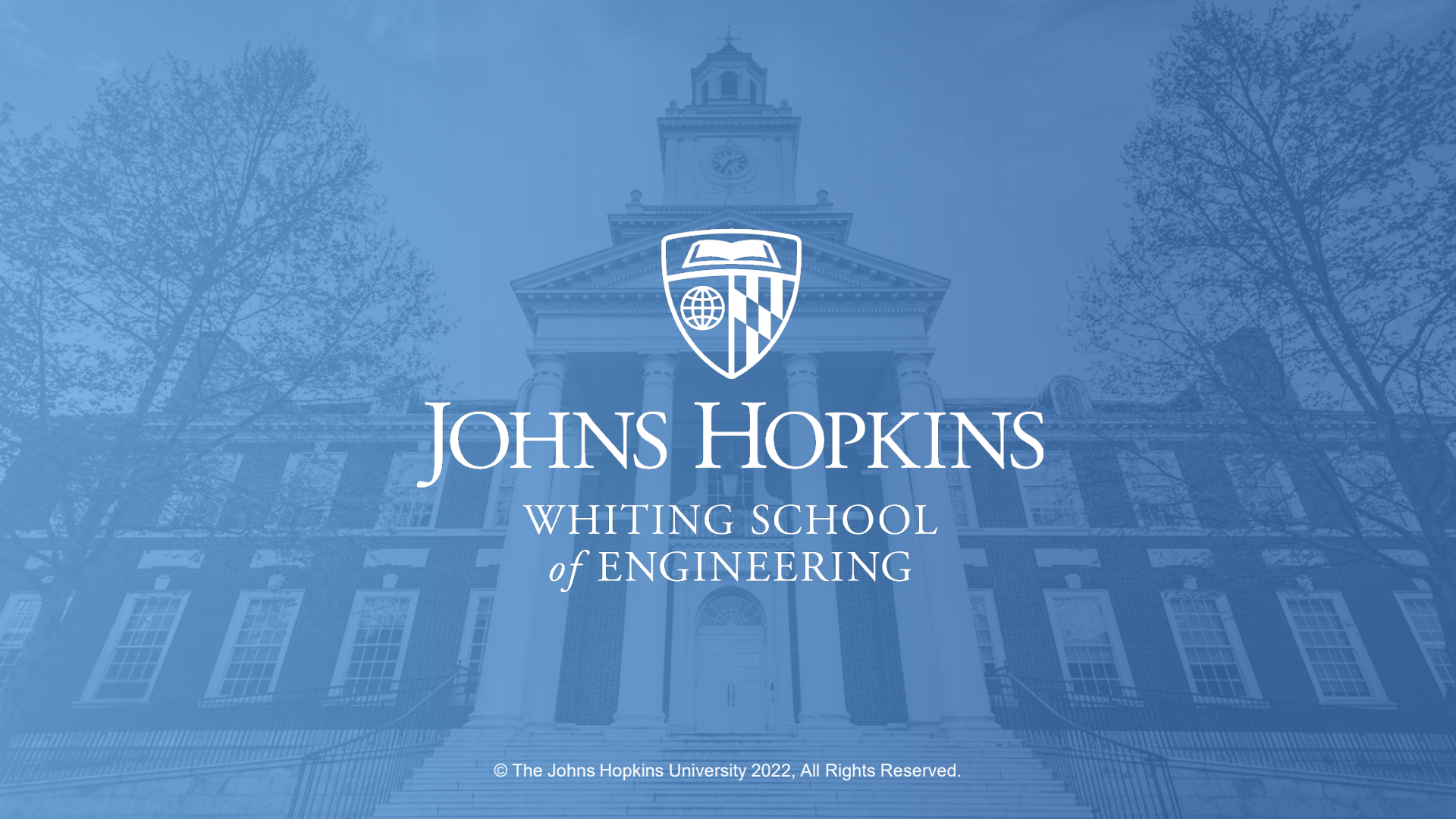
- Combining the results for the numerator and the denominator:

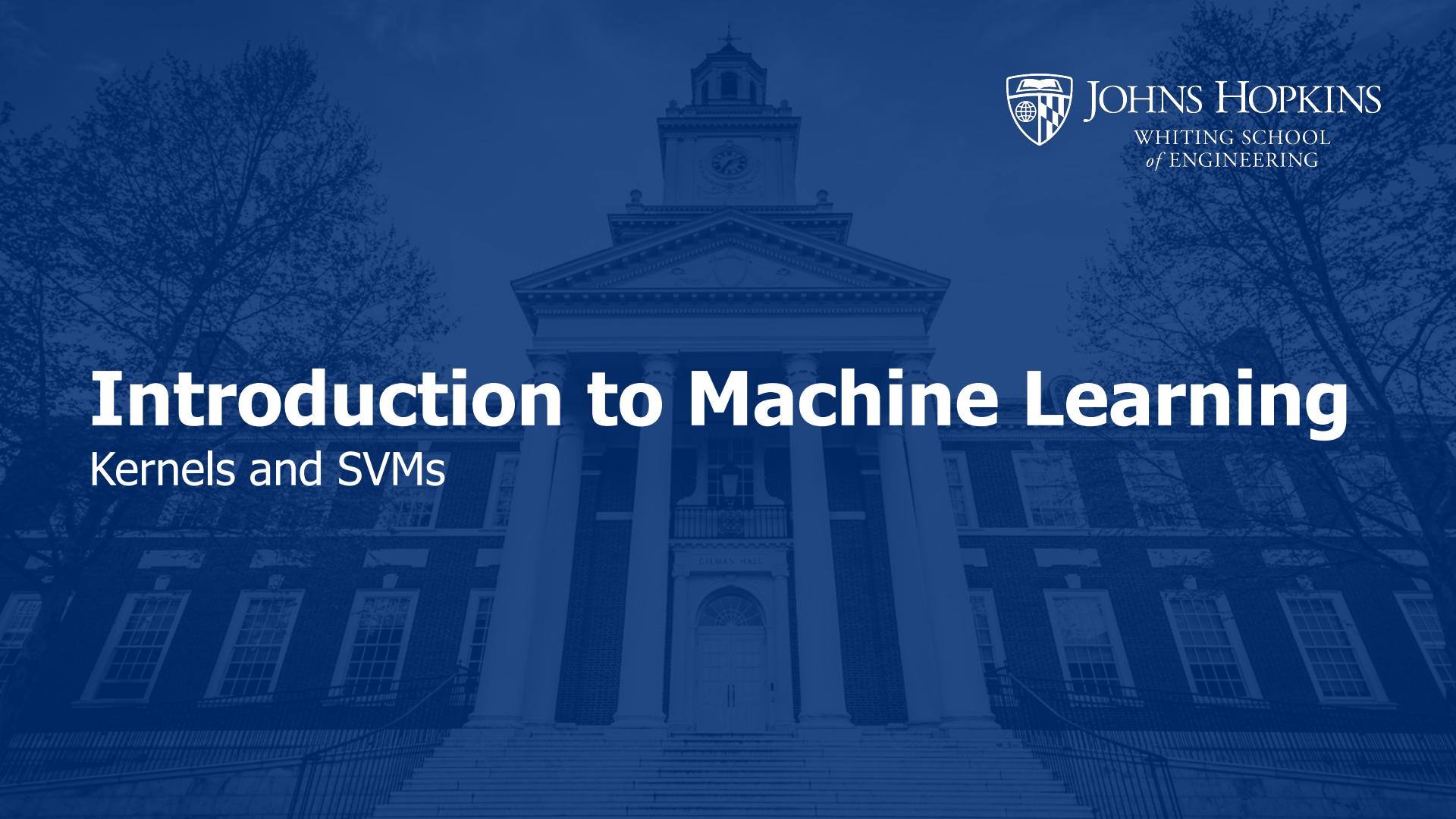
$$\begin{aligned} g(\mathbf{w}_n) &= \frac{\mathbf{w}'^{\top}\mathbf{w}_n}{\|\mathbf{w}_n\|} \\ &> \frac{n\delta}{\sqrt{n}} \\ &= \delta\sqrt{n}. \end{aligned}$$

- Since $g(\mathbf{w}_n) \leq 1$ this requires that $\delta\sqrt{n} \leq 1$
- This gives us

$$n \leq \frac{1}{\delta^2}$$

Q.E.D.



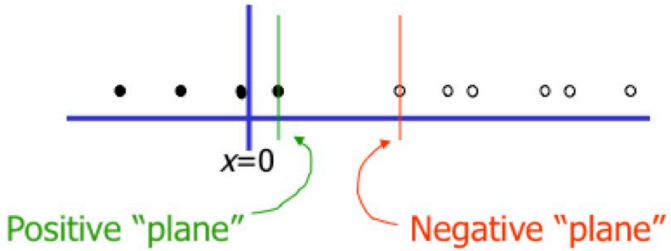


Introduction to Machine Learning

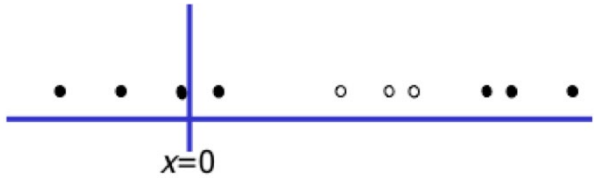
Kernels and SVMs

Limitation of the SVM

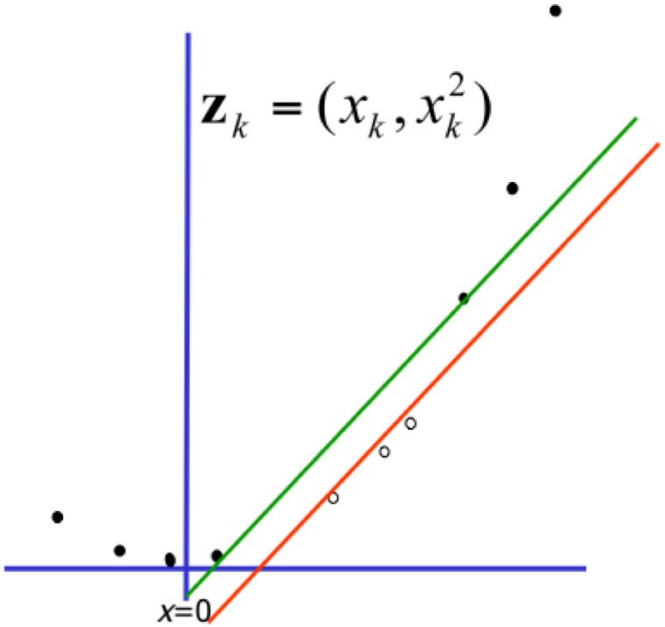
- SVMs are linear classifiers.
- A linearly separable problem.



- A nonlinearly separable problem.



Increasing Dimensionality



The Kernel Trick

- Most linear classifiers rely on dot products.
- Often relies on a distance-based calculation.
- A kernel is defined as a function of two vectors:

$$K\left(\mathbf{x}_i, \mathbf{x}_j\right) = \mathbf{x}_i^\top \mathbf{x}_j$$

- Consider mapping to a higher dimensional space: $\phi : \mathbf{X} \rightarrow \phi(\mathbf{x})$
- Corresponding dot product becomes:

$$K_\phi\left(\mathbf{x}_i, \mathbf{x}_j\right) = \phi(\mathbf{x}_i)^\top \phi\left(\mathbf{x}_j\right)$$

Cover's Theorem

Theorem: There are $C(N, d)$ homogeneous linear dichotomies of N points in general position in Euclidean d-space, where

$$C(N, d) = 2\sum_{k=0}^{d-1} \binom{N-1}{k}$$

- When $N \leq d + 1$ exponential growth suggests increasing likelihood of linear separability.
- When $N < d + 1$ subexponential growth suggests decreasing likelihood of linear separability.

Theorem (informal): In a classification problem, if the data is cast into a high dimensional space nonlinearly, as long as the resulting space is not densely populated, the data becomes more likely to be linearly separable than in the lower-dimensional space.

Mercer's Theorem

Definition: A positive, semi-definite matrix **M** is a matrix such that, for all non-zero vectors $\mathbf{z} \in \mathbb{R}^n, \mathbf{x}^\top \mathbf{M}\mathbf{x} \geq 0$

Equivalently, a positive, semi-definite matrix **M** has all non-zero eigenvalues.

Definition: A positive, semi-definite kernel *K* on a set **S** is a function $K : \mathbf{S} \times \mathbf{S} \to \mathbb{R}$ such that $\forall n \in \mathbb{N}, \forall x_i \in \mathbf{S}$

$$\mathbf{C} = \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix}, c_{ij} = K(x_i, x_j)$$

Theorem: Any continuous, symmetric, positive, semi-definite kernel function $K(\mathbf{x}, \mathbf{y})$ can be expressed as a dot product in a higher-dimensional space.

- Linearize a nonlinear problem, by using a kernel to transform to a higher-dimensional space, thus increasing the probability of the data being linearly separable.

An Example

- Suppose we have vectors of the form $\mathbf{x} = [x_1 \quad x_2]^\top$
- Suppose our kernel is $K(\mathbf{x}_i, \mathbf{x}_j) = \left(1 + \mathbf{x}_i^\top \mathbf{x}_j\right)^2$
- Need to show that $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$

$$
\begin{aligned}
K(\mathbf{x}_i, \mathbf{x}_j) &= \left(1 + \mathbf{x}_i^\top \mathbf{x}_j\right)^2 \\
&= \left(1 + 2x_{i1}x_{j1}x_{i2}x_{j2} + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{j1} + 2x_{i2}x_{j2}\right) \\
&= \begin{bmatrix} 1 & x_{i1}^2 & \sqrt{2}x_{i1}x_{i2} & x_{i2}^2 & \sqrt{2}x_{i1} & \sqrt{2}x_{i2} \end{bmatrix}^\top \\
&\times \begin{bmatrix} 1 & x_{j1}^2 & \sqrt{2}x_{j1}x_{j2} & x_{j2}^2 & \sqrt{2}x_{j1} & \sqrt{2}x_{j2} \end{bmatrix}
\end{aligned}
$$

- Then $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$ where

$$\phi(x) = \begin{bmatrix} 1 & x_1^2 & \sqrt{2}x_1x_2 & x_2^2 & \sqrt{2}x_1 & \sqrt{2}x_2 \end{bmatrix}$$

Standard Kernels

- Polynomial Kernel

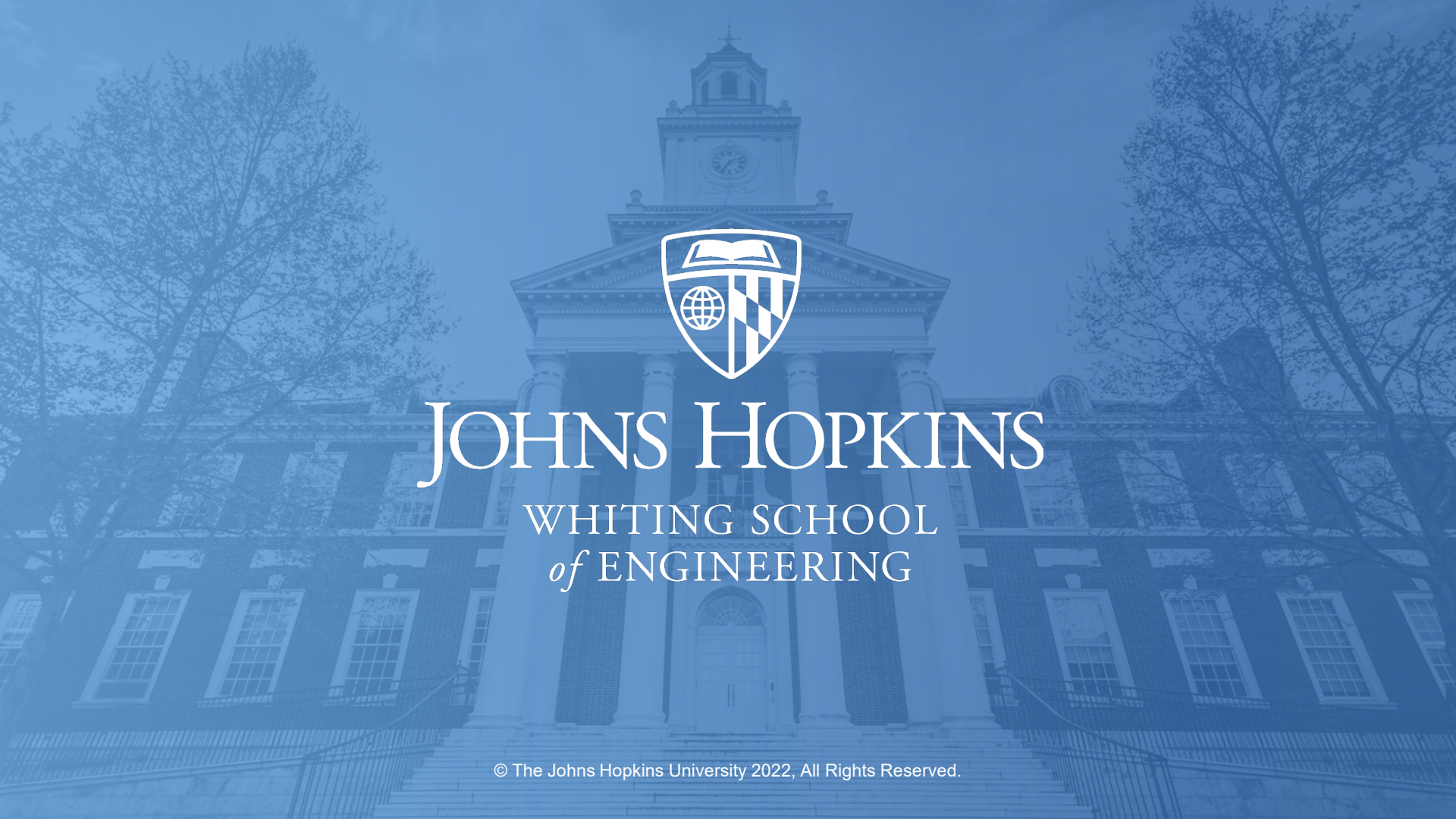
$$K_d\left(\mathbf{x}_i, \mathbf{x}_j\right) = \left(1 + \mathbf{x}_i^\top \mathbf{x}_j\right)^d$$

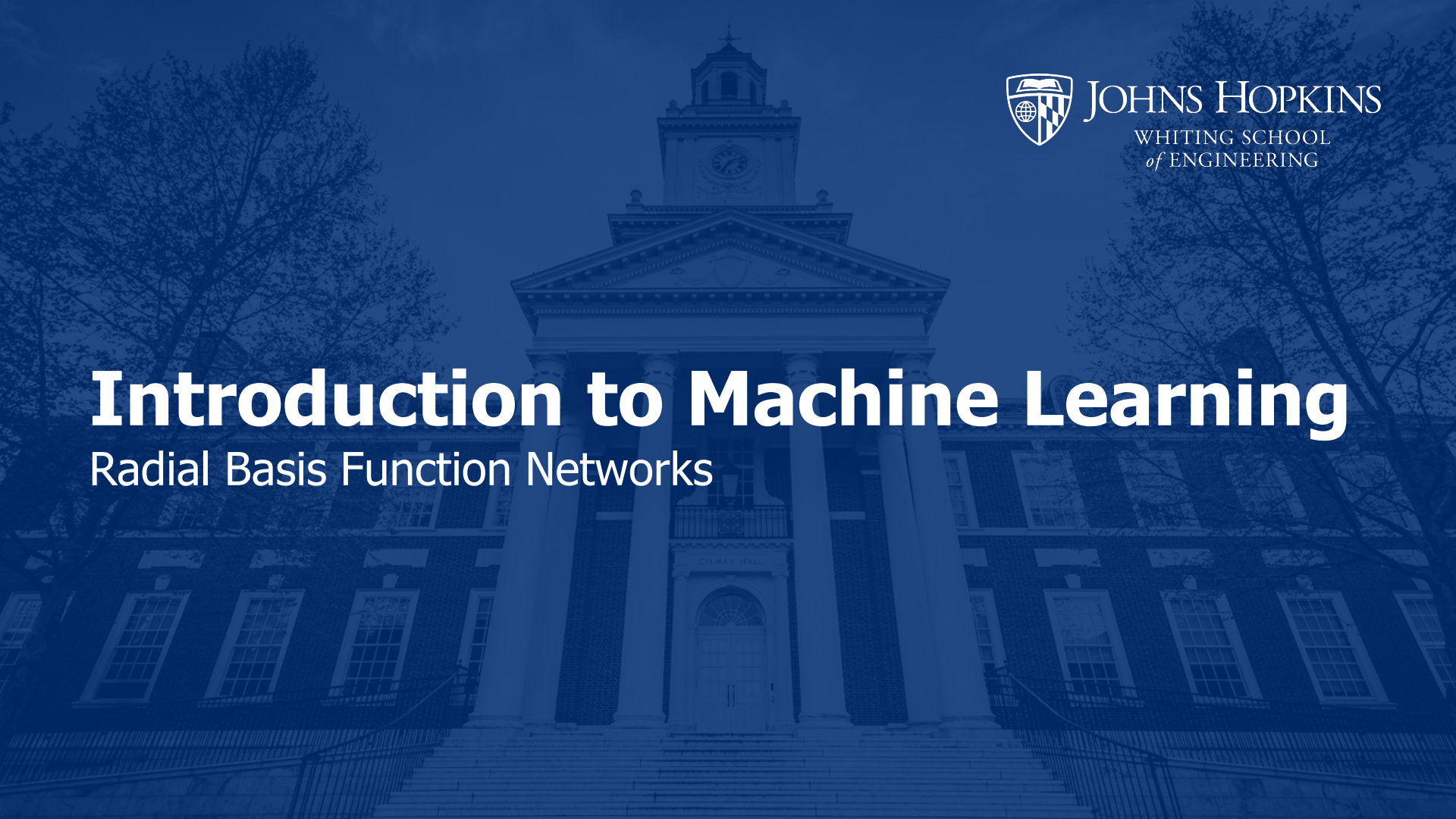
- Gaussian Kernel

$$K_{\mathcal{G}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \exp\left(-\frac{\left(\mathbf{x}_i - \mathbf{x}_j\right)^2}{2\sigma^2}\right)$$

- Sigmoid Kernel

$$K_\sigma\left(\mathbf{x}_i, \mathbf{x}_j\right) = \tanh\left(\kappa \mathbf{x}_i^\top \mathbf{x}_j - \delta\right)$$

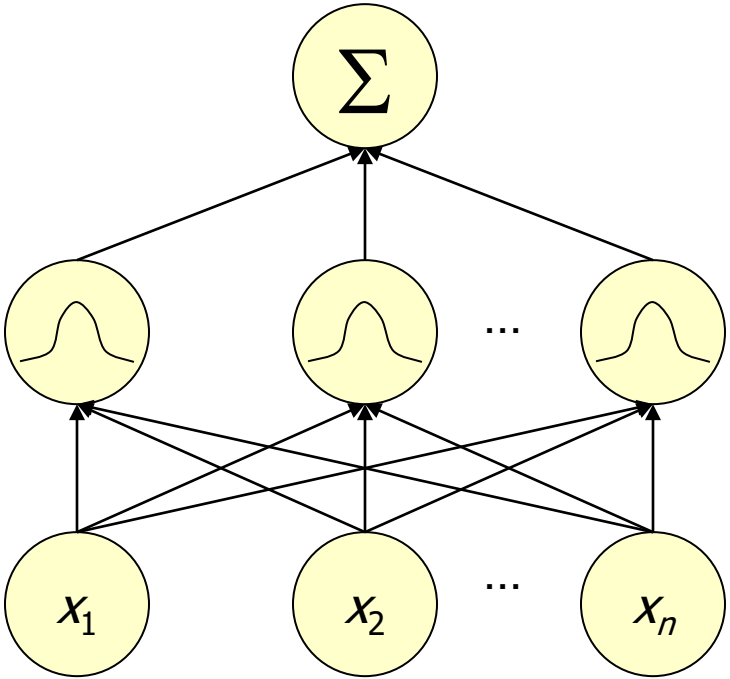




Introduction to Machine Learning

Radial Basis Function Networks

Radial Basis Function Network



The Radial Basis Function

- A symmetric function with "radial" symmetry.
- "On center, off surround" property of the visual cortex.

$$f(\mathbf{x}) = G(\|\mathbf{x} - \boldsymbol{\mu}\|)$$

- Gaussian Kernel

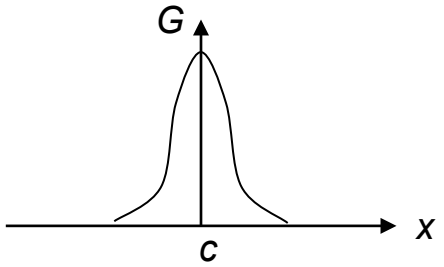
$$G(y) = \exp\left(-\frac{y^2}{\sigma^2}\right) \qquad \text{where } \sigma \text{ is spread}$$

- Usually, this is represented as

$$K(\mathbf{x}, \boldsymbol{\mu}) = \exp\left(-\gamma\|\mathbf{x} - \boldsymbol{\mu}\|^2\right) = \exp\left(-\frac{\|\mathbf{x} - \boldsymbol{\mu}\|^2}{2\sigma^2}\right) \qquad \text{where} \qquad \gamma = \frac{1}{2\sigma^2}$$

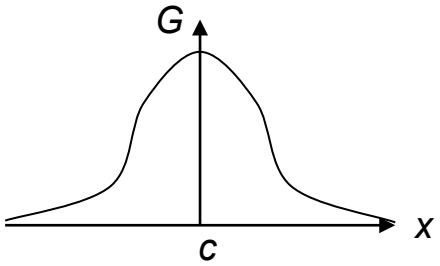
Spread/Selectivity

Small spread



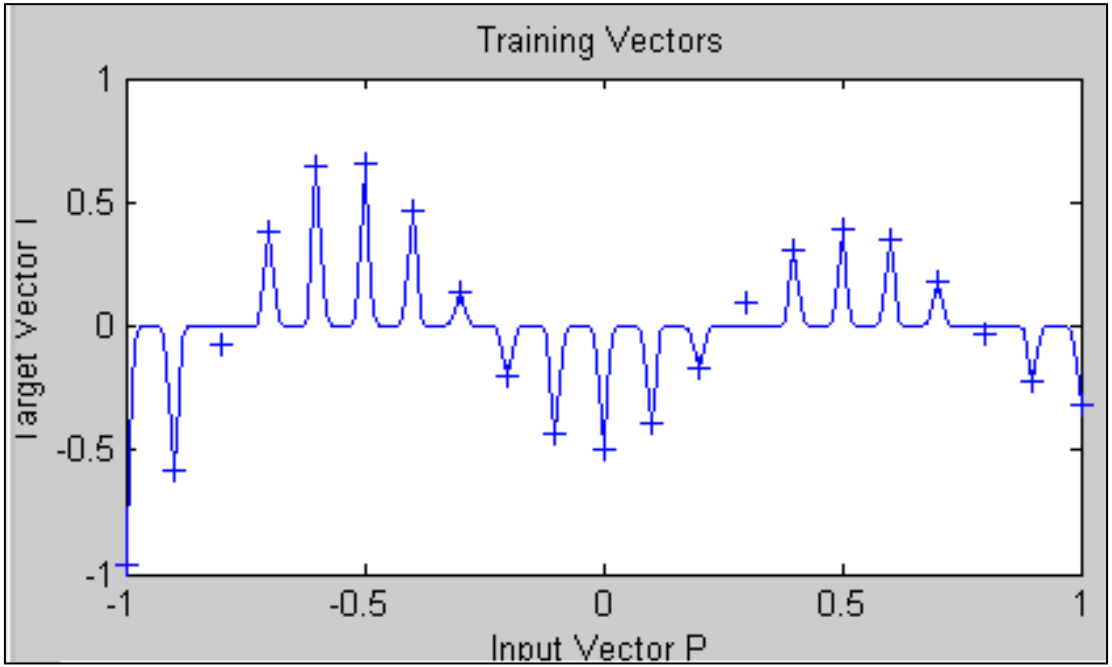
selective

Large spread

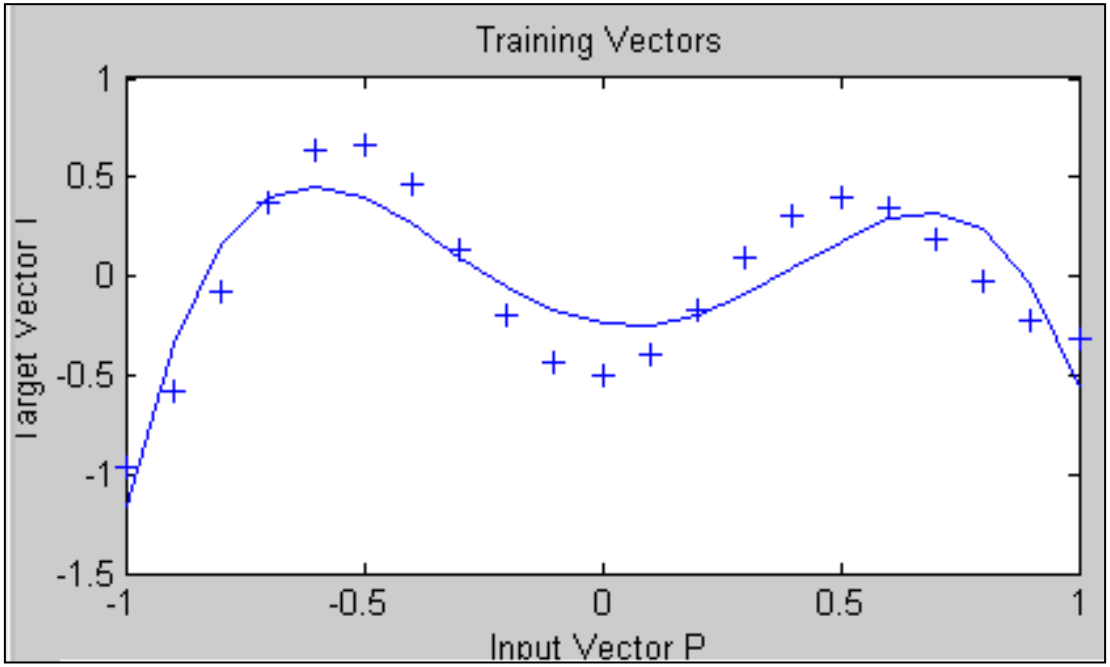


not selective

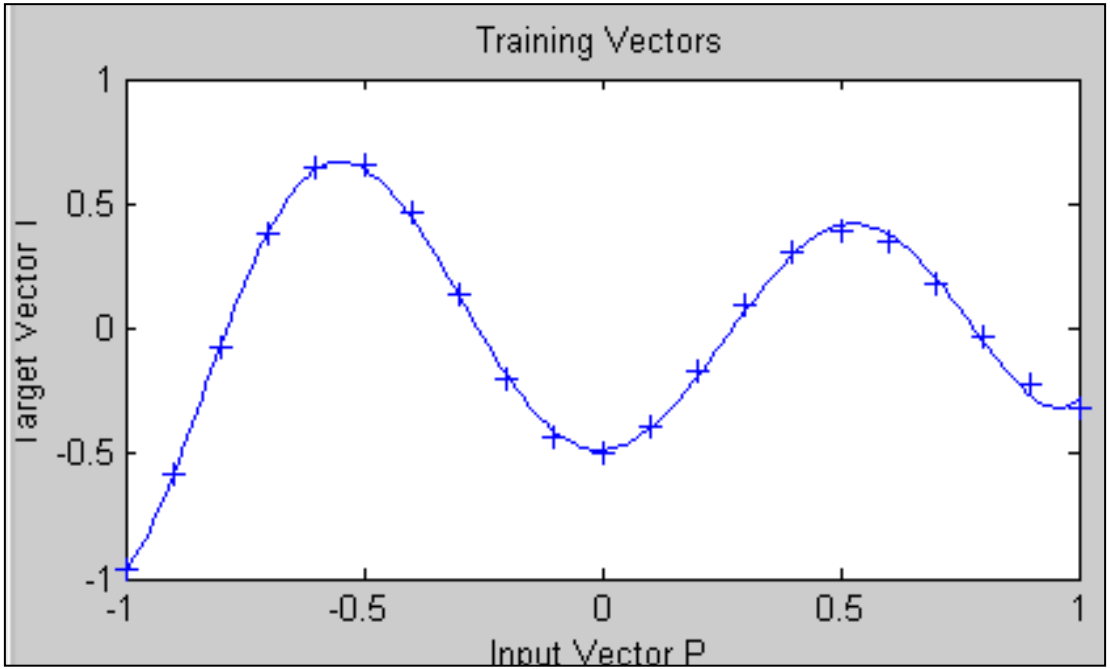
Small Spread



Large Spread



Ideal Spread



RBF Network

- Bridges the gap between
  - Neural networks (e.g., feedforward networks)
  - Kernel machines (e.g., SVMs)
  - Nonparametric/instance-based learning (e.g., nearest neighbor)
- Setting the spread
  - Tune a global parameter
  - Clustering the data and estimating cluster distributions
  - Train via gradient descent
- Linear combiner – gradient descent

Network Architecture

Three layers

1. Input Layer: One node per feature
2. Hidden Layer: $k \leq n$ nodes applying RPF kernel
3. Output layer:
   - Classification: One class per node
   - Regression: Single node

- The RBF kernel: $$K_j(\mathbf{x}, \boldsymbol{\mu}_j) = \exp\left(-\frac{1}{2\sigma_j^2}\|\mathbf{x} - \boldsymbol{\mu}_j\|^2\right)$$
- Then $$f(x) \approx \sum_{j=1}^{k} w_j K_j(\mathbf{x}, \boldsymbol{\mu}_j)$$

Training RBF Parameters

- Apply *k*-means clustering
- Associate one hidden node per example
- The center $\boldsymbol{\mu}_j$ is the centroid of the cluster or the individual example $\boldsymbol{x}_j$
- Once again, this yields

$$K_j(\mathbf{x}, \boldsymbol{\mu}_j) = \exp\left(-\frac{1}{2\sigma_j^2}\|\mathbf{x} - \boldsymbol{\mu}_j\|^2\right)$$

Training RBF Parameters (cont.)

- Apply gradient descent to learn the centers μ_{hj} and spreads σ_h
- We can find the derivatives with respect to the centers and spreads using the chain rule and a process that looks like back propagation (see later in course).

$$\Delta\mu_{hj} = \eta \sum_t \left[\sum_j (r_i^t - y_i^t) w_{ih} \right] p_h^t \frac{\left(x_j^t - \mu_{hj}\right)}{\sigma_h^2}$$

$$\Delta\sigma_h = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) w_{ih} \right] p_h^t \frac{\|\mathbf{x}^t - \boldsymbol{\mu}_h\|_2^2}{\sigma_h^3}$$

RBF Output

- Loss function:
  - For classification: Cross entropy

$$\mathcal{L} = r^t \log y^t + \left(1 - r^t\right) \log \left(1 - y^t\right)$$

  - For regression: Squared error

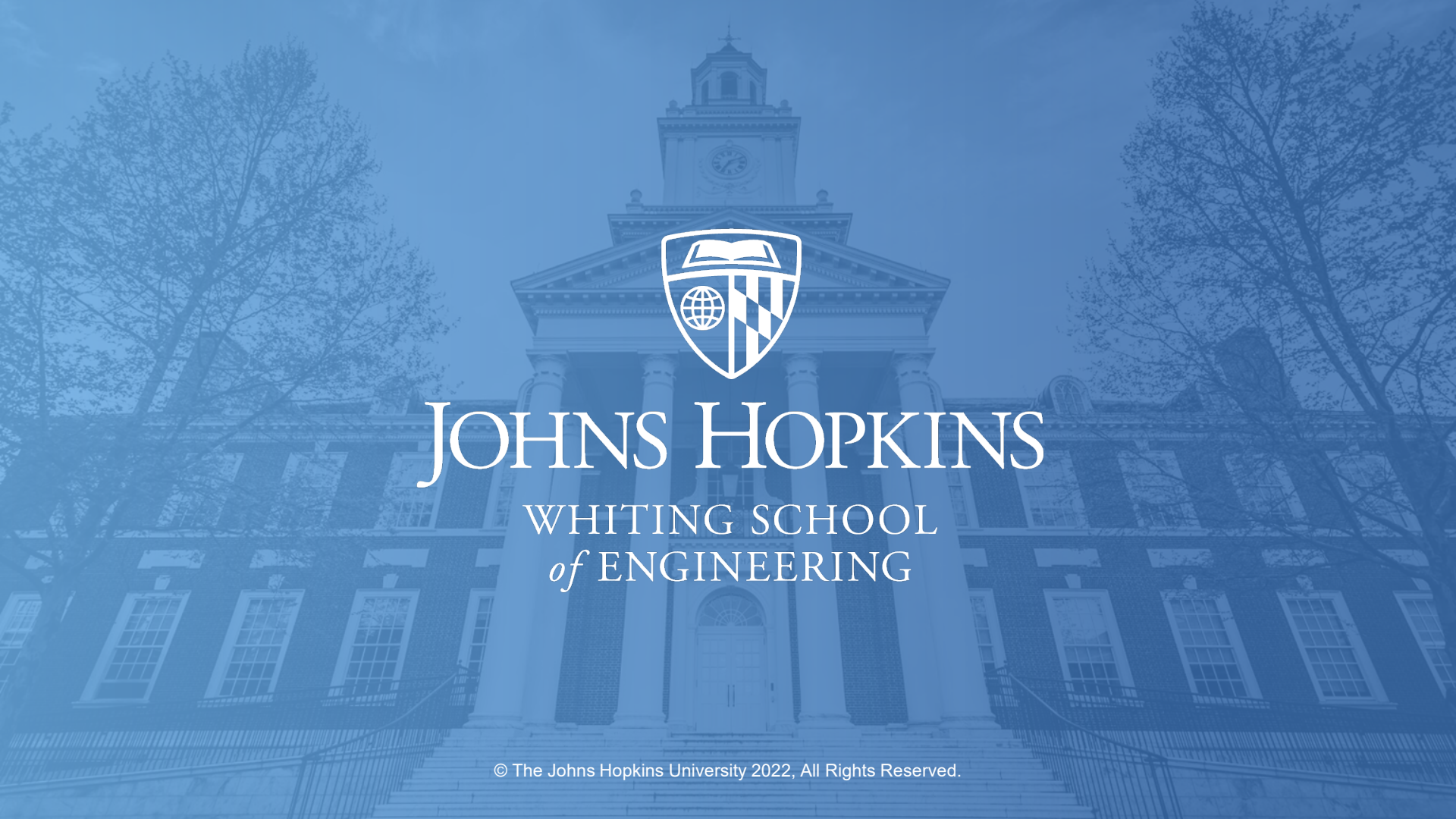
$$\mathcal{L} = \left(r^t - y^t\right)^2$$

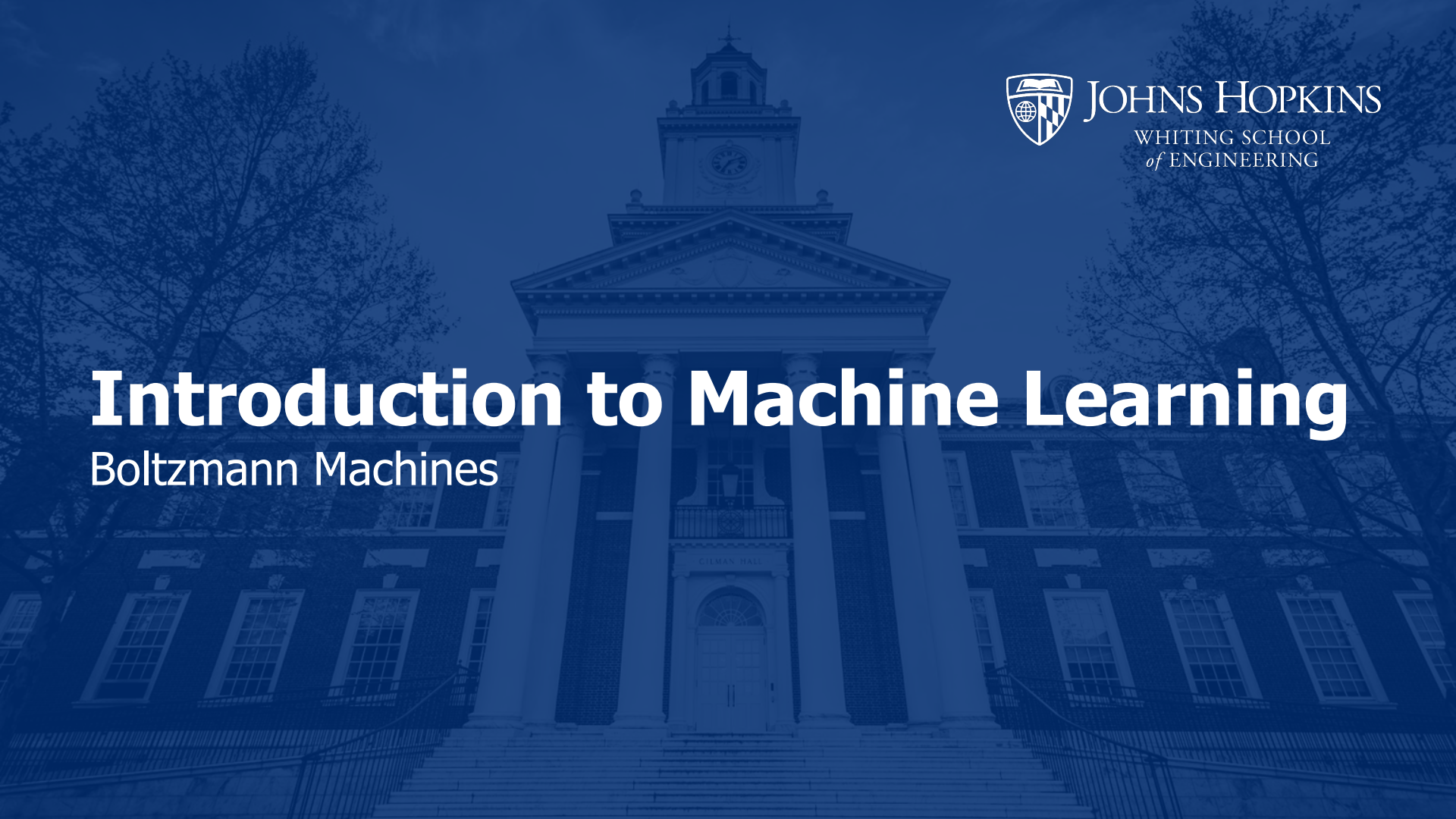
- Update rule is based on $\partial \mathcal{L} / \partial w$
  - For classification:

$$\Delta w = \eta \left(r^t - y^t\right) p_h$$

  - For regression:

$$\Delta \mathrm{w} = \eta \left(r^t - y^t\right) p_h$$

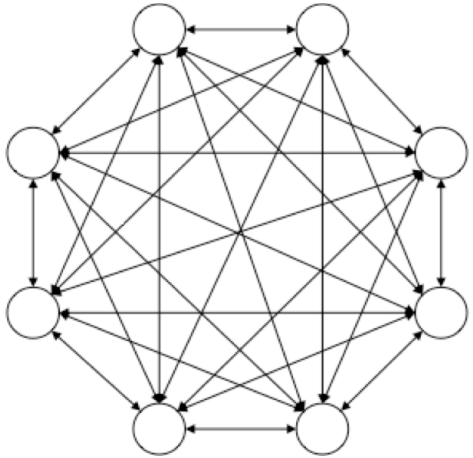




Introduction to Machine Learning

Boltzmann Machines

Hopfield Networks



- An auto-associative network.
  - Denoising
  - Orthogonal patterns

    $n \geq 7p$ neurons for p patterns
- Fully connected (recurrent)
  - All nodes have values in $\{-1, +1\}$

    $w_{ii} = 0$ i.e., no self-loops

    $w_{ij} = w_{ji}$

$$f(\text{net}_j) = \begin{cases} +1 & \text{net}_j > 0 \\ -1 & \text{net}_j < 0 \end{cases}$$

$$net_j = \sum_i w_{ji} x_i$$

$$\begin{pmatrix} 0 & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} & w_{1,6} & w_{1,7} & w_{1,8} \\ w_{2,1} & 0 & w_{2,3} & w_{2,4} & w_{2,5} & w_{2,6} & w_{2,7} & w_{2,8} \\ w_{3,1} & w_{3,2} & 0 & w_{3,4} & w_{3,5} & w_{3,6} & w_{3,7} & w_{3,8} \\ w_{4,1} & w_{4,2} & w_{4,3} & 0 & w_{4,5} & w_{4,6} & w_{4,7} & w_{4,8} \\ w_{5,1} & w_{5,2} & w_{5,3} & w_{5,4} & 0 & w_{5,6} & w_{5,7} & w_{5,8} \\ w_{6,1} & w_{6,2} & w_{6,3} & w_{6,4} & w_{6,5} & 0 & w_{6,7} & w_{6,8} \\ w_{7,1} & w_{7,2} & w_{7,3} & w_{7,4} & w_{7,5} & w_{7,6} & 0 & w_{7,8} \\ w_{8,1} & w_{8,2} & w_{8,3} & w_{8,4} & w_{8,5} & w_{8,6} & w_{8,7} & 0 \end{pmatrix}$$

Energy in Hopfield Networks

- Let h_i be the activation level of node *i*.
- The activation levels are calculated as

$$h_i = \sum_j w_{ij} x_j$$

- Define the energy of a node *i* as

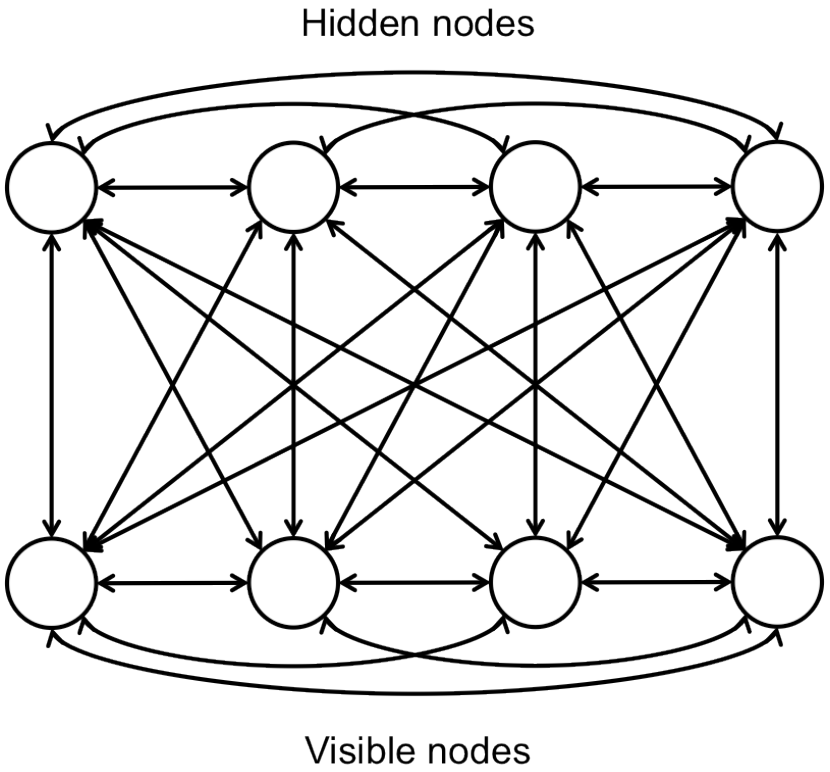
$$E_i = -\frac{1}{2} h_i x_i$$

- The energy of the network is then

$$E(x) = \sum_i E_i = -\sum_i \frac{1}{2} h_i x_i = -\frac{1}{2} \sum_{ij} w_{ij} x_i x_j - \sum_i \theta_i x_i$$

Boltzmann Machines

- A generalization of the Hopfield Network.
- Represents a joint probability distribution.
- Two categories of nodes in the network.
  - Visible nodes
  - Hidden nodes (i.e., latent variables)
- Graph is still fully-connected.



Phases of Operation

- Two phases of operation:
  - Training phase
  - Operational phase
- Conforms to a Boltzmann distribution.
- Assumptions
  - The states of an input vector persist until the network reaches equilibrium
  - The nodes in the network can be updated in any order.
- During training: visible nodes are "clamped" to the values in the input vector.
- A "free running" mode is also included where visible nodes are no longer clamped.

Training Phase

- Let **x** denote the state of the Boltzmann machine.
- Let x_i denote the state of the i^{th} visible node.
- Similar to Hopfield network, the "energy" of the Boltzmann machine is

$$E(\mathbf{x}) = -\frac{1}{2}\sum_i \sum_j w_{ij} x_i x_j$$

- "Gibbs" distribution

$$p_i = \frac{1}{Z}\exp\left(-\frac{E_i}{T}\right)$$

where p_i is the probability of state *i* over **X** occurring, *T* is temperature, and $Z = \sum_j \exp\left(-\frac{E_j}{T}\right)$ is the partition function (normalizer).

The Probability Distribution

$$P(\mathbf{X} = \mathbf{x}) = \frac{1}{Z}\exp\left(-\frac{E(\mathbf{x})}{T}\right) = \frac{1}{Z}\exp\left(\frac{1}{2T}\sum_i\sum_j w_{ij}x_i x_j\right)$$

Consider the probability of a variable in the network, conditioned on the rest of the network.

$$\begin{aligned} P\left(X_j = x_j \mid \{X_i = x_i\}_{i=1,i\neq j}^{n}\right) &= \frac{P\left(\{X_i = x_i\}_{i=1}^{n}\right)}{P\left(\{X_i = x_i\}_{i=1,i\neq j}^{n}\right)} \\ &= \frac{\frac{1}{Z}\exp\left(\frac{1}{2T}\sum_i w_{ij}x_i x_j\right)}{\frac{1}{Z}\exp\left(\frac{1}{2T}\sum_{i,i\neq j} w_{ij}x_i x_j\right)} \\ &= \frac{1}{1+\exp\left(-\frac{x_j}{T}\sum_i w_{ij}x_i x_j\right)} \end{aligned}$$

Simulated Annealing

- Set temperature T "high" at the start (problem dependent)
- Gradually reduce temperature over time.
- At convergence, stable states denote marginal distributions.
- For input vector $\mathbf{x}$, divide up as $\mathbf{x} = [\mathbf{x}_\alpha, \mathbf{x}_\beta]^\top$ where $\mathbf{x}_\alpha$ is the set of "visible" nodes, and $\mathbf{x}_\beta$ is the set of "hidden" nodes.
- Training phases:
  - Positive phase: Network runs in clamped mode.
  - Negative phase: Network runs in free-running mode.

Gradient Descent

- Still want to train by following the gradient of a loss function.
- Use log-likelihood.

$$\mathcal{L}(\mathbf{w}) = \sum_{\mathbf{x}_\alpha \in \mathcal{D}} \left(\log \sum_{\mathbf{x}_\beta} \exp\left(-\frac{E(\mathbf{x})}{T} \right) - \log \sum_{\mathbf{x}} \exp\left(-\frac{E(\mathbf{x})}{T} \right) \right)$$

- The partial derivative with respect to weights

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_{ji}} = \frac{1}{T} \sum_{\mathbf{x}_\alpha \in \mathcal{D}} \left(\sum_{\mathbf{x}_\beta} P\left(\mathbf{x}_\beta = \mathbf{x}_\beta \mid \mathbf{x}_\alpha = \mathbf{x}_\alpha\right) x_j x_i - \sum_{\mathbf{x}} P(\mathbf{X} = \mathbf{x}) x_j x_i \right)$$

- First term: Positive phase
- Second term: Negative phase

Gradient Descent

- Consider these two terms.
- Positive phase: Hebbian learning

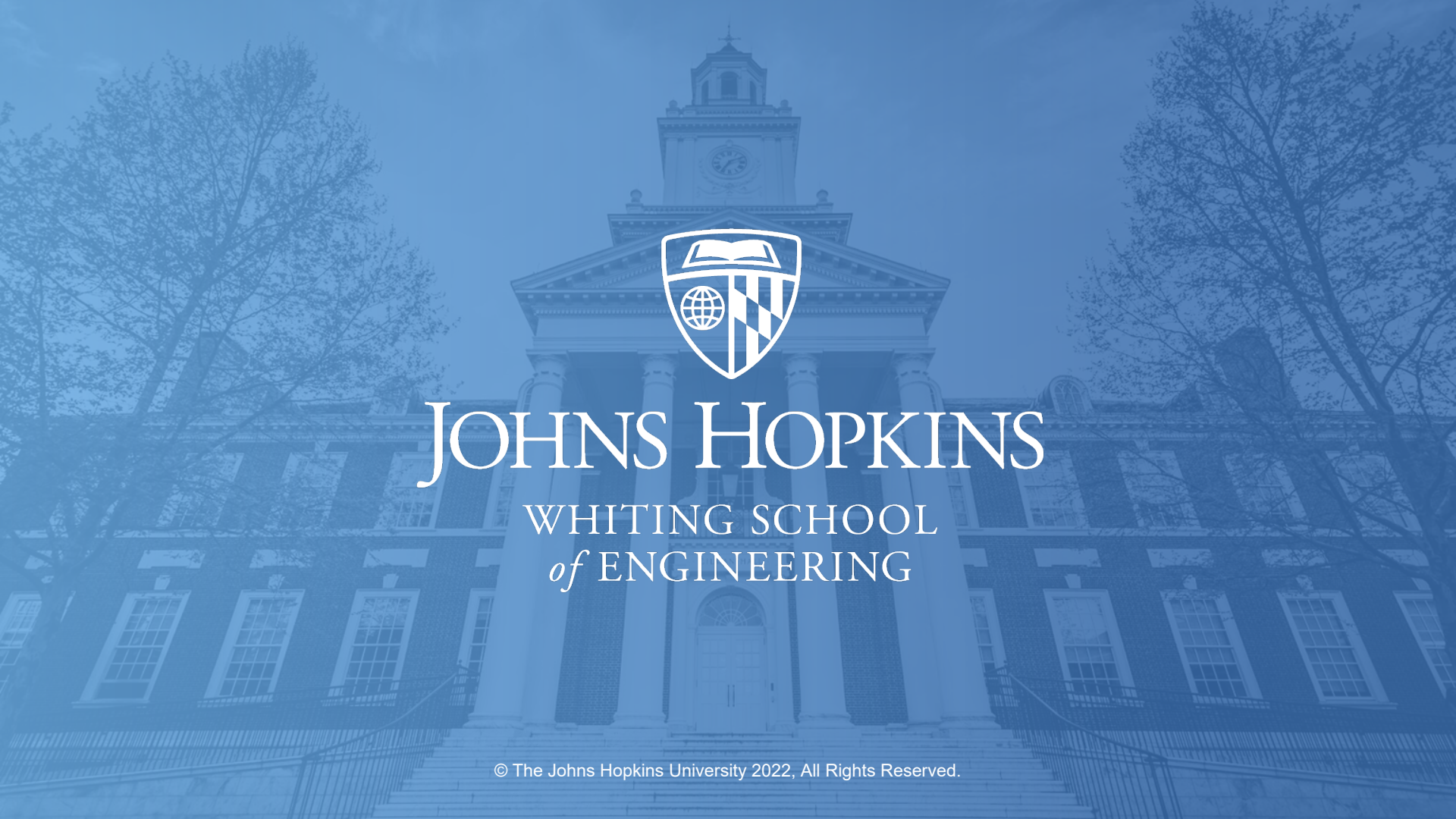
$$\rho_{ji}^{+} = \langle x_j x_i \rangle^{+} = \sum_{\mathbf{x}_\alpha} \sum_{\mathbf{x}_\beta} P\left(X_\beta = x_\beta \mid X_\alpha = x_\alpha\right) x_j x_i$$

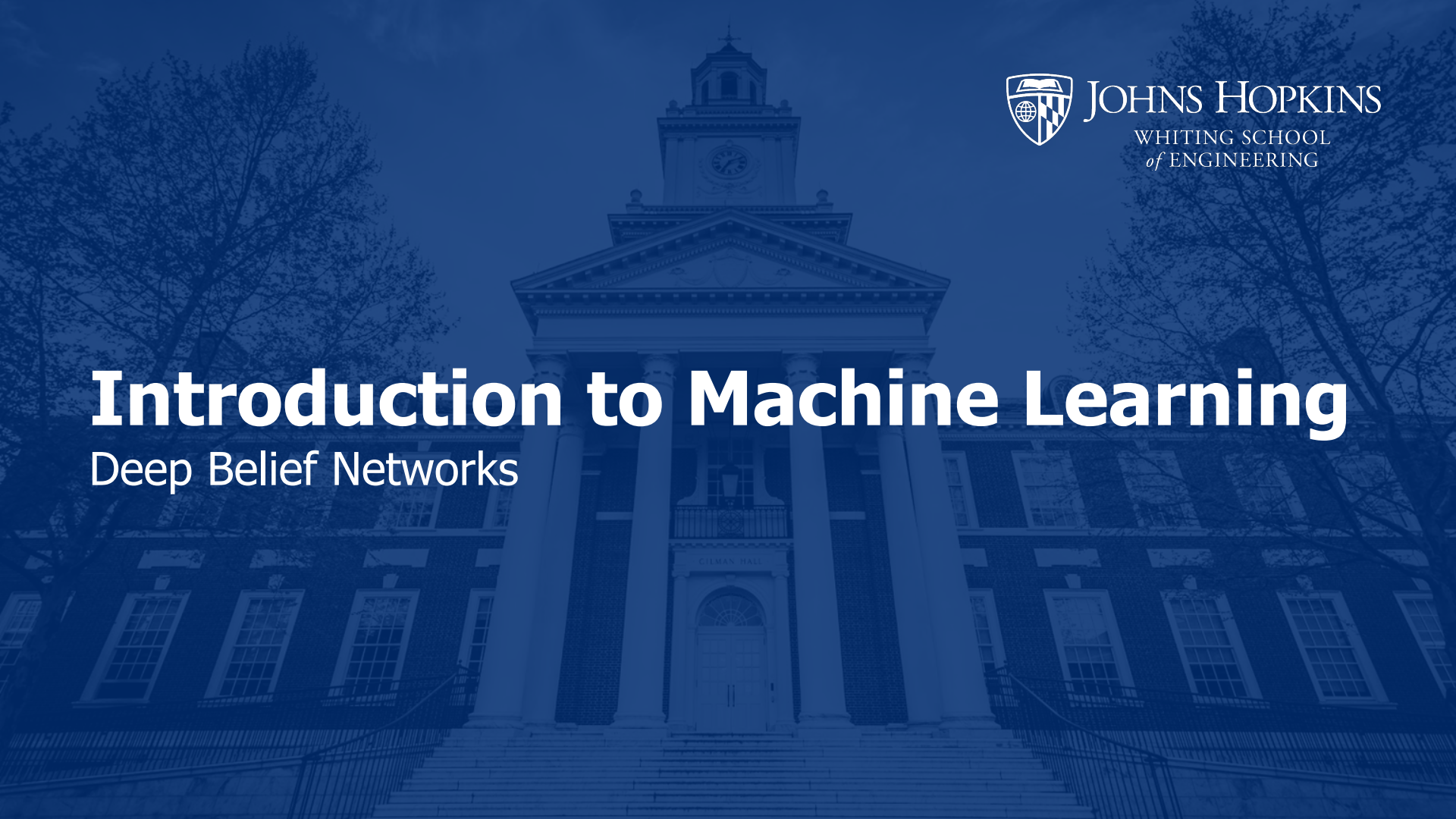
- Negative phase: Forgetting

$$\rho_{ji}^{-} = \langle x_j x_i \rangle^{-} = \sum_{\mathbf{x}_\alpha} P(\mathbf{X} = \mathbf{x}) x_j x_i$$

- New gradient calculation

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_{ji}} = \frac{1}{T}\left(\rho_{ji}^{+} - \rho_{ji}^{-}\right) \Rightarrow \Delta w_{ji} = \eta\left(\rho_{ji}^{+} - \rho_{ji}^{-}\right), \eta = \epsilon / T$$

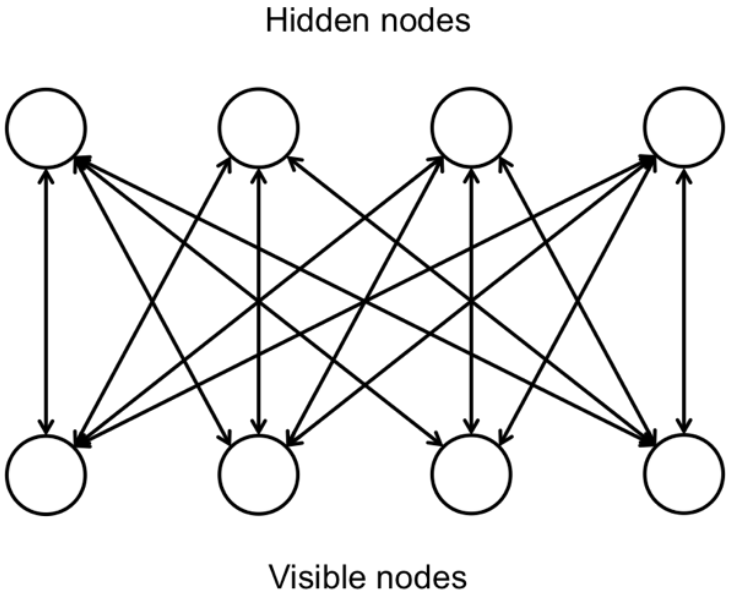




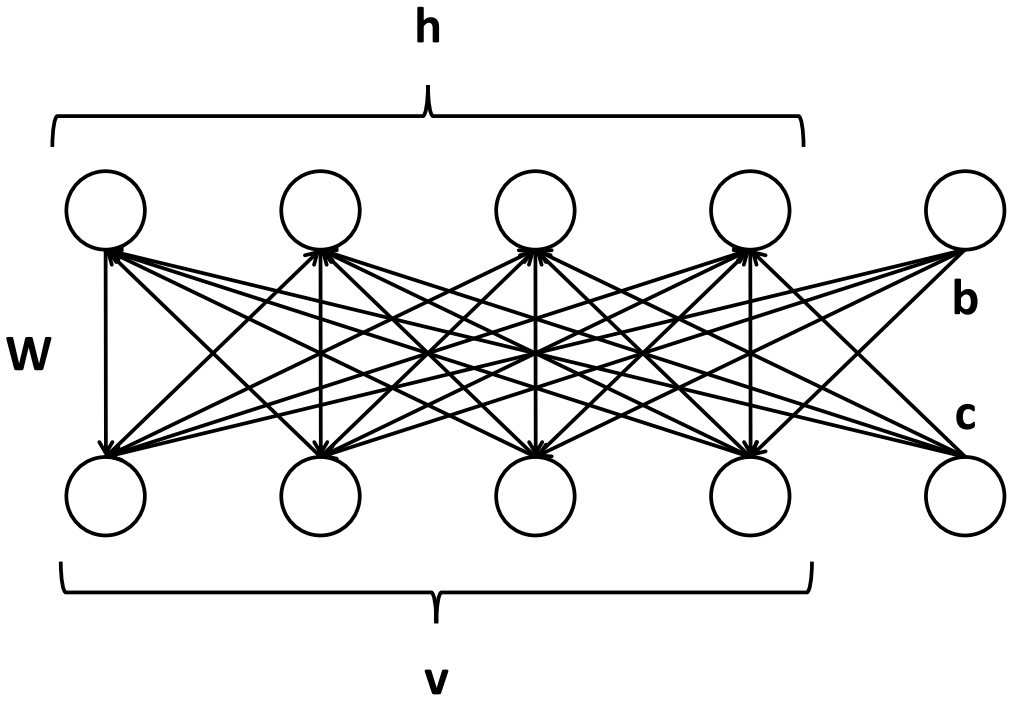
Introduction to Machine Learning

Deep Belief Networks

Restricted Boltzmann Machine



Restricted Boltzmann Machine (w/ Bias)



Energy Function

- Define energy as

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h} - \mathbf{h}^\top \mathbf{W} \mathbf{v}$$

- Let $\sigma(\mathbf{x})$ denote the logistic (sigmoid) function.
- Apply iterative sampling.

$$\mathbf{h}^{(t+1)} \sim P\left(\mathbf{h} \mid \mathbf{v}^{(t)}\right) = \sigma\left(\mathbf{W}\mathbf{v}^{(t)} + \mathbf{c}\right)$$

$$\mathbf{v}^{(t+1)} \sim P\left(\mathbf{v} \mid \mathbf{h}^{(t+1)}\right) = \sigma\left(\mathbf{h}^{(t+1)}\mathbf{W} + \mathbf{b}\right)$$

- As $t \to \infty$ the accuracy of $P(\mathbf{v}, \mathbf{h})$ increases.

Training an RBM

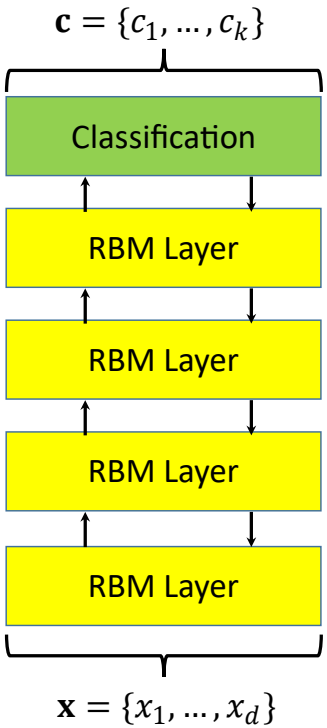
Contrastive Divergence (CD) – Invented by Geoff Hinton

- First, set visible nodes to a training example.
- Next, iterate a "truncated" Markov chain.

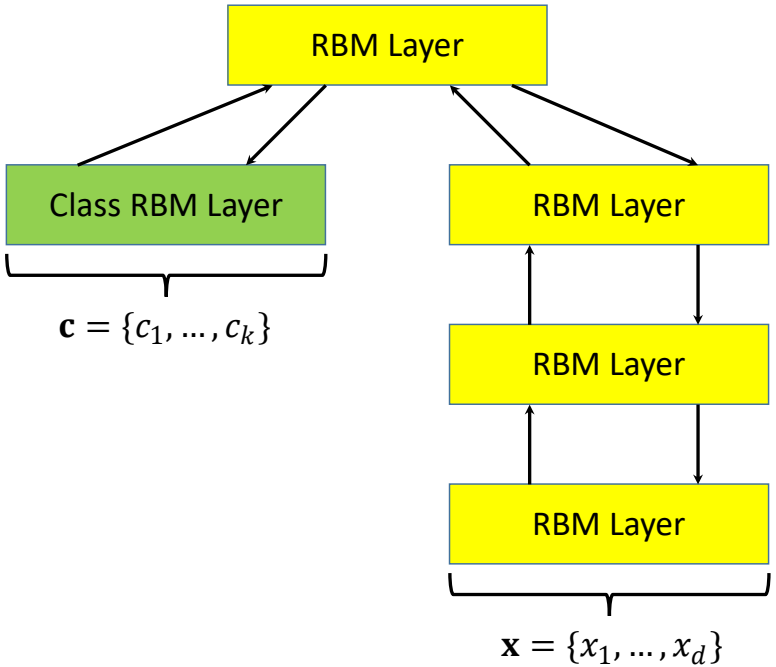
Standard, recommended method is CD-1 – Gibbs Sampling

1. Assign **x** to visible nodes **v**.
2. Generate one sample for hidden nodes: $\mathbf{h} \sim P(\mathbf{h} \mid \mathbf{v})$
3. Reconstruct **v** from **h** through sampling: $\mathbf{v} \sim \mathrm{P}(\mathbf{v} \mid \mathbf{h})$
4. Update the weights: $\Delta w_{ij} \leftarrow \eta \left(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{recon}} \right)$ where $\langle \cdot \rangle_P$ denotes expected value over the distribution *P*.
5. Repeat over all **x** until convergence.

Deep Belief Network



ClassDBN

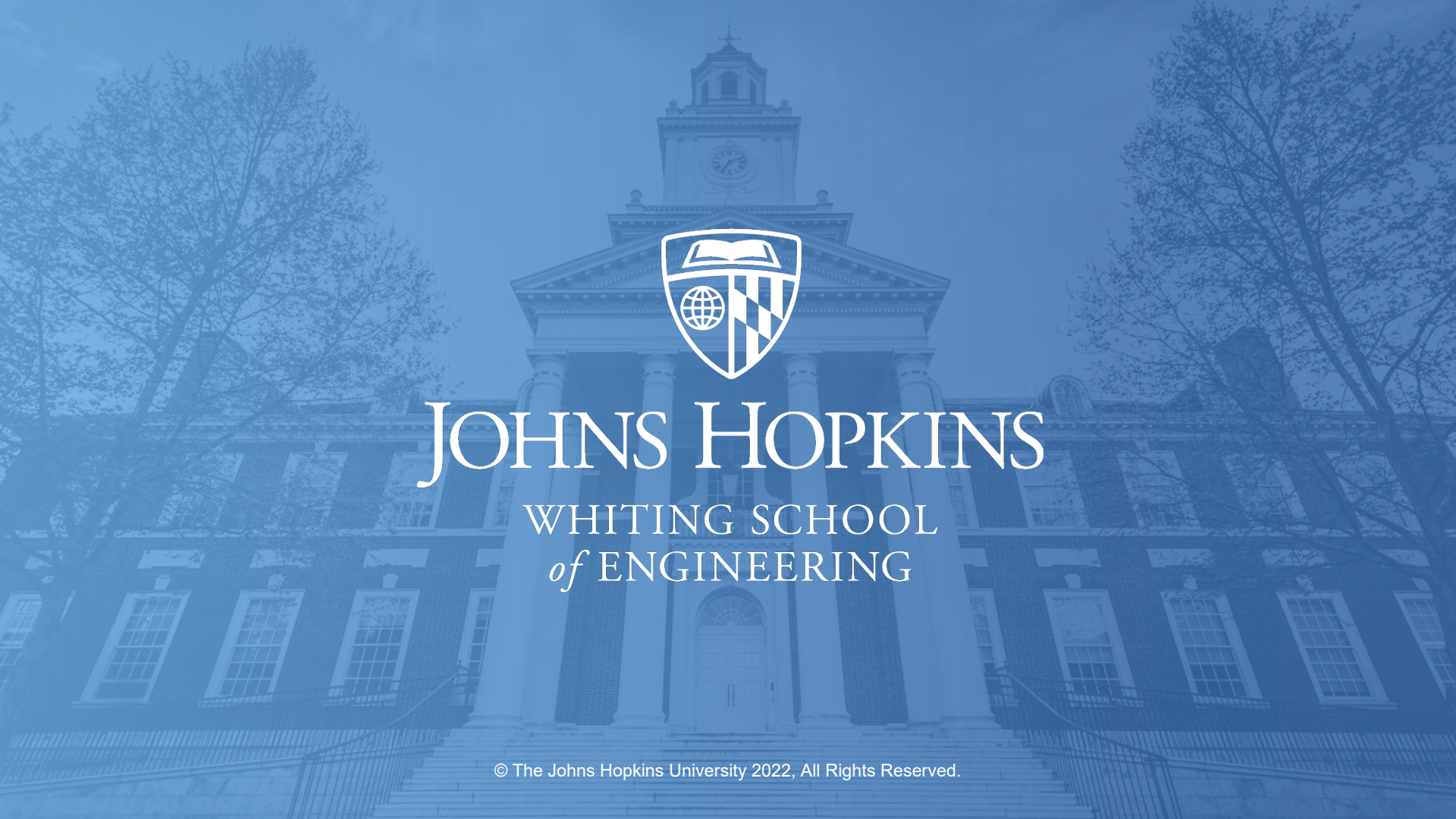

RBM Layer
Class RBM Layer
RBM Layer
RBM Layer
RBM Layer
$\mathbf{c} = \{c_1, \ldots, c_k\}$
$\mathbf{x} = \{x_1, \ldots, x_d\}$

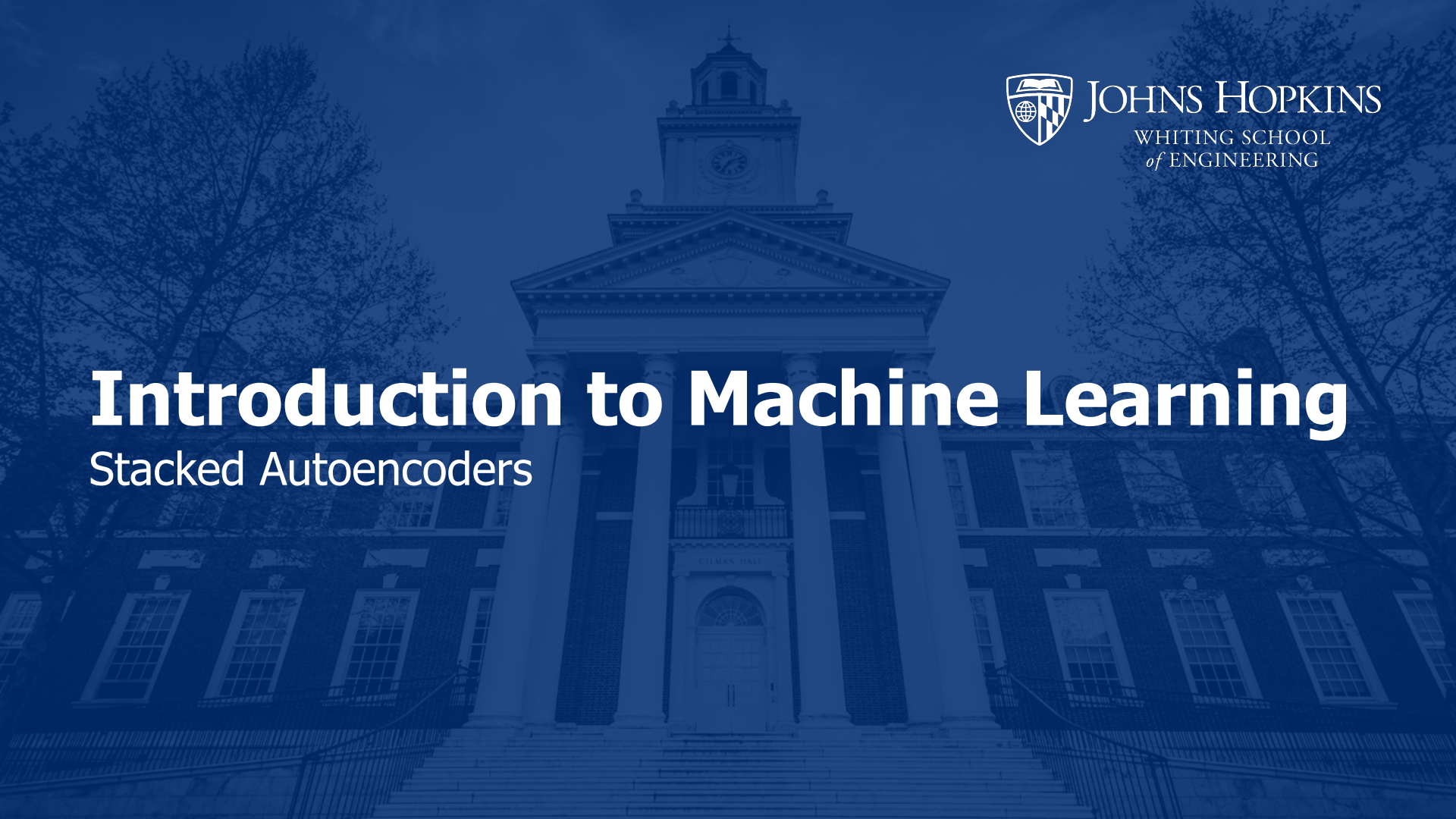
Training a DBN

- Layerwise pre-training.

1. Train the first RBM, and generate samples at the hidden layer.
2. Use the generated samples to generate another RBM.
3. Repeat this process until the desired depth is reached.
4. Add a classification layer.
5. Apply gradient descent/backpropagation to the output layer or to fine-tune entire network.

- Note that the ClassRBM architecture would use CD for all of the layers.



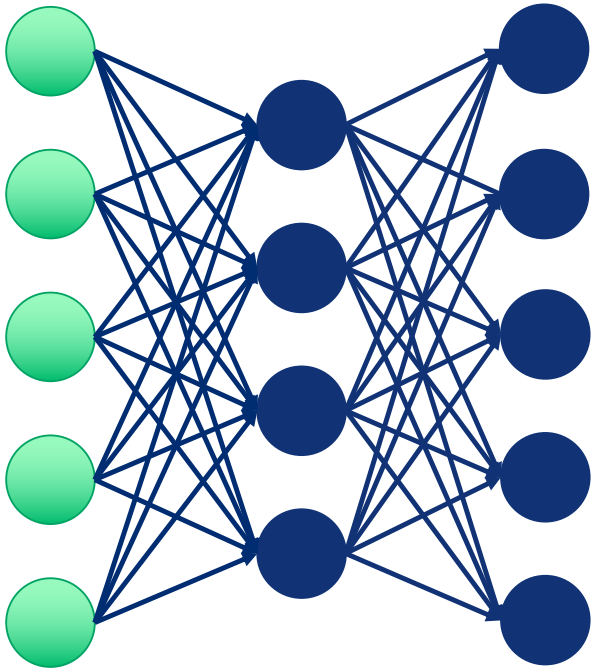


Introduction to Machine Learning

Stacked Autoencoders

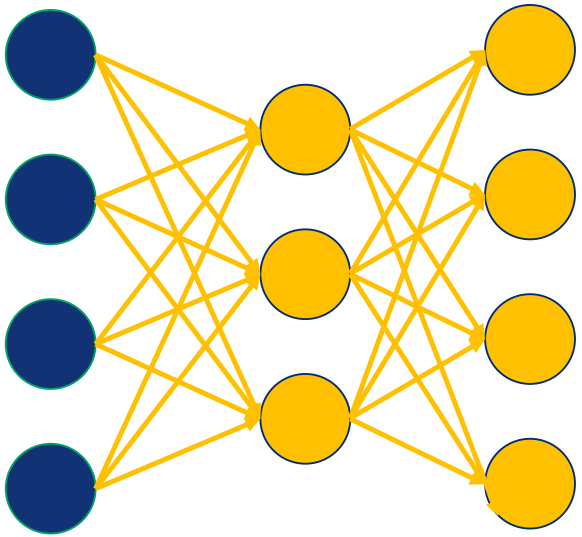
Autoencoder

- Output matches input
- Applications
  - Dimensionality reduction
  - Feature extraction
  - Denoising
- Often used for image/audio processing



Stacking

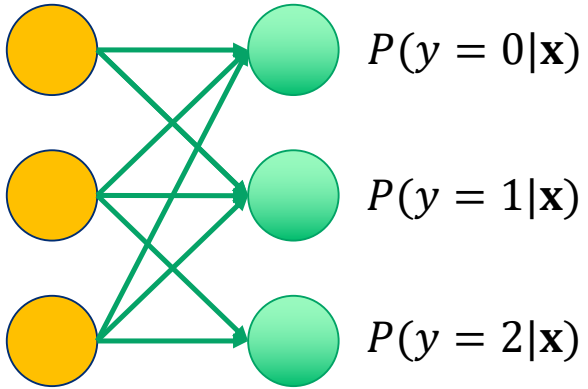
- Autoencoders are auto-associative.
- Can stack a sequence of autoencoders.
- Data generated for each autoencoder is from the encoding layers.



Prediction

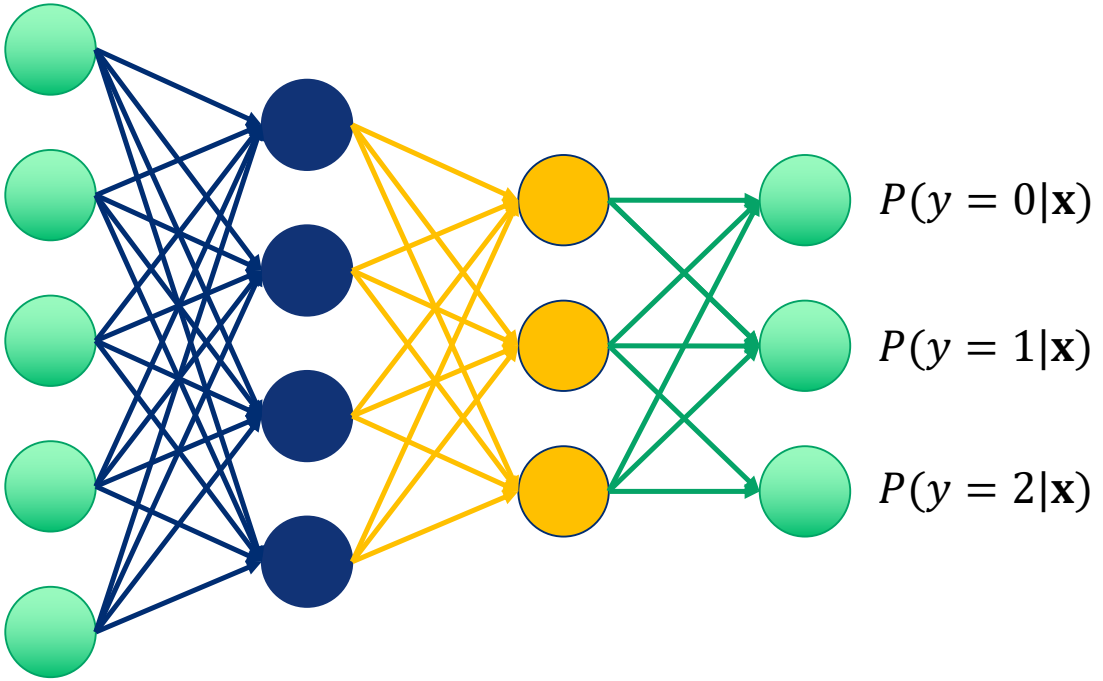
- Want to train a deep multi-layer network.
- Need to contend with the vanishing gradient problem.
- Train in a layerwise fashion.
- Classification:

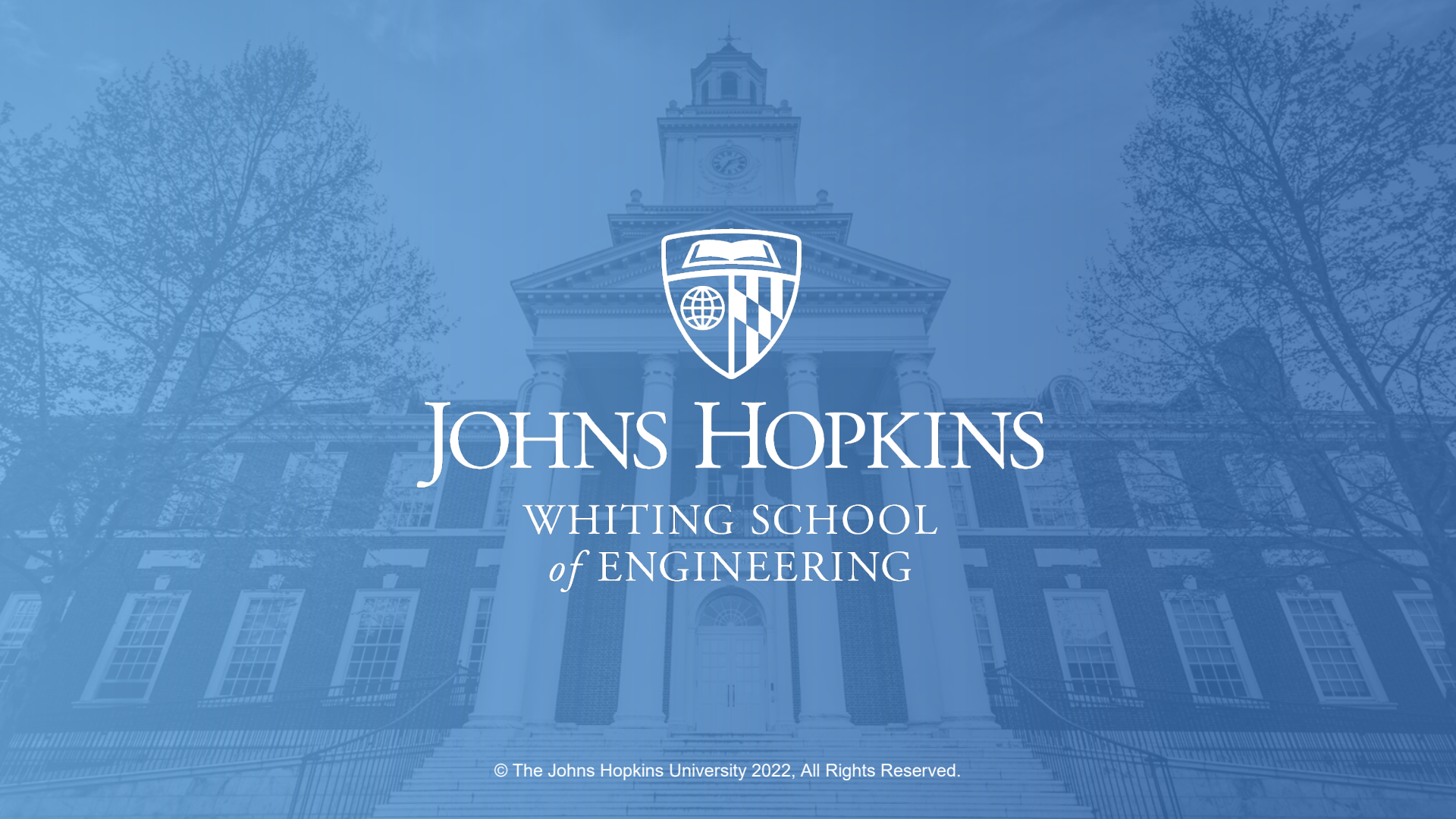
$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{c=0}^{K} e^{\mathbf{x}^\top \mathbf{w}_c}}$$

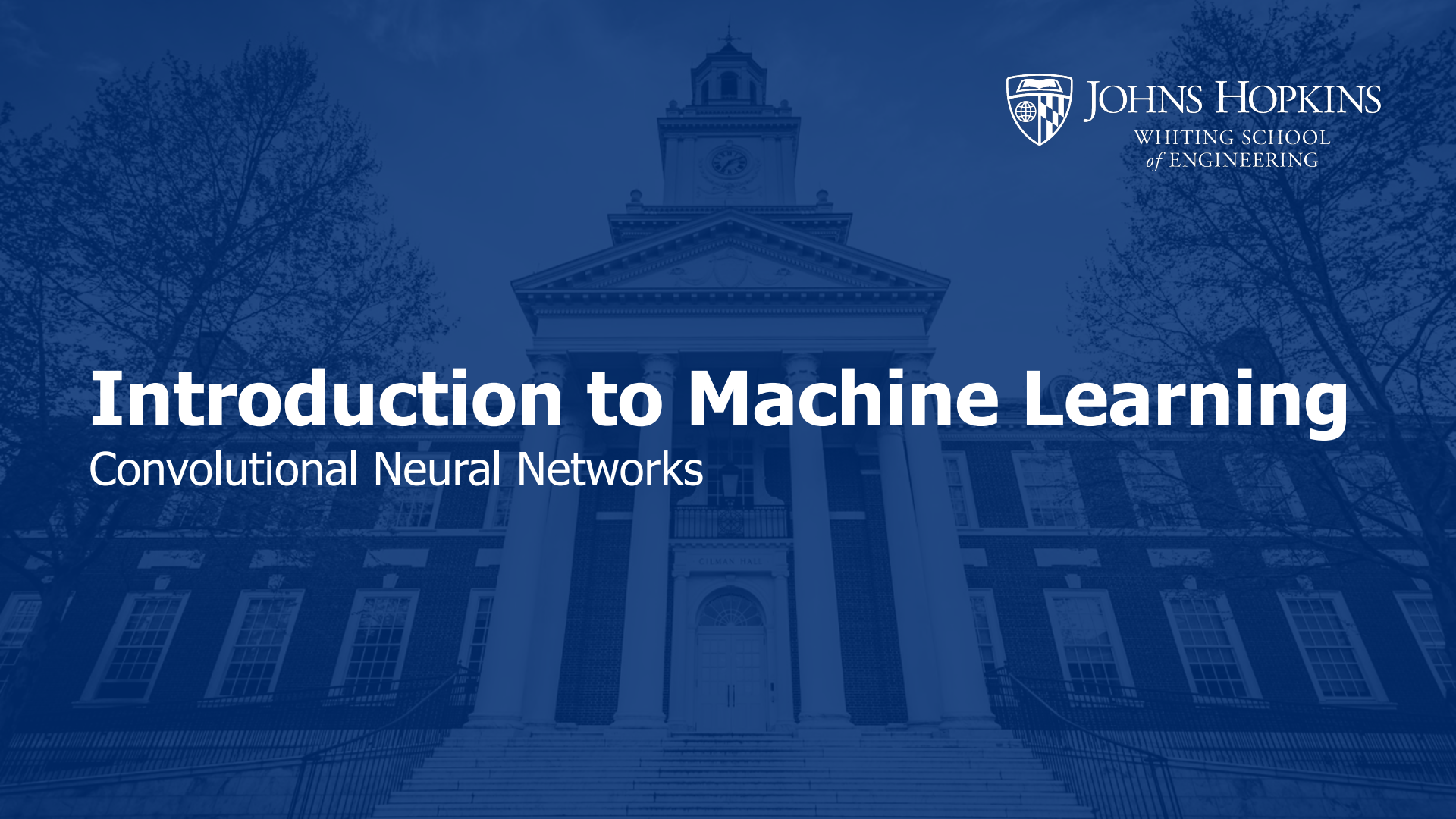


Training Process

- Pre-train layers.
- Stack the layers as train.
- Train the output layer.
  - Apply backprop to output.
  - Apply backprop to network.
- Fine tune the result.







Introduction to Machine Learning

Convolutional Neural Networks

Density Estimation

- Recall that we can estimate the density nonparametrically as follows:

$$\hat{p}(\mathbf{x}) \approx \frac{1}{nV} \sum_{t=1}^{n} K\left(\frac{\mathbf{x} - \mathbf{x}^t}{V}\right)$$

- Notice that $\hat{p}(\mathbf{x})$ is an approximation, from which we can estimate an expected value:

$$\bar{p}(\mathbf{x}) = E[\hat{p}(\mathbf{x})] = \int \underbrace{\frac{1}{V} K\left(\frac{\mathbf{x} - \mathbf{v}}{V}\right)}_{\delta(\mathbf{x}-\mathbf{v})} p(\mathbf{v}) d\mathbf{v}$$

Convolution

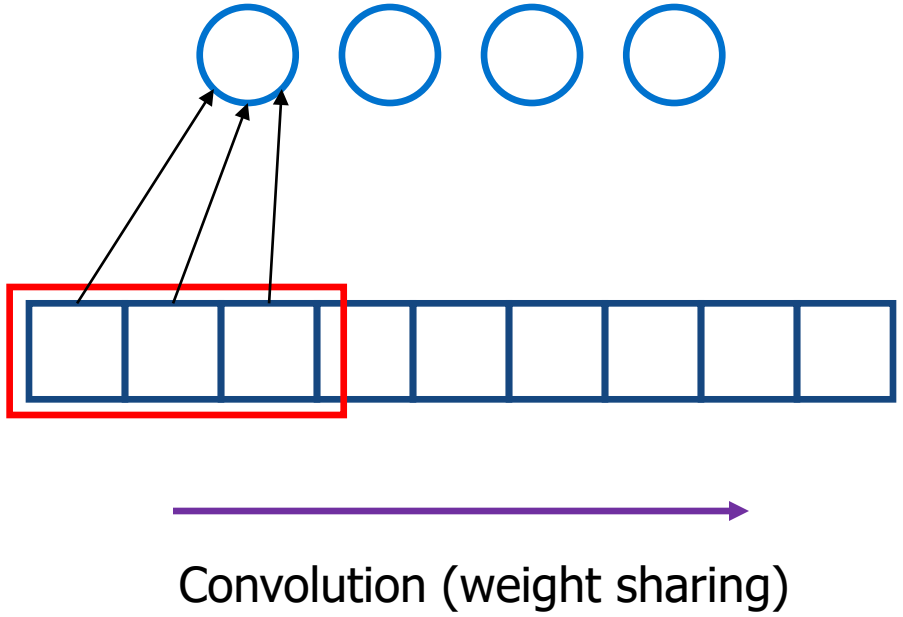
- Substituting, we get

$$\bar{p}(\mathbf{x}) = \int \delta(\mathbf{x} - \mathbf{v}) p(\mathbf{v}) d\mathbf{v}$$

- This is called "convolution."
- This operation is the basis for the convolutional neural network.
- The book provides details in 2D, let's consider a 1D case.

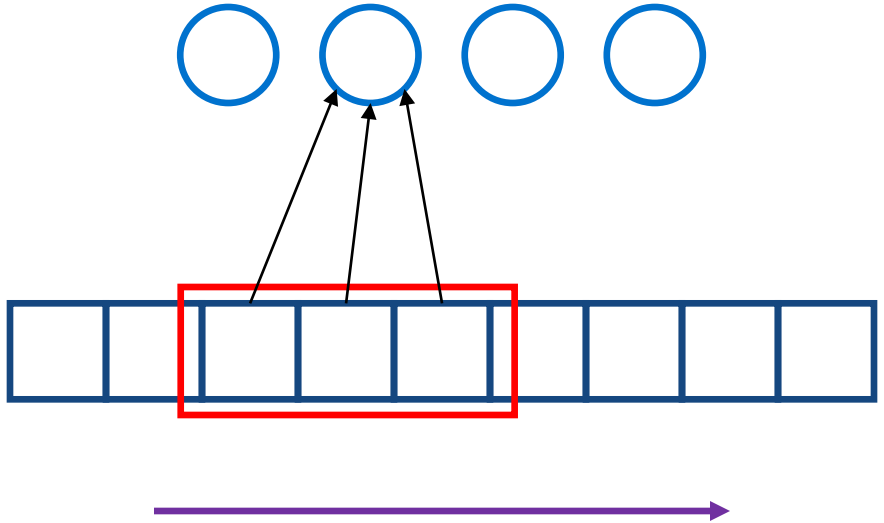
Convolutional Layers

A convolutional layer (stride 2) – weight sharing:



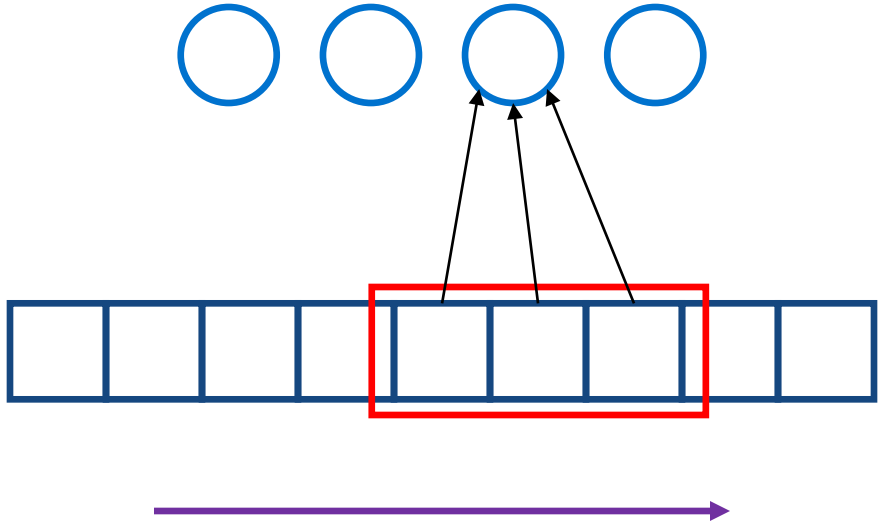
Convolutional Layers

A convolutional layer (stride 2) – weight sharing:



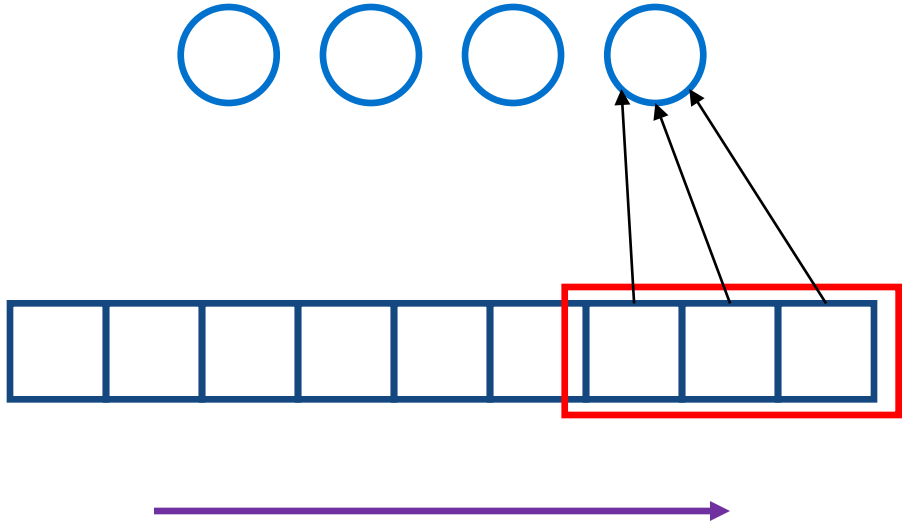
Convolutional Layers

A convolutional layer (stride 2) – weight sharing:



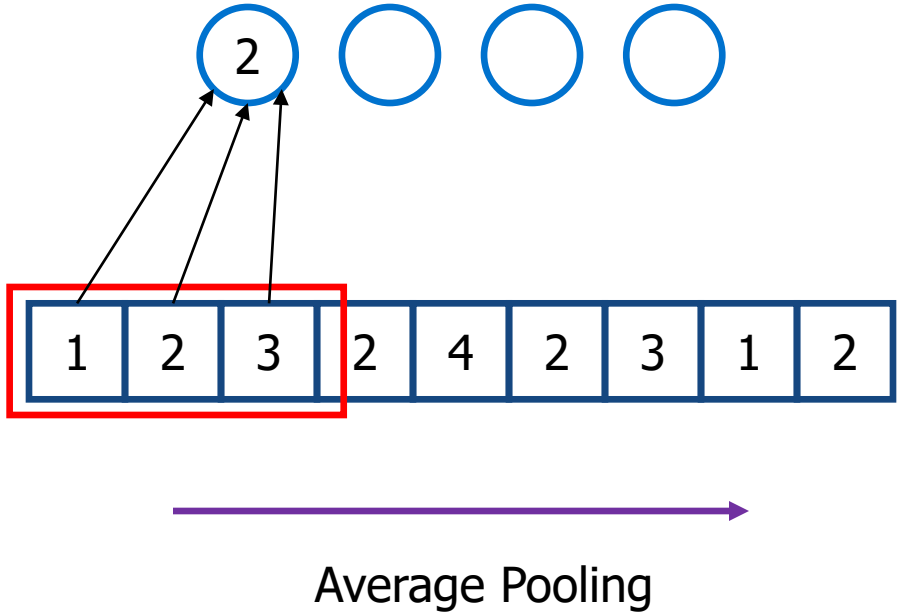
Convolutional Layers

A convolutional layer (stride 2) – weight sharing:



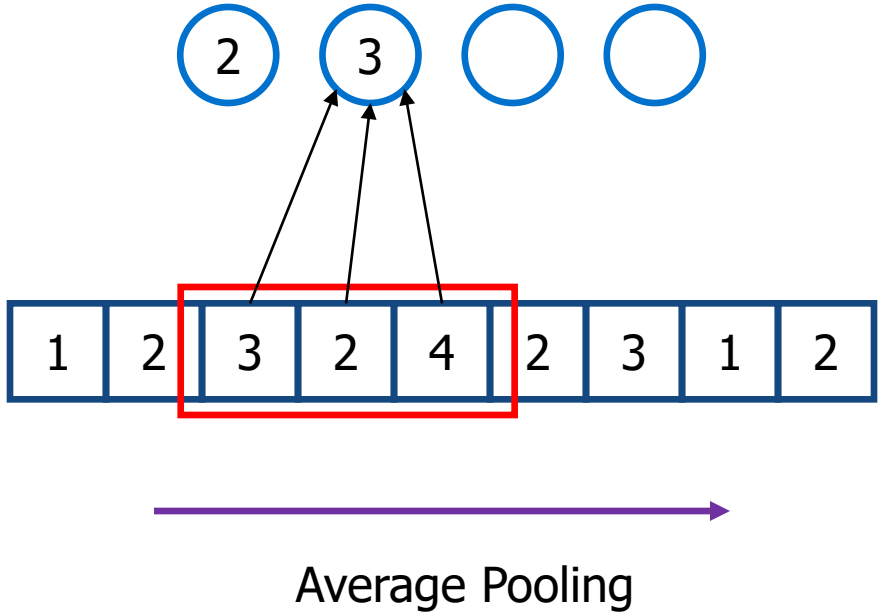
Pooling Layers – Maximum or Average

An average pooling layer (stride 2):



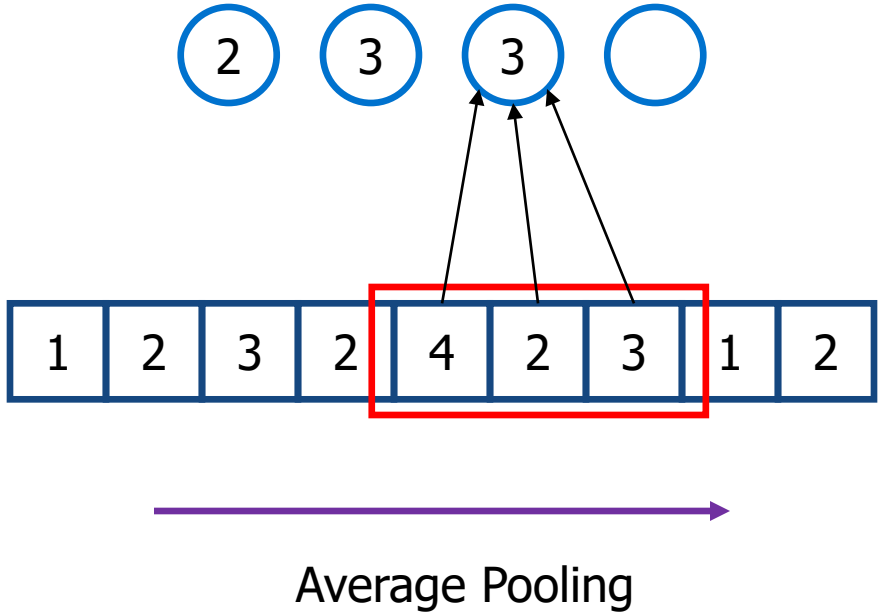
Pooling Layers – Maximum or Average

An average pooling layer (stride 2):



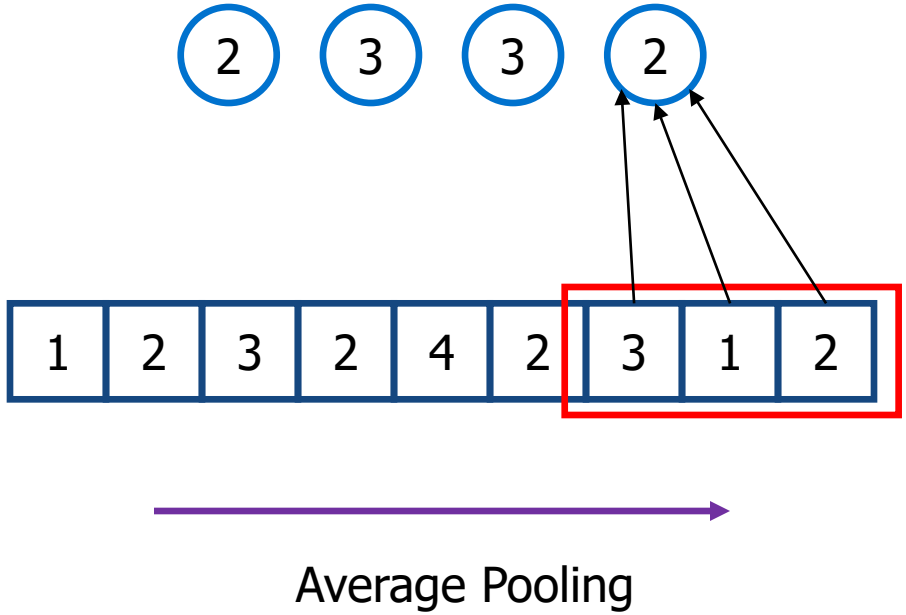
Pooling Layers – Maximum or Average

An average pooling layer (stride 2):



Pooling Layers – Maximum or Average

An average pooling layer (stride 2):



Batch Normalization

- Recall that it is recommended to normalize all inputs for a neural network.
- It is customary to apply z-score normalization, resulting in a consistent spread and value range of the data.
- This can be applied to the hidden nodes as well.
- For each mini-batch used in stochastic gradient descent, when propagating to hidden unit *j*, but before applying the activation at the hidden node, compute

$$\tilde{a}_j = \frac{a_j - \mu_j}{\sigma_j}$$

- Apply parameters (to be learned) to compute $\hat{a}_j = \gamma_j \tilde{a}_j + \beta_j$ and then apply the activation function.
- For prediction, computing μ_j and β_j from the entire training set.

2D Convolutional Network

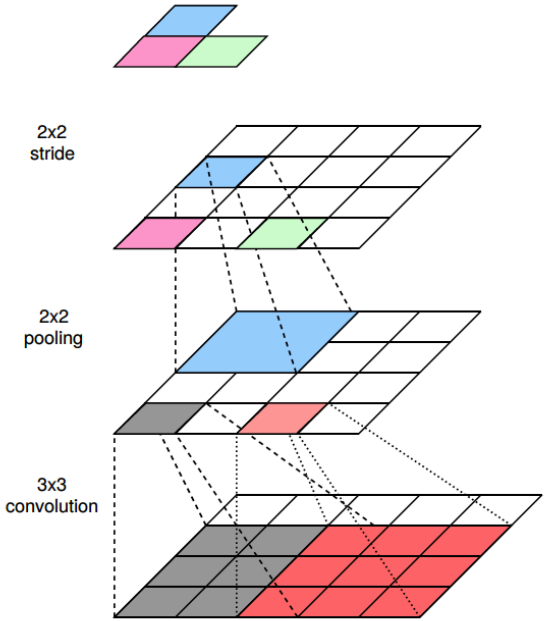


Figure 12.5 The 3×3 convolutional layer applies the convolution at all points. The pooling layer pools the values in a 2×2 patch, and a stride of 2 lowers the resolution by half.

LeNet-5 (Handwritten Digits)

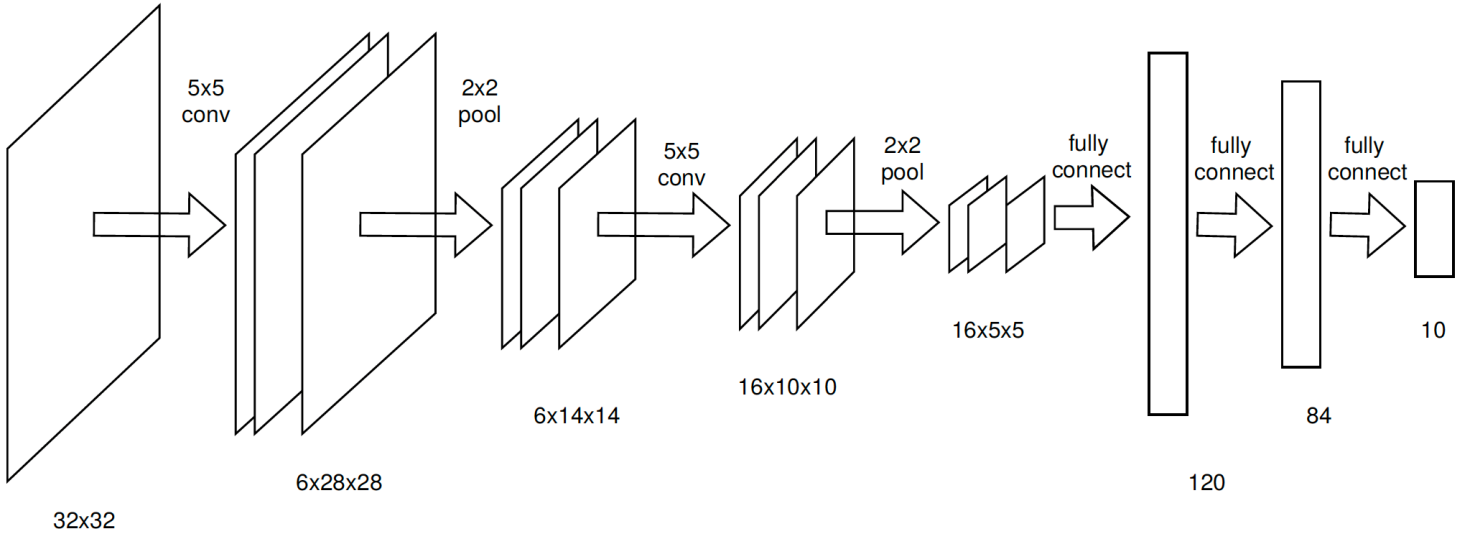
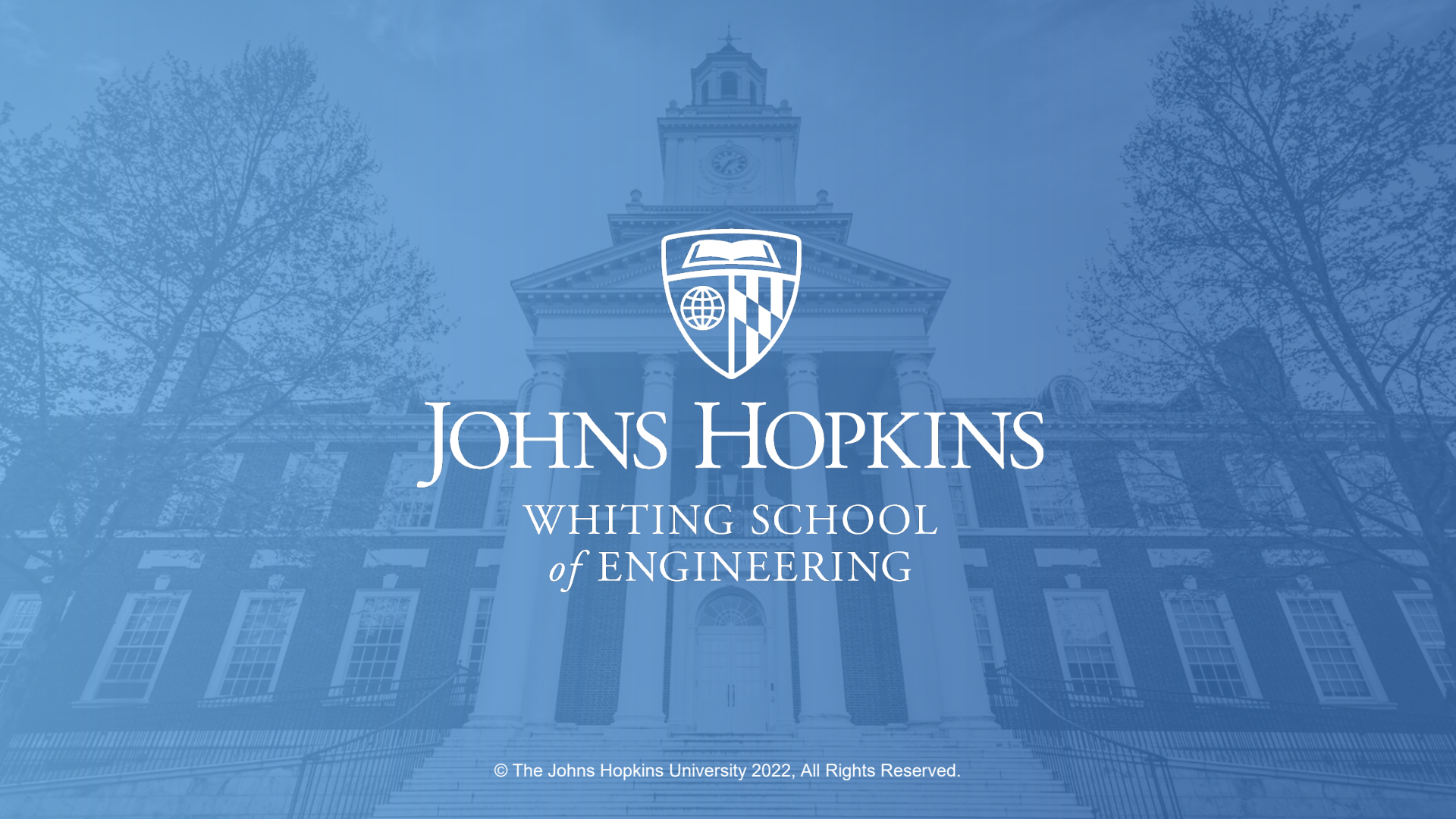
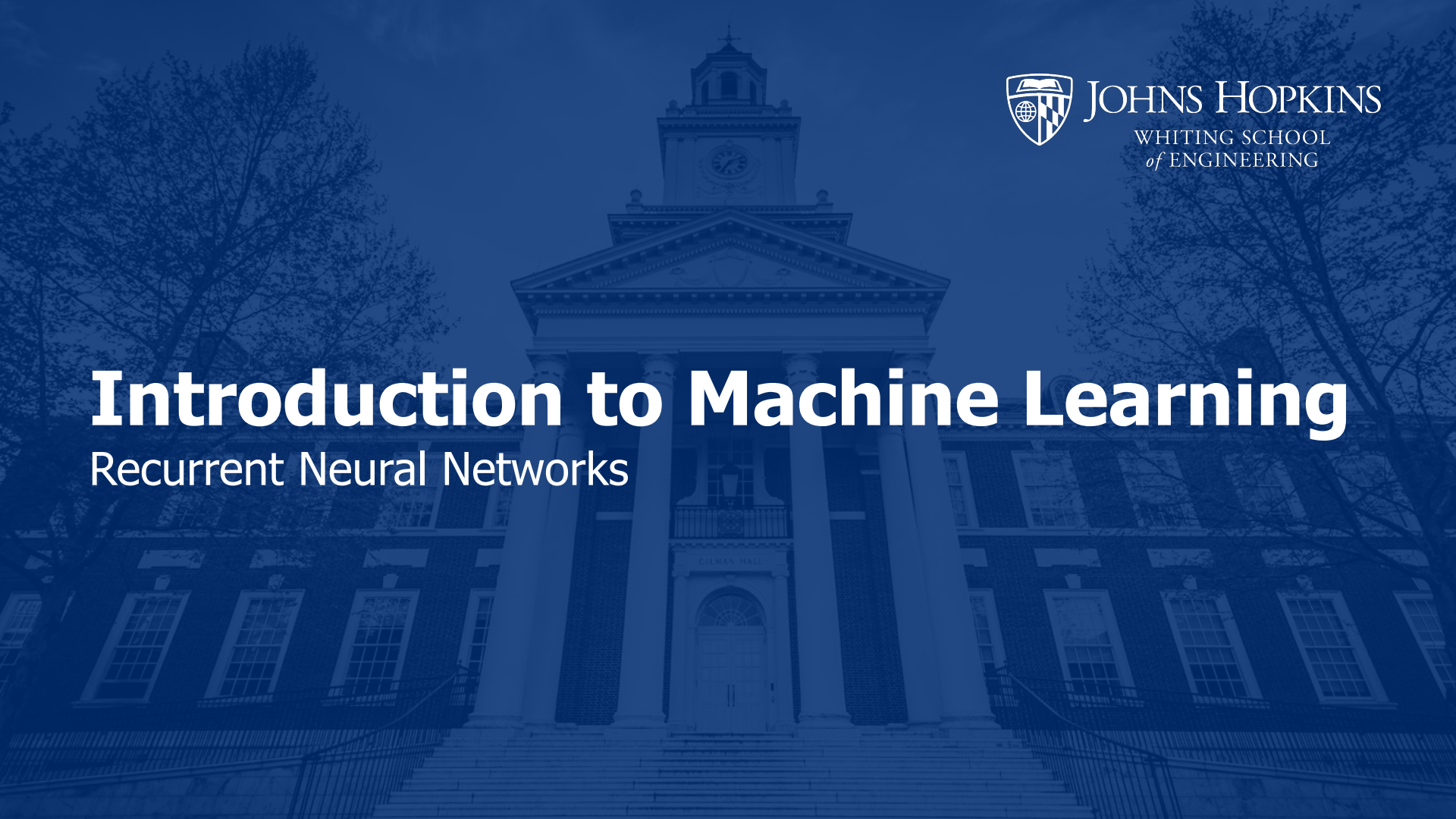


Figure 12.6 LeNet-5 network for recognition of handwritten digits (Le Cun et al 1998).

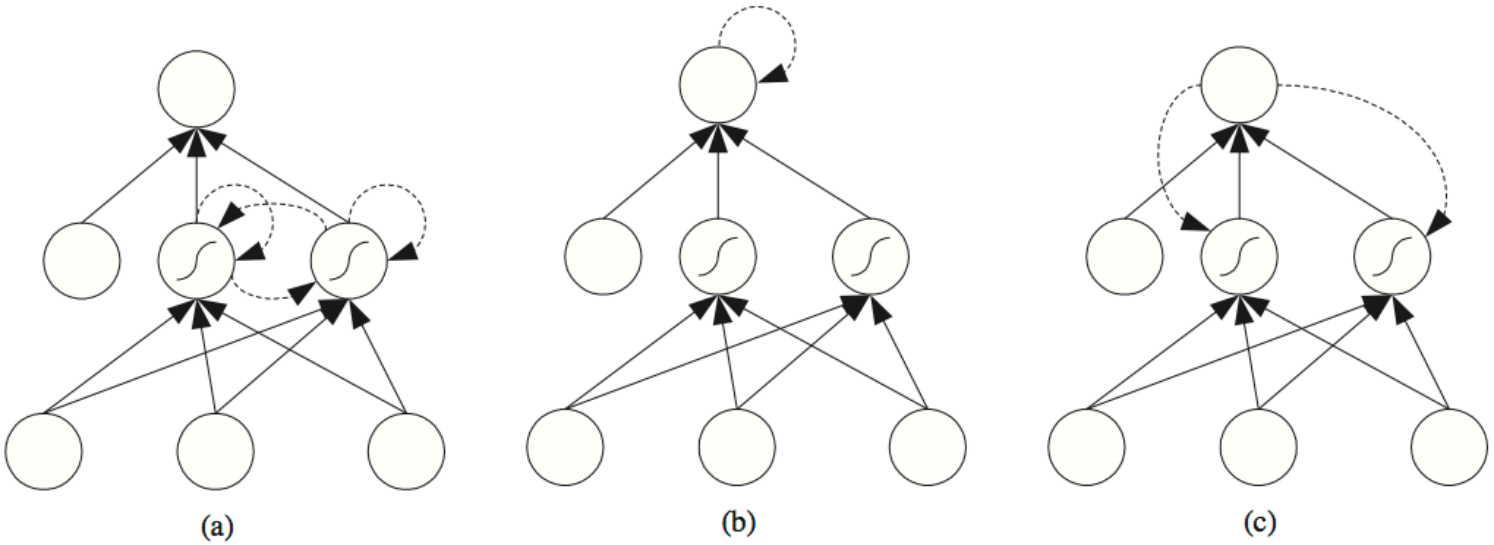




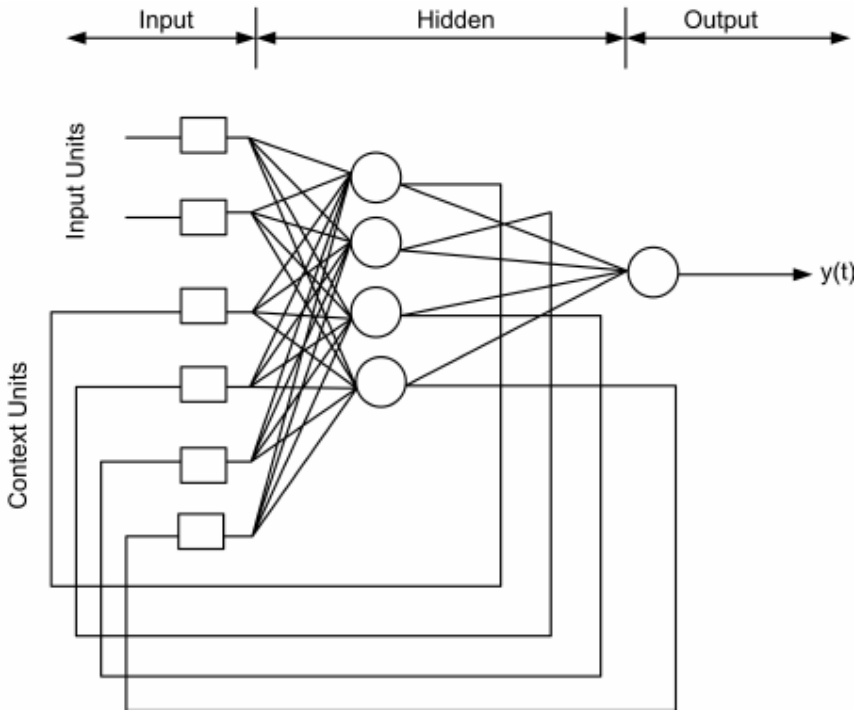
Introduction to Machine Learning

Recurrent Neural Networks

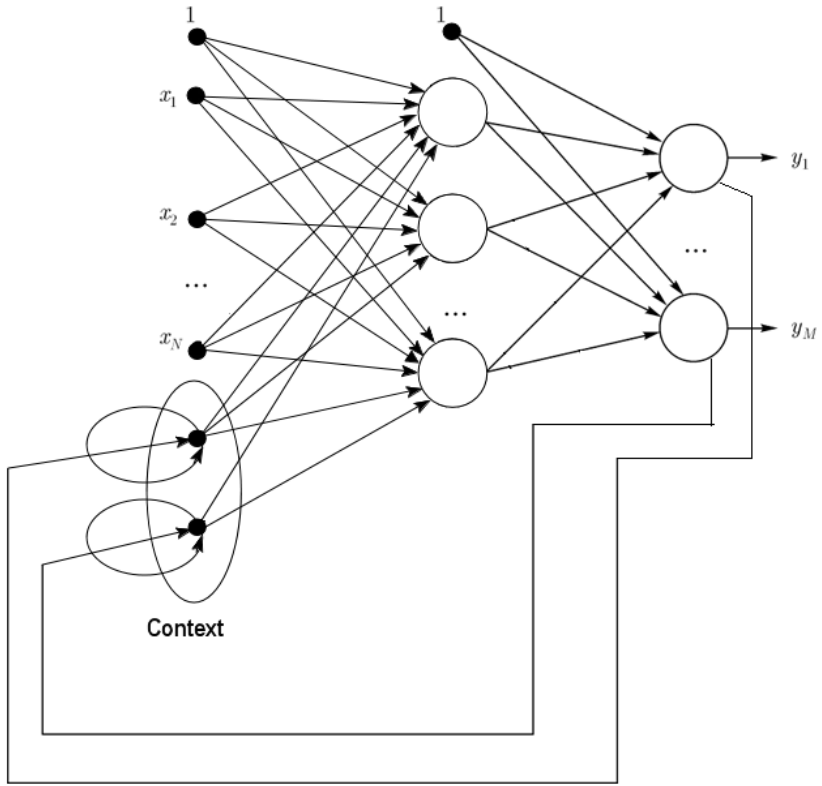
What is a Recurrent Neural Net?



Elman Network



Jordan Network



Backpropagation Through Time

- Epoch-wise BPTT.
- Let t_0 be initial epoch and t_T be final epoch.

$$\mathrm{Err}^{\mathrm{total}}\ (t_0, t_T) = \frac{1}{2} \sum_{t=t_0}^{t_T} \sum_{k \in \text{ outputs}} \mathrm{Err}_k(t)^2$$

$$\delta_k(\tau) = \frac{-\partial Er^{\mathrm{total}}\ (t_0, t_T)}{\partial_{s_k}(\tau)} = \begin{cases} f_k'\left[s_k(\tau)\right] \mathrm{Err}_k(\tau) & \text{if } \tau = T \\ f_k'\left[s_k(\tau)\right] \left[\mathrm{Err}_k(\tau) + \sum_{l \in \text{ outputs}} w_{lk}\delta_l(\tau+1)\right] & \text{otherwise} \end{cases}$$

$s_k(t)$ is the net input to the k^{th} unit at time t.

- Injection of error.

Real Time Recurrent Learning

$\mathbf{y}(t)$ is an n-tuple of outputs of all units at time *t*.

$\mathbf{x}(t)$ is an m-tuple of external input signals at time *t*.

$$z(t) = \langle x_1(t), \ldots, x_m(t), y_1(t), \ldots, y_n(t) \rangle$$

- Update rule:

$$\Delta w_{ij}(t) = \eta \sum_{k \in \text{ outputs}} \text{Err}_k(t) p_{ij}^k(t)$$

$$\text{Err}_k(t) = \begin{cases} d_k(t) - y_k(T) & k \in \text{ outputs} \\ 0 & \text{otherwise} \end{cases}$$

$$p_{ij}^k(t+1) = f_k'\left(s_k(t)\right) \left[\sum_{l \in \text{ outputs}} w_{kl} p_{ij}^l + \delta_{ik} z_j(t) \right]$$