ILLINOIS INSTITUTE OF TECHNOLOGY

Feed-forward Neural Networks

CS-585

Natural Language Processing

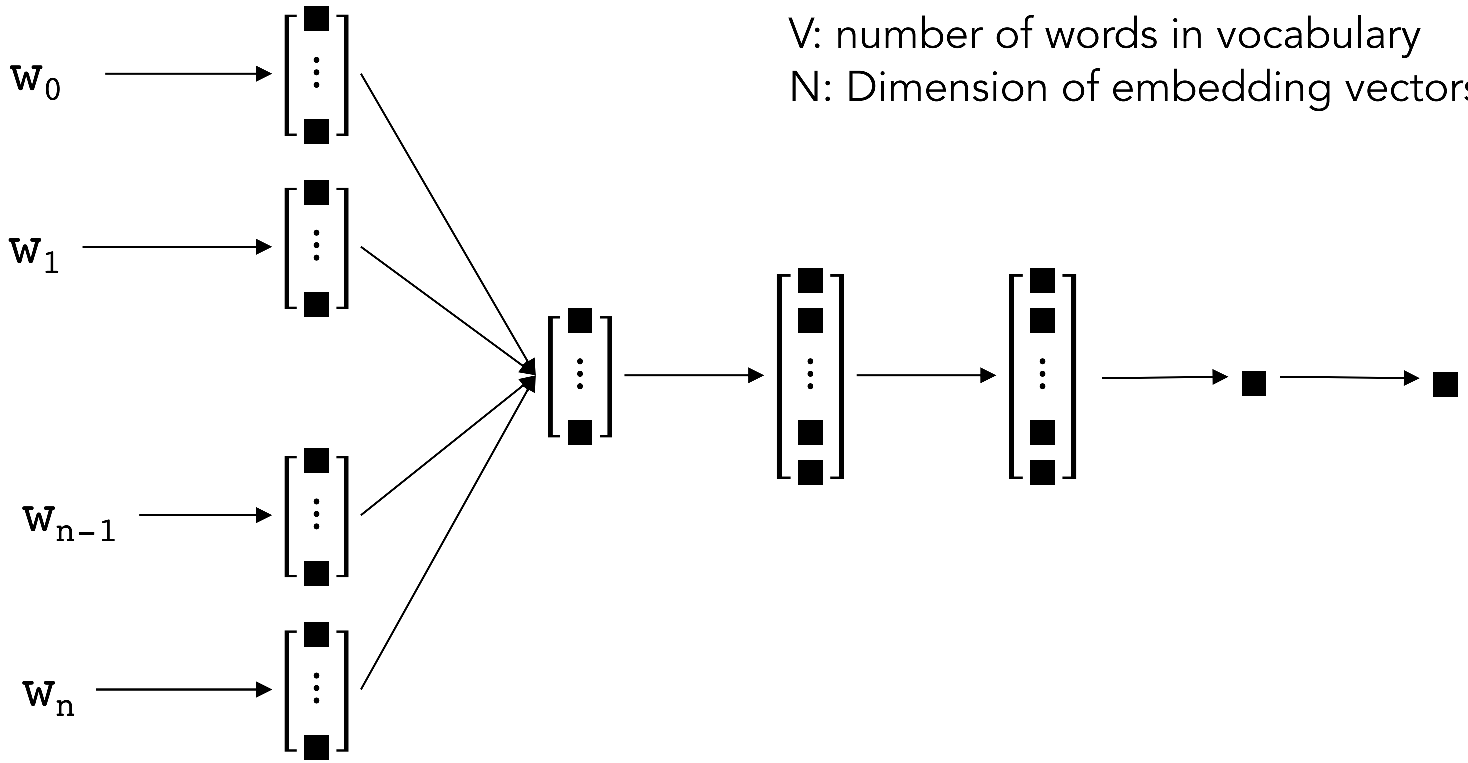
Derrick Higgins

TEXT CATEGORIZATION WITH NEURAL NETWORKS

Text categorization with neural networks

- Neural networks
  - Use vector/matrix/tensor representations
  - Apply a sequence of algebraic operations (matrix multiplication, etc.)
  - Trained using some variant of gradient descent
- Text categorization architecture
  - Bag of words representation (embeddings) at input layer (for now)
  - One or more hidden layers ("fully-connected" layers)
  - Probabilistic output layer: logistic sigmoid (binary classification) or softmax (multiclass classification)

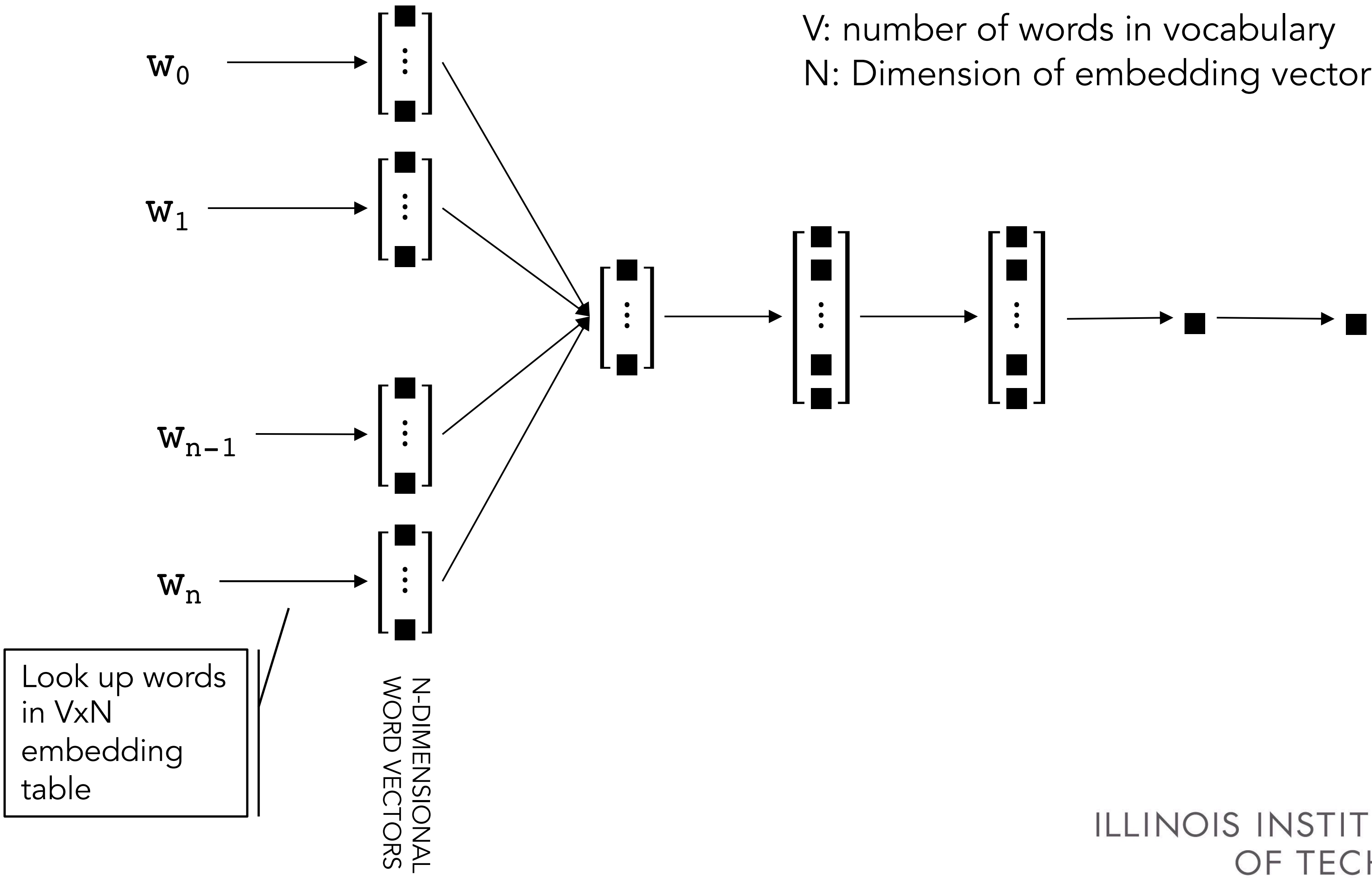
Binary text classification model – feed-forward neural network



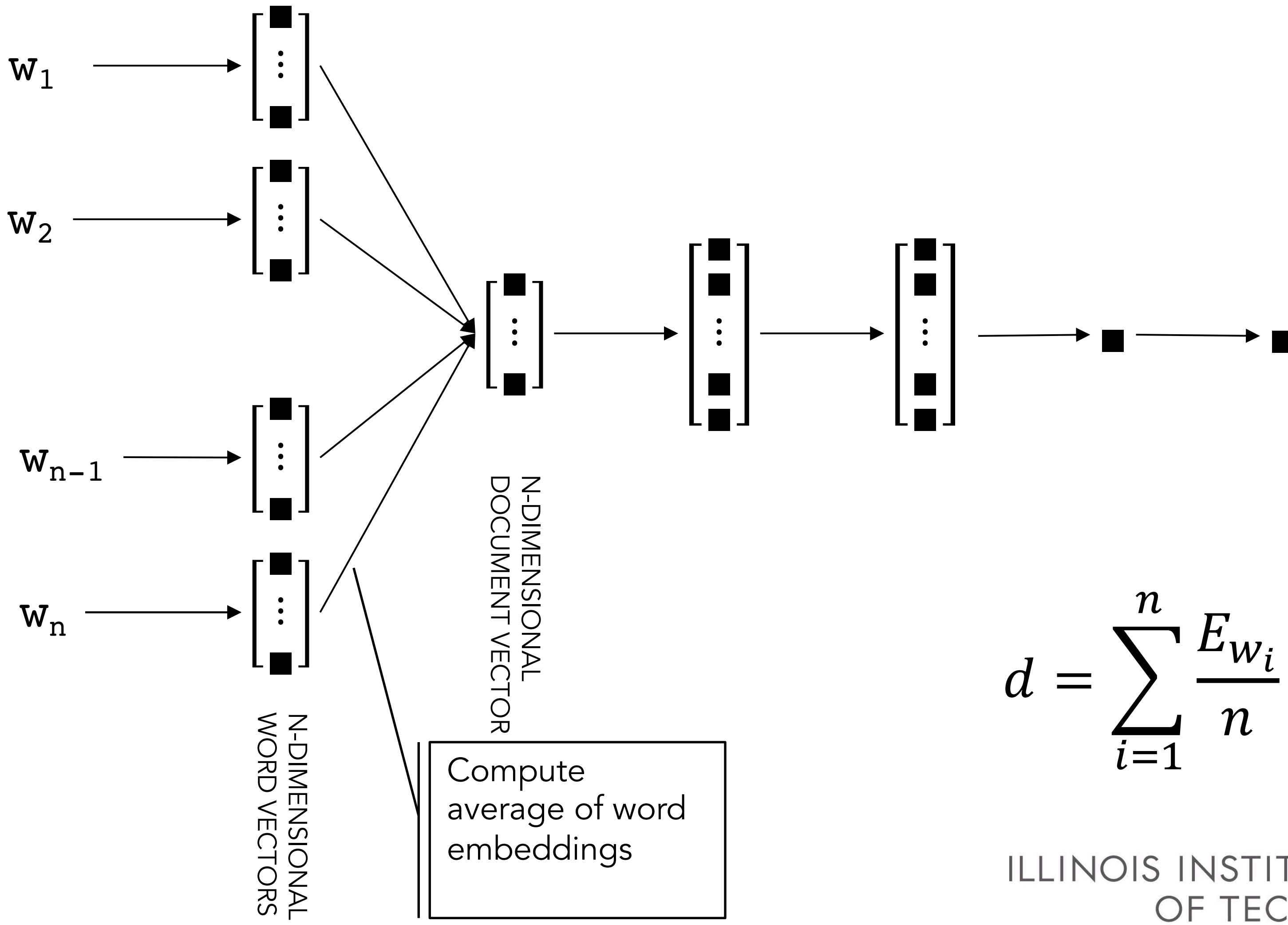
V: number of words in vocabulary
N: Dimension of embedding vectors

Binary text classification model – feed-forward neural network

V: number of words in vocabulary
N: Dimension of embedding vectors

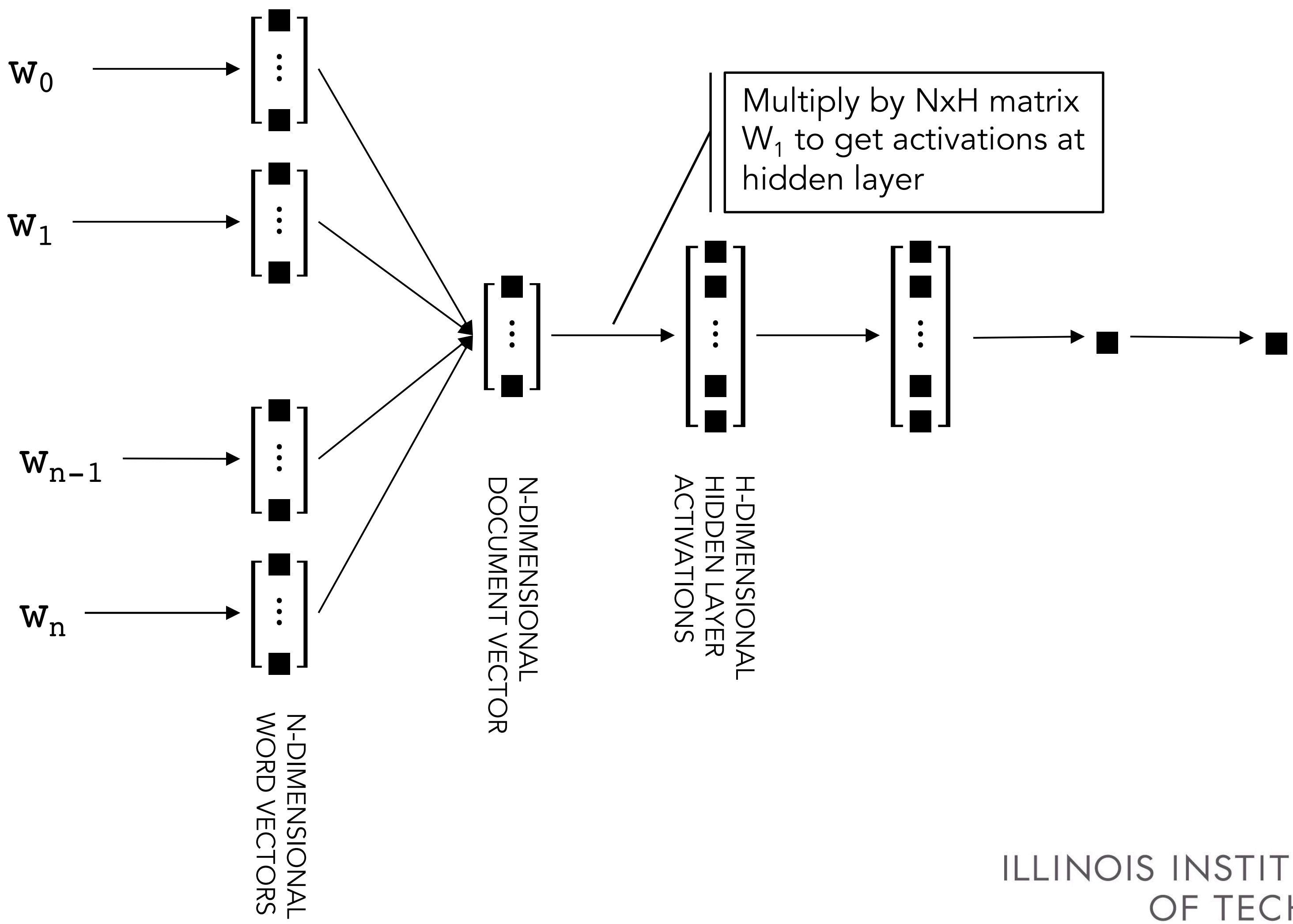


Binary text classification model – feed-forward neural network

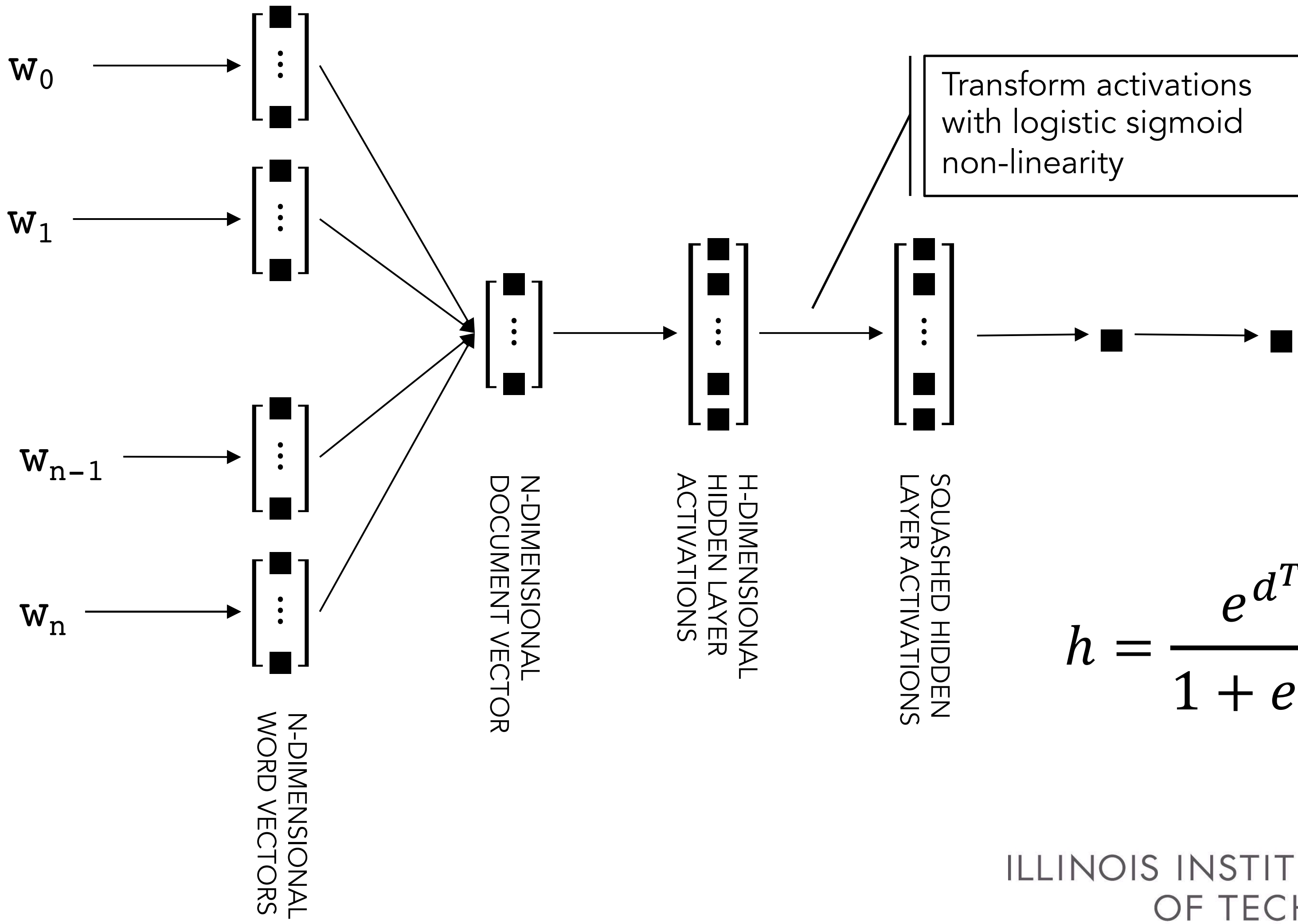


$$d = \sum_{i=1}^{n} \frac{E_{w_i}}{n}$$

Binary text classification model – feed-forward neural network

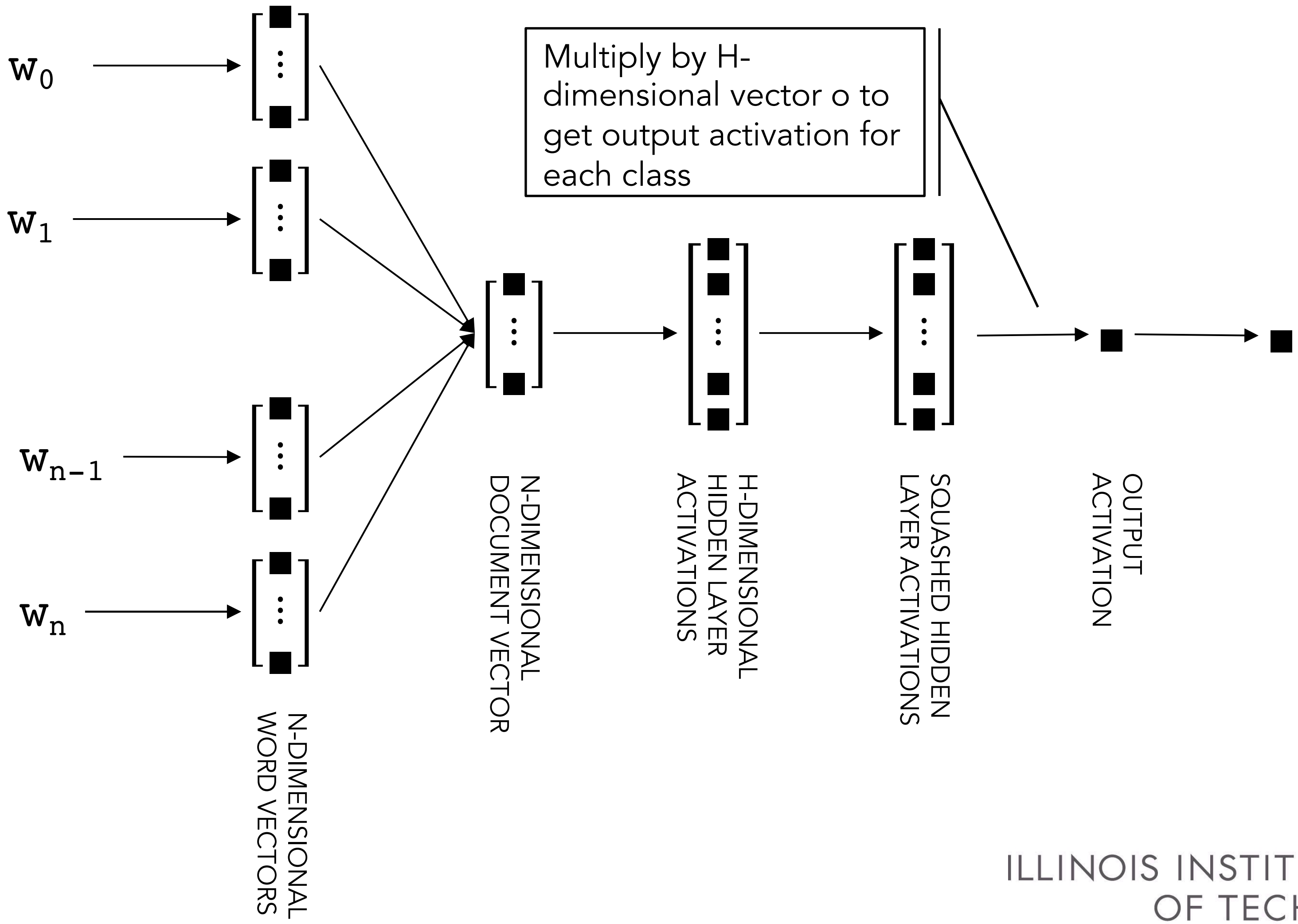


Binary text classification model – feed-forward neural network

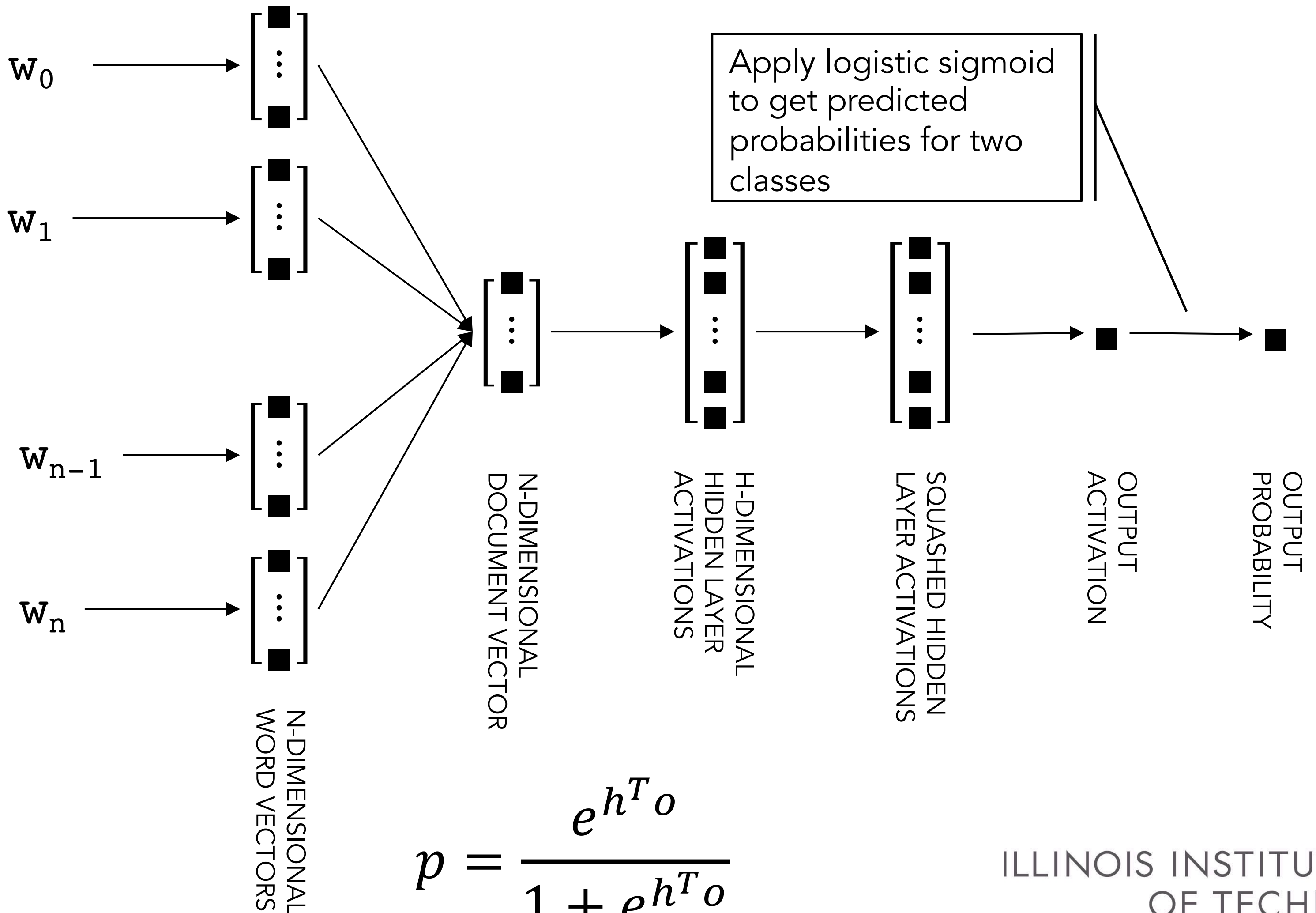


$$h = \frac{e^{d^T W_1}}{1 + e^{d^T W_1}}$$

Binary text classification model – feed-forward neural network



Binary text classification model – feed-forward neural network



$$p = \frac{e^{h^T o}}{1 + e^{h^T o}}$$

Feed-forward text categorization network: summary

Words → document representation

$$d = \sum_{i=1}^{n} \frac{E_{w_i}}{n}$$

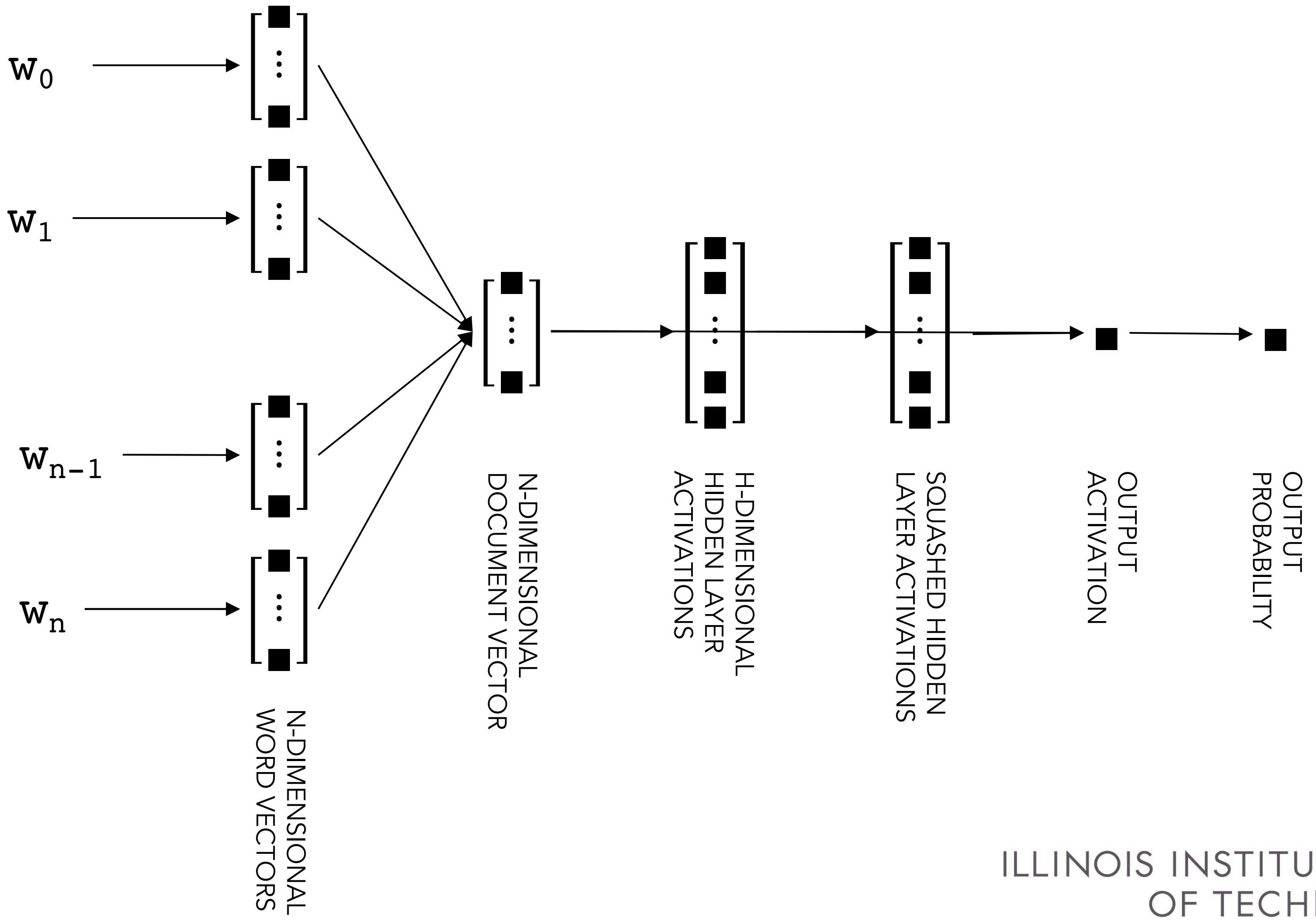
Document representation → hidden layer

$$h = Sigmoid(d^T W_1) = \frac{e^{d^T W_1}}{1 + e^{d^T W_1}}$$

Hidden layer → output probability

$$p = Sigmoid(h^T o) = \frac{e^{h^T o}}{1 + e^{h^T o}}$$

Comparison to logistic regression



Logistic regression: summary

Words → document representation

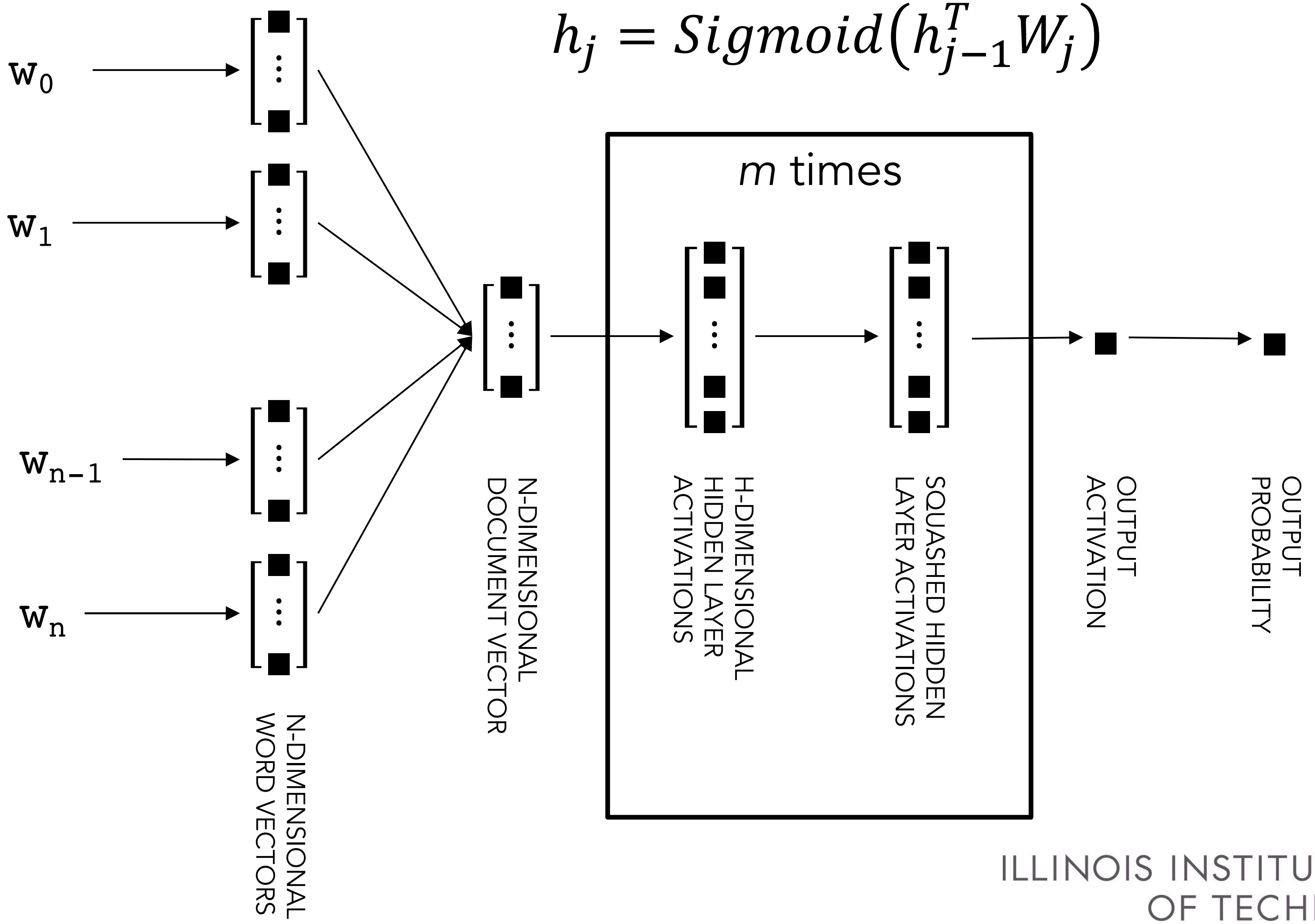
$$d = \sum_{i=1}^{n} \frac{E_{w_i}}{n}$$

Document representation → output probability

$$p = Sigmoid(d^T o) = \frac{e^{d^T o}}{1 + e^{d^T o}}$$

Feed-forward neural network: multiple hidden layers

$$h_j = Sigmoid\left(h_{j-1}^T W_j\right)$$



What is the point of nonlinearities?

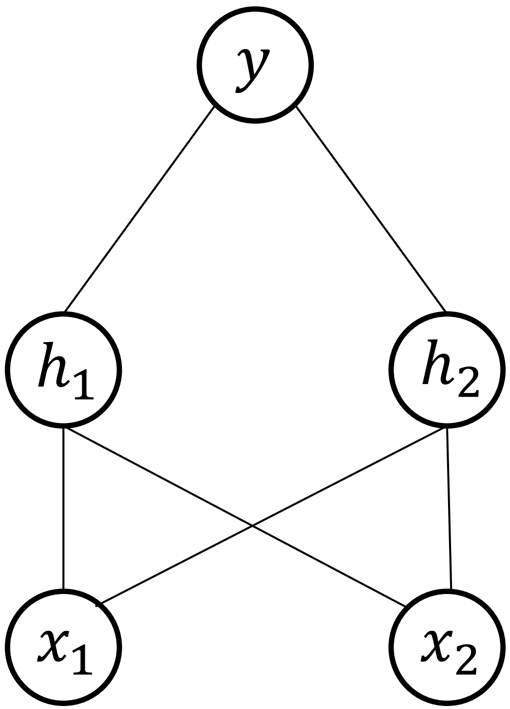
Expressive capacity

- Some functions cannot be expressed / represented / learned without them

XOR

- A well-known example is the exclusive OR (XOR) problem
- Given two input nodes $x_1, x_2 \in \{0,1\}$ and a single output node y it is impossible to set hidden layer weights such that $y = 1$ iff $x_1 \neq x_2$

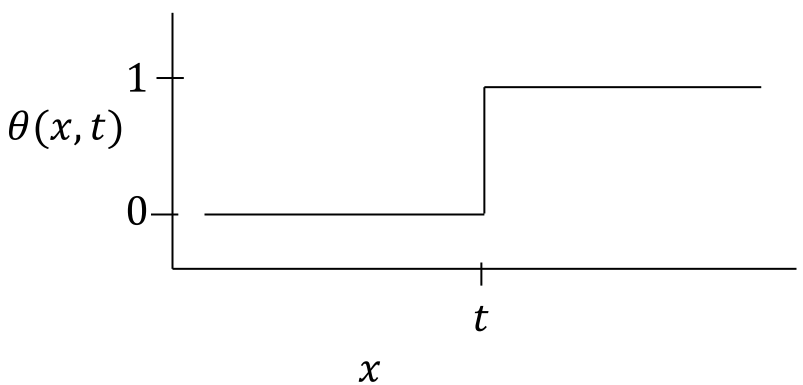
$$h_1 = w_1^1 x_1 + w_2^1 x_1$$
$$h_2 = w_1^2 x_2 + w_2^2 x_2$$
$$y = o_1 h_1 + o_2 h_2$$

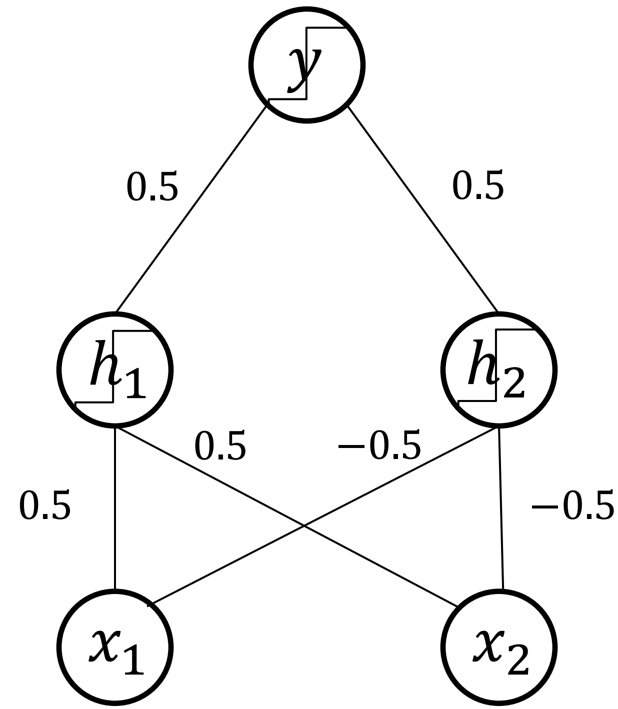


XOR

- With a nonlinearity in the network the function can be represented

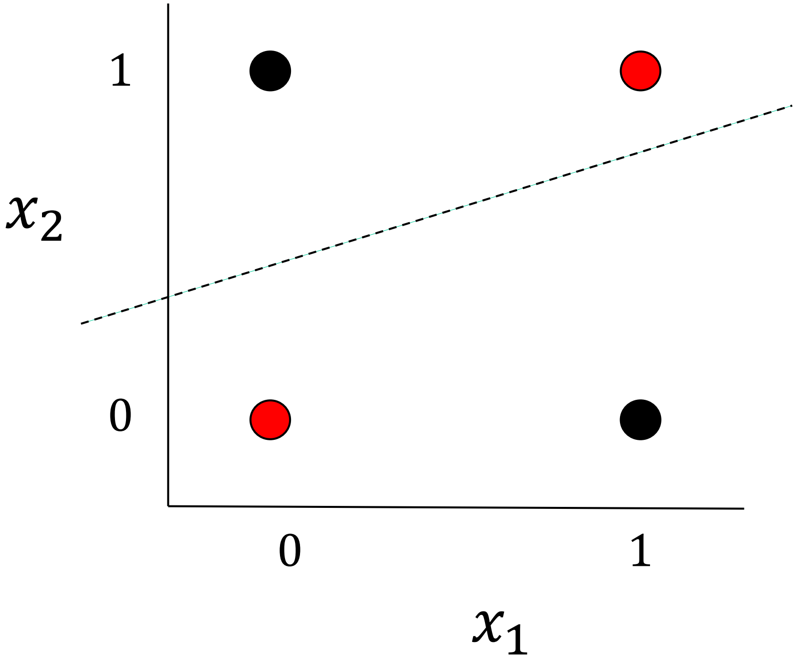
$$h_1 = \theta(w_1^1 x_1 + w_2^1 x_1, 0.25)$$
$$h_2 = \theta(w_1^2 x_2 + w_2^2 x_2, -0.75)$$
$$y = \theta(o_1 h_1 + o_2 h_2, 0.75)$$





XOR

- Matrix multiplication is just a linear operation, and XOR requires a non-linear decision boundary



Expressive capacity of neural networks

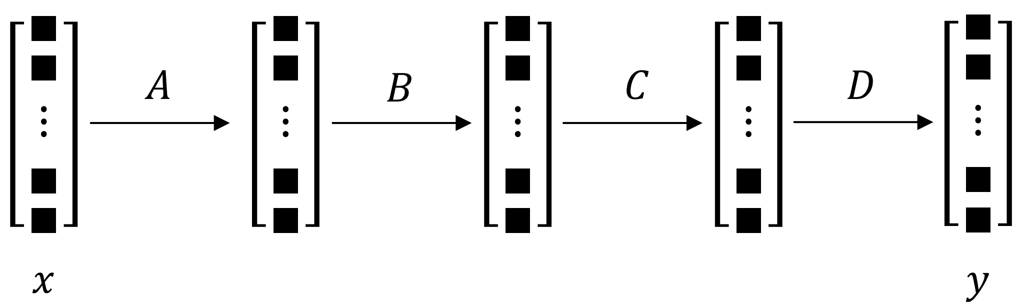
- A feed-forward neural network with fully-connected layers, at least one hidden layer with nonlinear activations (such as sigmoid) can represent any function of its inputs with arbitrary precision
  - Depends only on number of hidden nodes in the network
- Can be thought of as a “universal function approximator”

What is the point of multiple hidden layers?

- A network with a single hidden layer can represent any function as well as a network with multiple hidden layers
- **But** it may require an exponentially greater number of nodes
- Deeper networks are better for representing complex relationships between inputs and outputs
- **But** they can introduce difficulties for optimization
  - Regularization helps
  - Also residual connections (advanced topic)

Hidden layers are only useful with nonlinearities

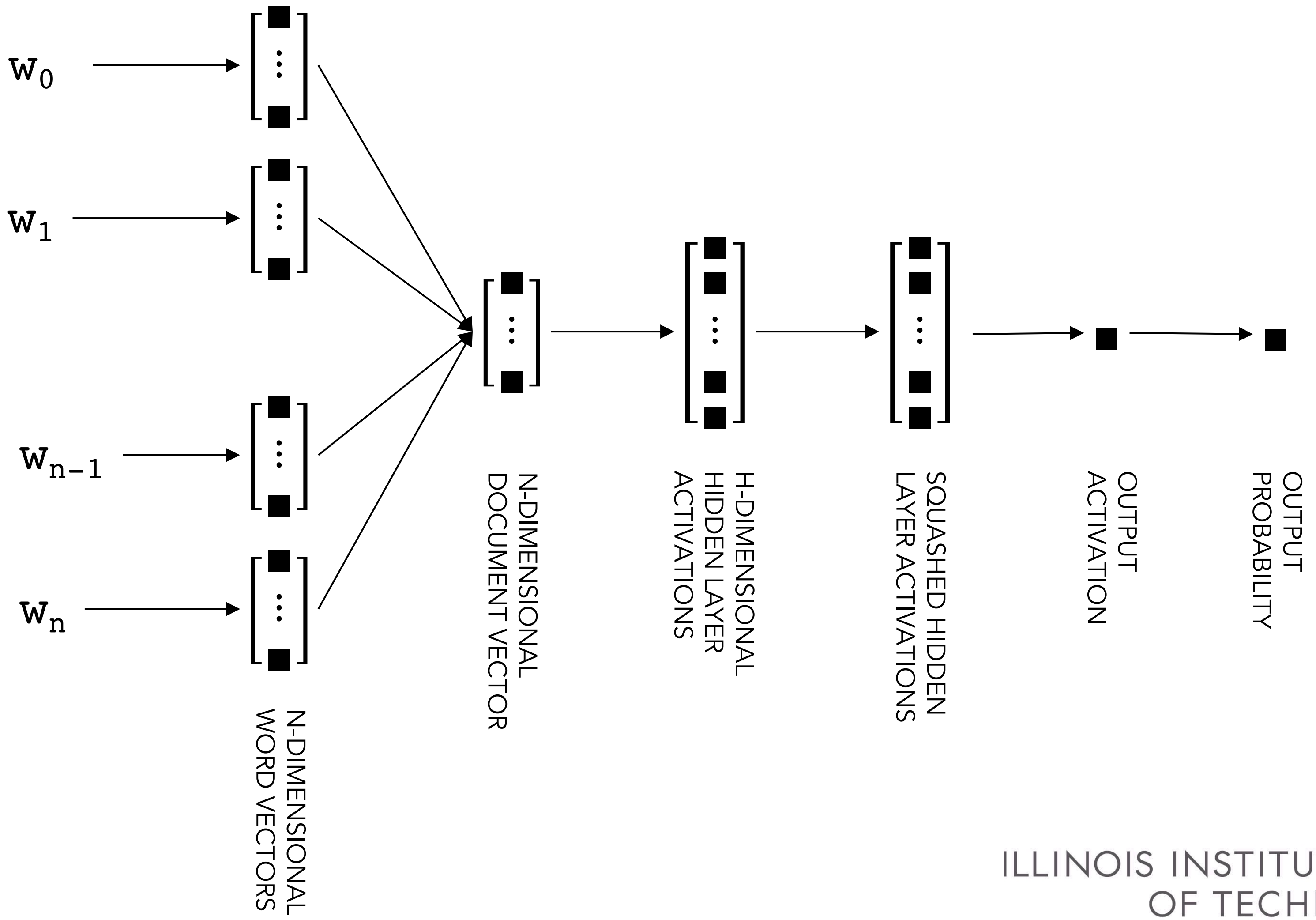
- Remember that without nonlinear activation functions, each layer of a feedforward neural network is just a linear transform of the previous layer (matrix multiplication)
- And successive matrix multiplications are always expressible as a single matrix multiplication



$$y = x^T ABCD = x^T(ABCD)$$

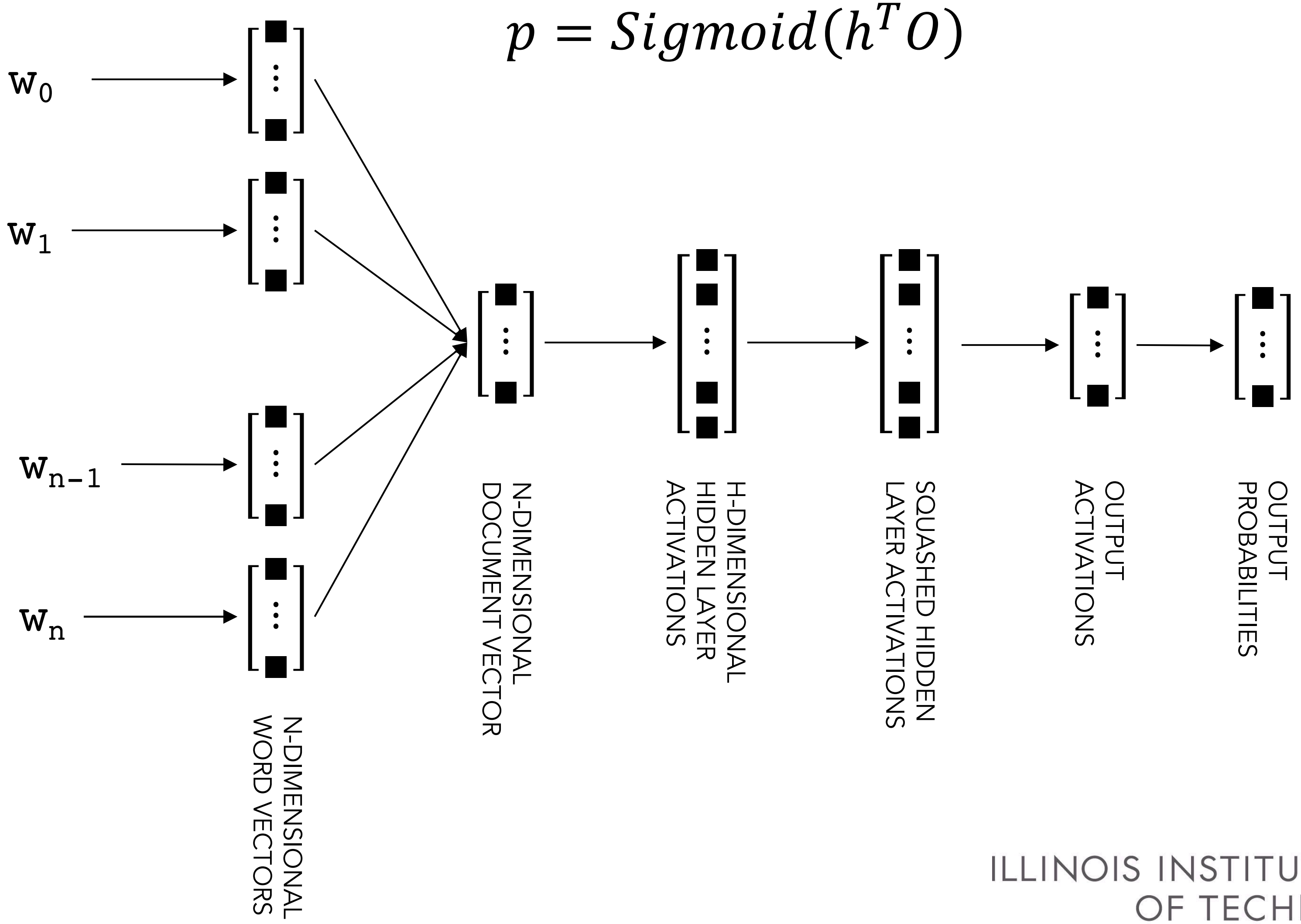
$$M \stackrel{\text{def}}{=} ABCD$$
$$y = x^T M$$

Multilabel classification



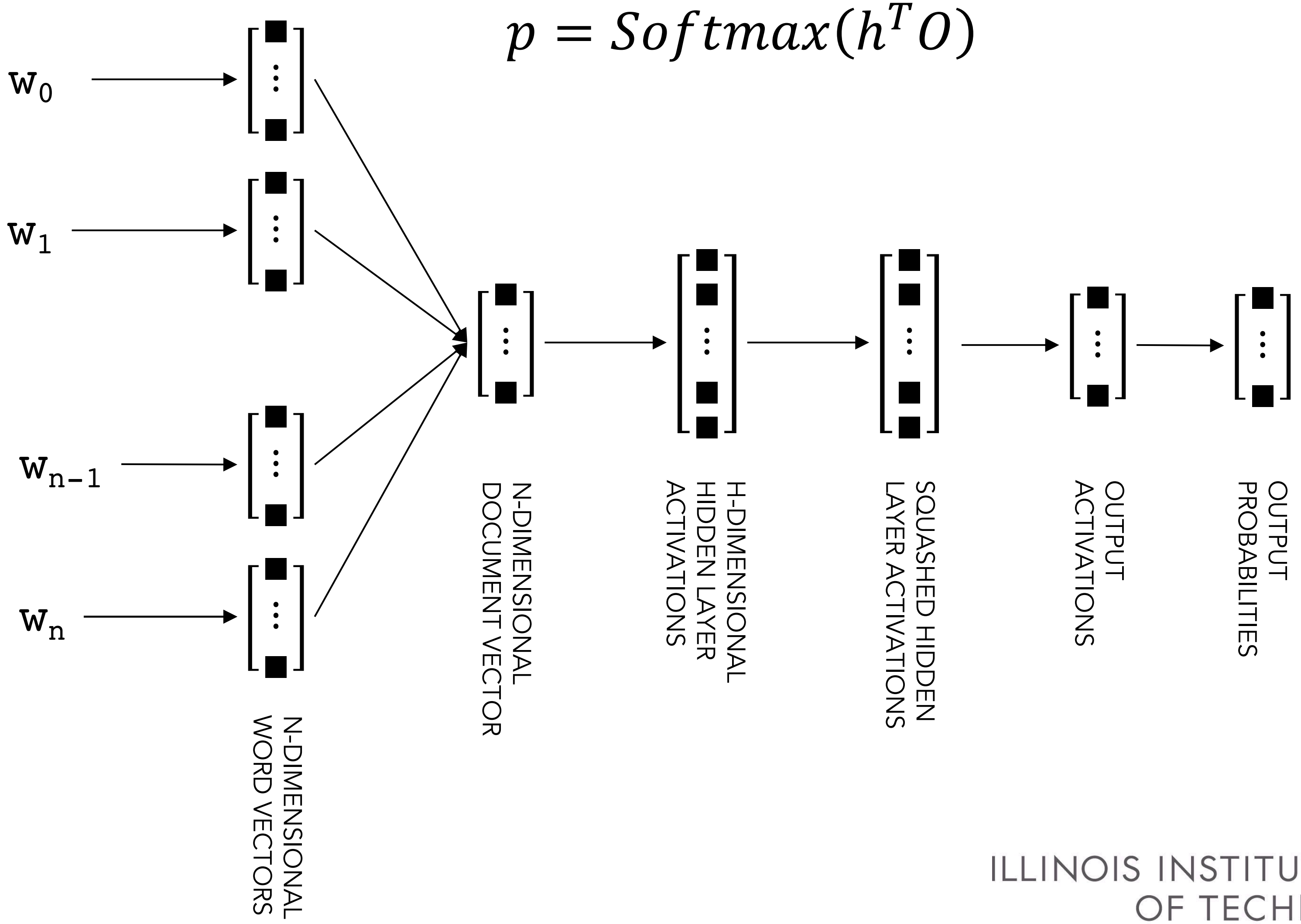
Multilabel classification

$$p = Sigmoid(h^T O)$$



Multiclass classification

$$p = Softmax(h^T O)$$



Multilabel vs. multiclass

- Multilabel classification
  - Labels are not mutually exclusive
  - Probabilities do not sum to one
  - Logistic sigmoid nonlinearity at output layer
- Multiclass classification
  - Labels are mutually exclusive
  - Probabilities must sum to one
  - Softmax nonlinearity at output layer

NEURAL NETWORK TOOL CHEST

Nonlinearities

- Softmax and logistic sigmoid are the most common nonlinearities used in neural networks for NLP, but there are a few others to be familiar with.
- The general constraints on nonlinearities (or *activation functions*) is that they be monotonic (continuously increasing or decreasing) and differentiable
- The primary nonlinear functions used are
  - Softmax
  - Logistic sigmoid
  - Hyperbolic tangent (`tanh`)
  - Rectified linear (ReLU)

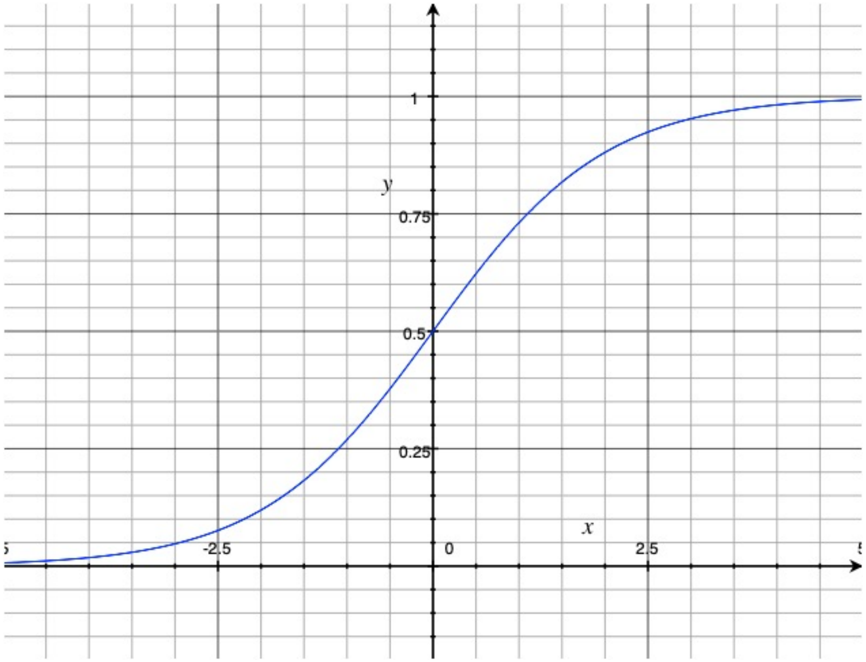
Nonlinearities: softmax

- Softmax is typically only used at the output layer of a network in order to get probabilistic/normalized outputs for multiclass classification problems
- It is sometimes treated as part of the loss function, rather than part of the network per se.

$$Softmax(\vec{x}) = \left[\frac{e^{\vec{x}_i}}{\sum_{\forall j} e^{\vec{x}_j}}\right]_{\forall i}$$

Nonlinearities: logistic sigmoid

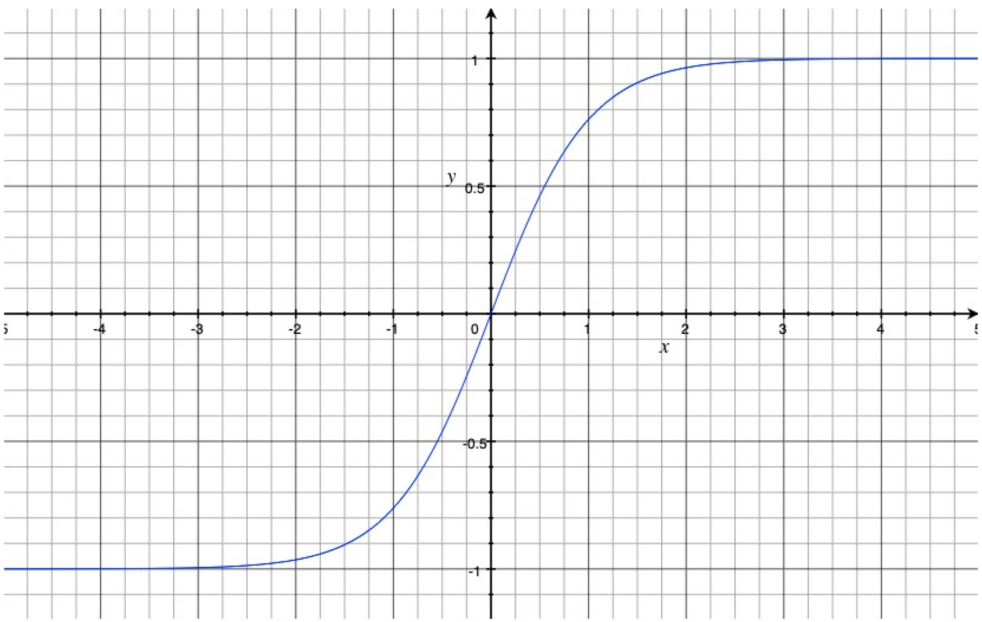
- Most commonly-used activation function for hidden layer of network
- Also at output layer for binary classification tasks
- Produces activations constrained to range $[0,1]$



$$Sigmoid(\vec{x}) = \frac{e^{\vec{x}}}{1 + e^{\vec{x}}}$$

Nonlinearities: hyperbolic tangent

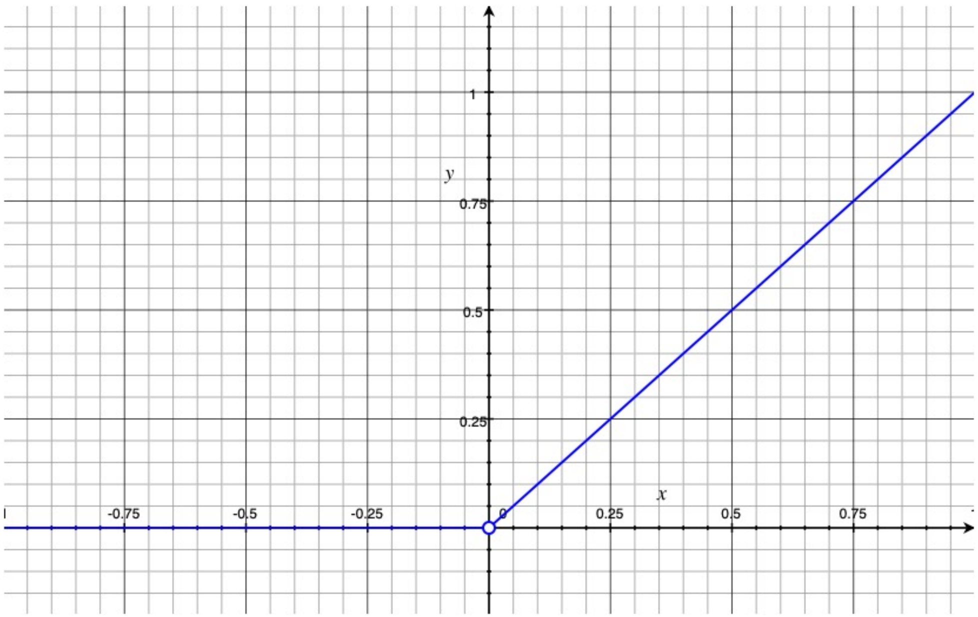
- Sigmoid-like activation function that allows negative outputs
- Used in LSTMs (later this semester)
- Produces activations constrained to range $[-1,1]$



$$\text{htan}(\vec{x}) = \frac{e^{\vec{x}} - e^{-\vec{x}}}{e^{\vec{x}} + e^{-\vec{x}}}$$

Nonlinearities: ReLU

- *Re*ctified *L*inear *U*nit
- Produces sparse activations (many zeroes)
- Technically not differentiable at 0, but can be dealt with computationally
- Produces activations constrained to range $[0, \infty]$



$$rectifier(\vec{x}) = \max(\vec{x}, 0)$$

Loss functions

Loss function	Usage	Formula
Binary cross-entropy	Binary or multilabel classification	$\mathcal{L} = -y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i)$
Categorical cross-entropy	Multiclass classification	$\mathcal{L} = -\sum_i y_i \log \hat{y}_i$
Squared error	Regression (prediction of a real-valued output)	$\mathcal{L} = \sum_i (y_i - \hat{y}_i)^2$

Also hinge, Huber, absolute error…

Logits

- The logistic sigmoid function is also referred to as the *inverse logit* function

$$Sigmoid(a) = b \Leftrightarrow Logit(b) = a$$

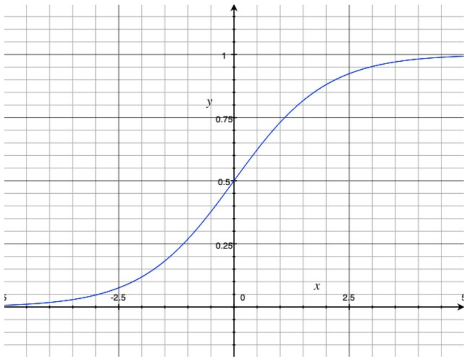
- The logit function translates probabilities into *log odds*

$$Logit(p) = \log \frac{p}{1-p}$$

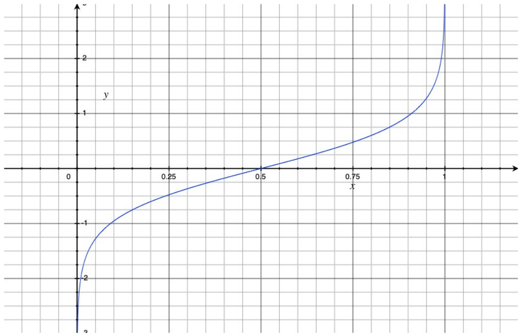
Logits

p	*Odds ratio* $\frac{p}{1-p}$	*Logits* $\ln\frac{p}{1-p}$
0.10	0.11	-2.20
0.25	0.33	-1.10
0.50	1.00	0.00
0.75	3.00	1.10
0.90	9.00	2.20

Sigmoid



Logit



THE ART OF NETWORK ENGINEERING

Parameters and Hyperparameters

- Parameters: model-internal values that are set through training in order to optimize against some loss function
  - Examples: word embeddings, weight matrices between network layers
- Hyperparameters: model architecture or optimization decisions that are fixed in advance of training
  - Examples: learning rate, number of hidden layers, number of nodes per layer, regularization hyperparameters

Hyperparameters in neural networks

- Many model types have hyperparameters
  - Naïve Bayes – Smoothing hyperparameter
  - Logistic Regression – L1/L2 penalty
  - KNN – k neighbors
- But neural networks have a *lot* of them. How to search?
  - Choose a value and hope for the best
  - Search many values and select the best one based on development data
- Performance may also vary across training runs with a different random seed

Regularization in neural networks

- Regularization: discouraging or regulating model complexity
  - Especially important for neural networks due to the *curse of dimensionality*
- In a high-dimensional space, there are many possible parameterizations (decision surfaces) that have equivalent performance according to our loss function (perhaps perfect accuracy on the training set)

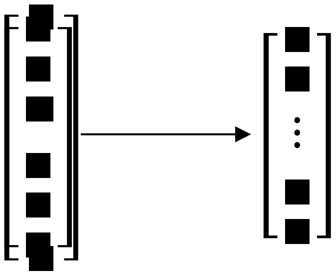
Regularization in neural networks

- L1 and L2 penalties we learned about in connection with logistic regression are used in neural networks as well
  - Different regularization penalties may be associated with weights at different layers
- Another regularization technique is *early stopping*—halting training before the loss has been fully minimized
  - Crude, but can be simpler than tweaking hyperparameters to get the desired result
  - Monitor performance on development dataset

Regularization in neural networks

Dropout is a regularization technique specific to neural networks

- During training, a fixed percentage of outputs at each layer are randomly set to zero
- This introduces noise into the inputs of the next layer, discouraging large weights
- It also discourages "co-adaptation" nodes in a layer that jointly perform a single function and can cause training to stall in a local minimum



Frozen and tied weights

- In a neural network, some weights may be fixed, rather than updating in the course of training. These are referred to as *frozen*.
  - For instance, word embeddings from word2vec may be used at the input layer of the network, but not updated in training a task-specific model
  - Alternatively, the embedding weights may be further refined through task-specific training. This is called *fine-tuning*
- *Tied* or *shared weights* are constrained to be the same within a network.
  - For instance, we could build a network to classify pairs of documents, and constrain the portions of the network specific to a single document to be the same across both.

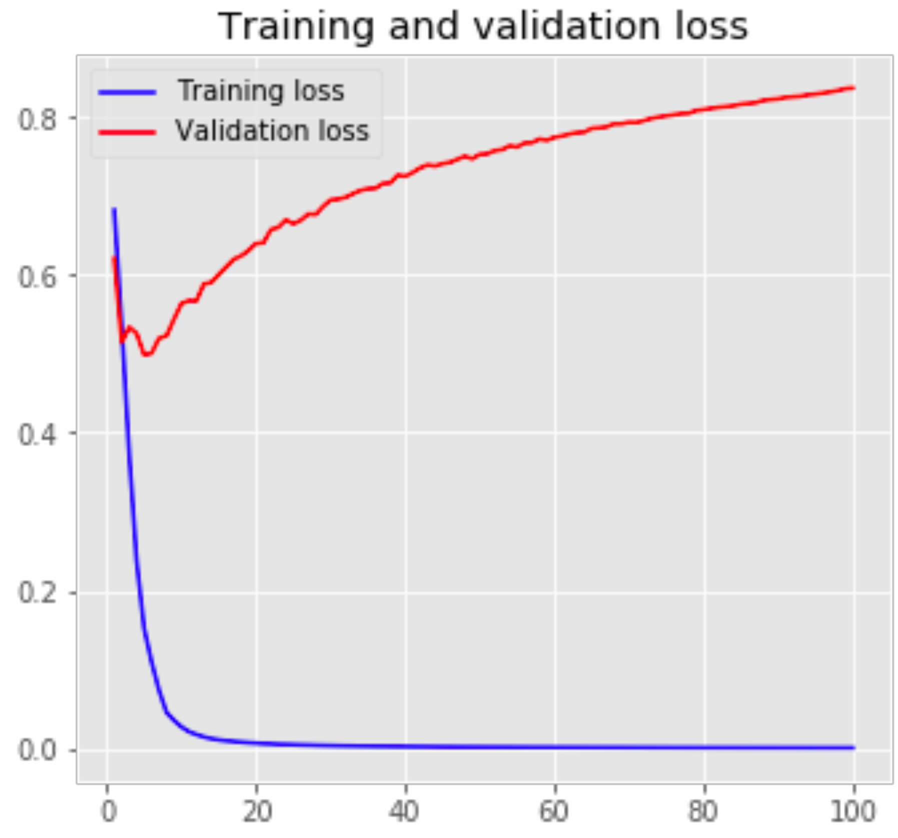
TROUBLESHOOTING NEURAL NETWORKS

Evaluation

- Neural networks have great expressive capacity
  - Therefore, we need to ensure that we monitor performance on held-out data to avoid overtraining
- Neural networks are difficult to optimize – non-convex error functions with local minima
  - Therefore, we need to monitor performance to ensure convergence

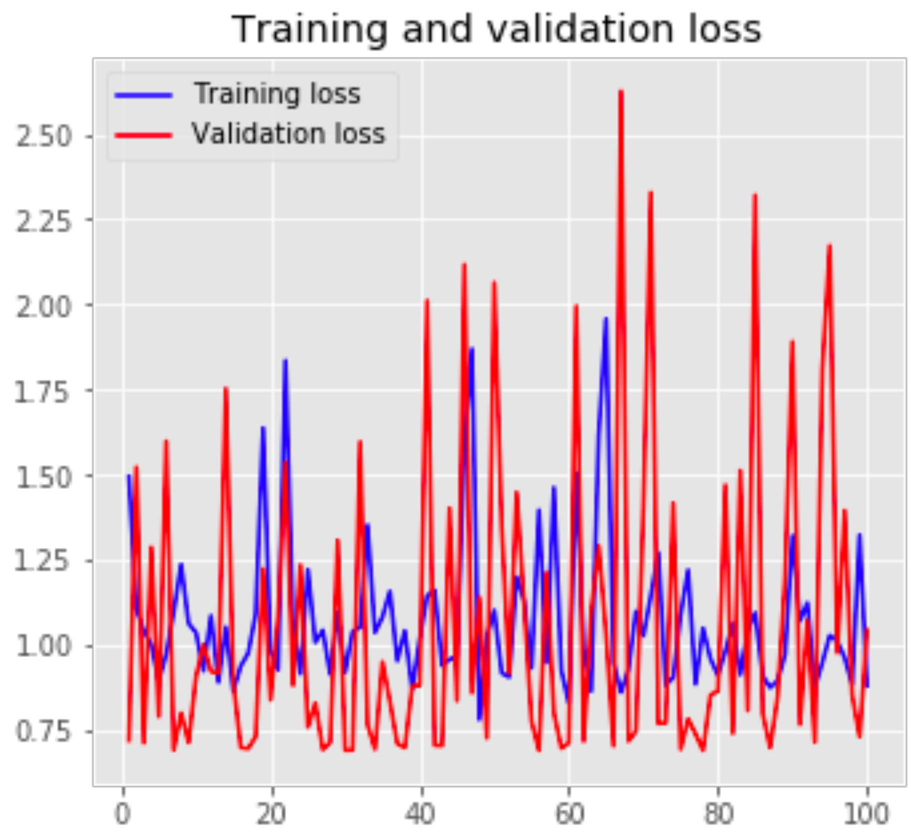
Common issues: overtraining

- Monitor performance on held-out set



Common issues: non-convergence

- Reduce learning rate
- If convergence is too slow, increase learning rate



Common issues: model complexity

- Start simple – remember, logistic regression is a neural network
- A single-layer bag-of-words model is a strong baseline!