# TTIC 31190:
# Natural Language Processing

Kevin Gimpel

Winter 2016

## Lecture 10:
## Neural Networks for NLP

# Announcements

- Assignment 2 due Friday
- project proposal due Tuesday, Feb. 16
- midterm on Thursday, Feb. 18

# Roadmap

- classification
- words
- lexical semantics
- language modeling
- sequence labeling
- neural network methods in NLP
- syntax and syntactic parsing
- semantic compositionality
- semantic parsing
- unsupervised learning
- machine translation and other applications

# What is a neural network?

- just think of a neural network as a function
- it has inputs and outputs
- the term “neural” typically means a particular type of functional building block (“neural layers”), but the term has expanded to mean many things

# Classifier Framework

$$\text{classify}(\boldsymbol{x}, \boldsymbol{\theta}) = \underset{\boldsymbol{y} \in \mathcal{L}}{\text{argmax}} \ \text{score}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$$

- linear model score function:

$$\text{score}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}) = \sum_i \theta_i f_i(\boldsymbol{x}, \boldsymbol{y})$$

- we can also use a neural network for the score function!

# **neural layer** = affine transform + nonlinearity

$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

nonlinearity

affine transform

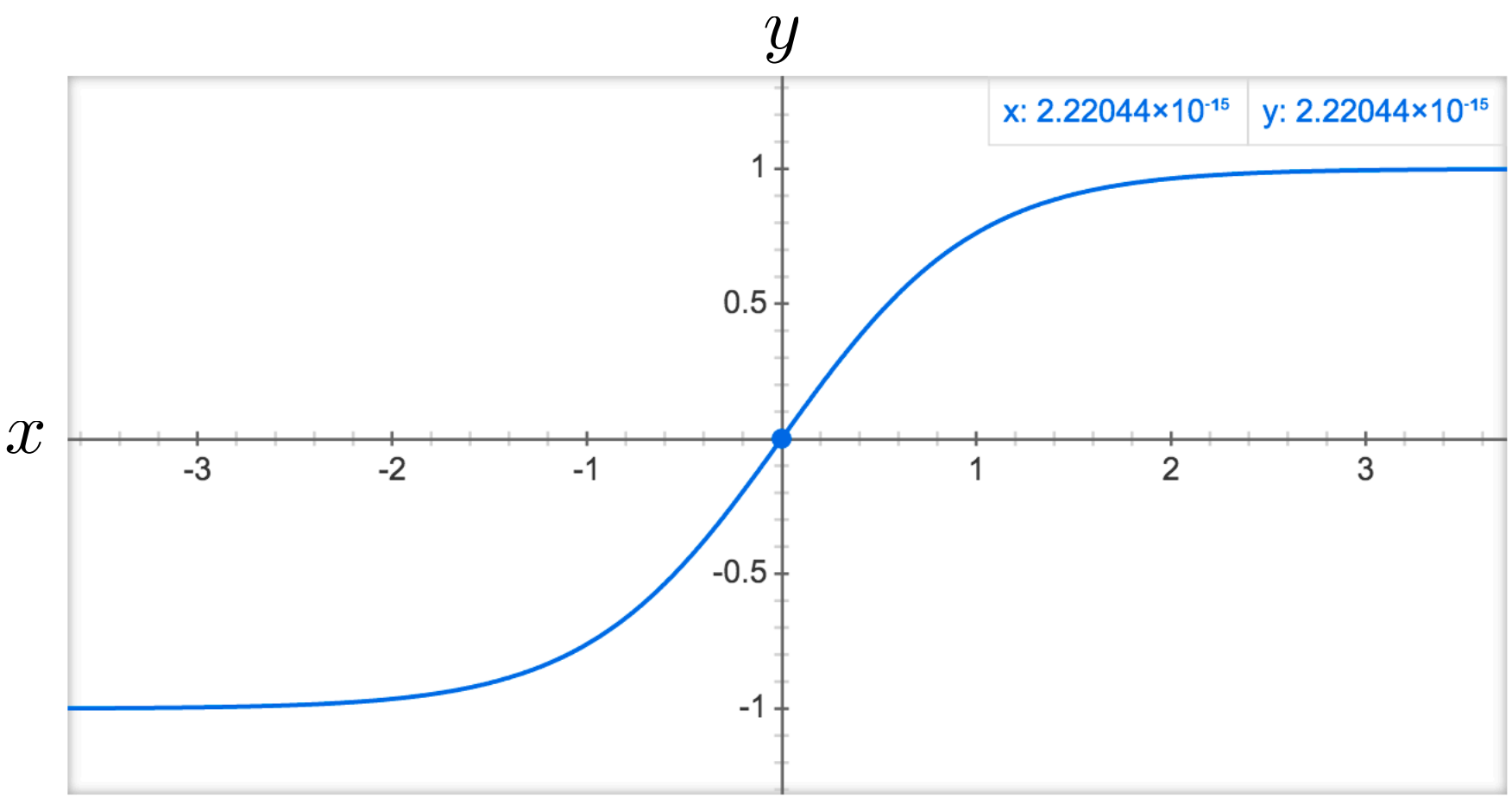
- this is a single "layer" of a neural network
- input vector is $\boldsymbol{x}$
- vector of "hidden units" is $\boldsymbol{z}^{(1)}$

# Nonlinearities

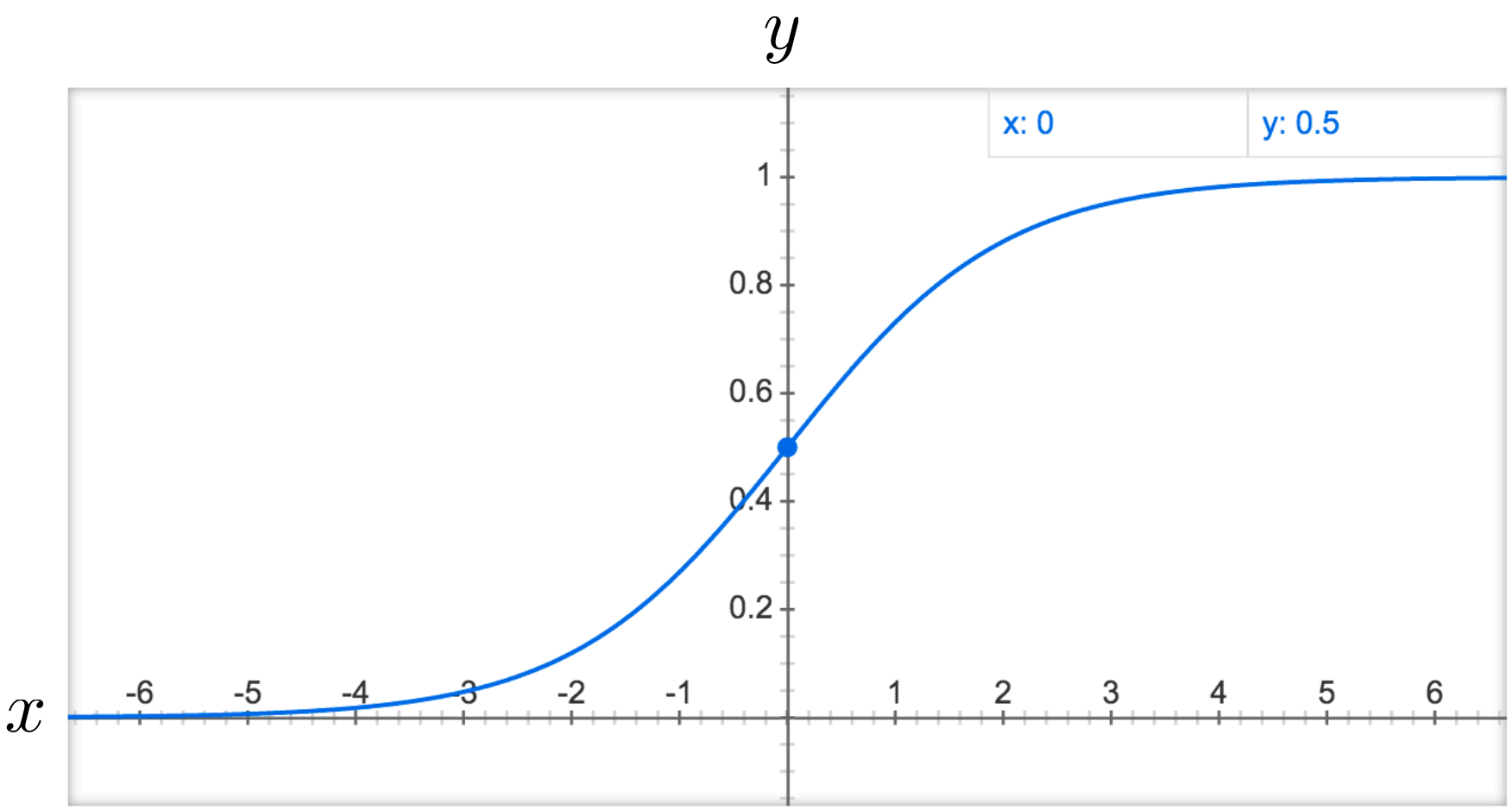
$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

- most common: elementwise application of *g* function to each entry in vector
- examples…

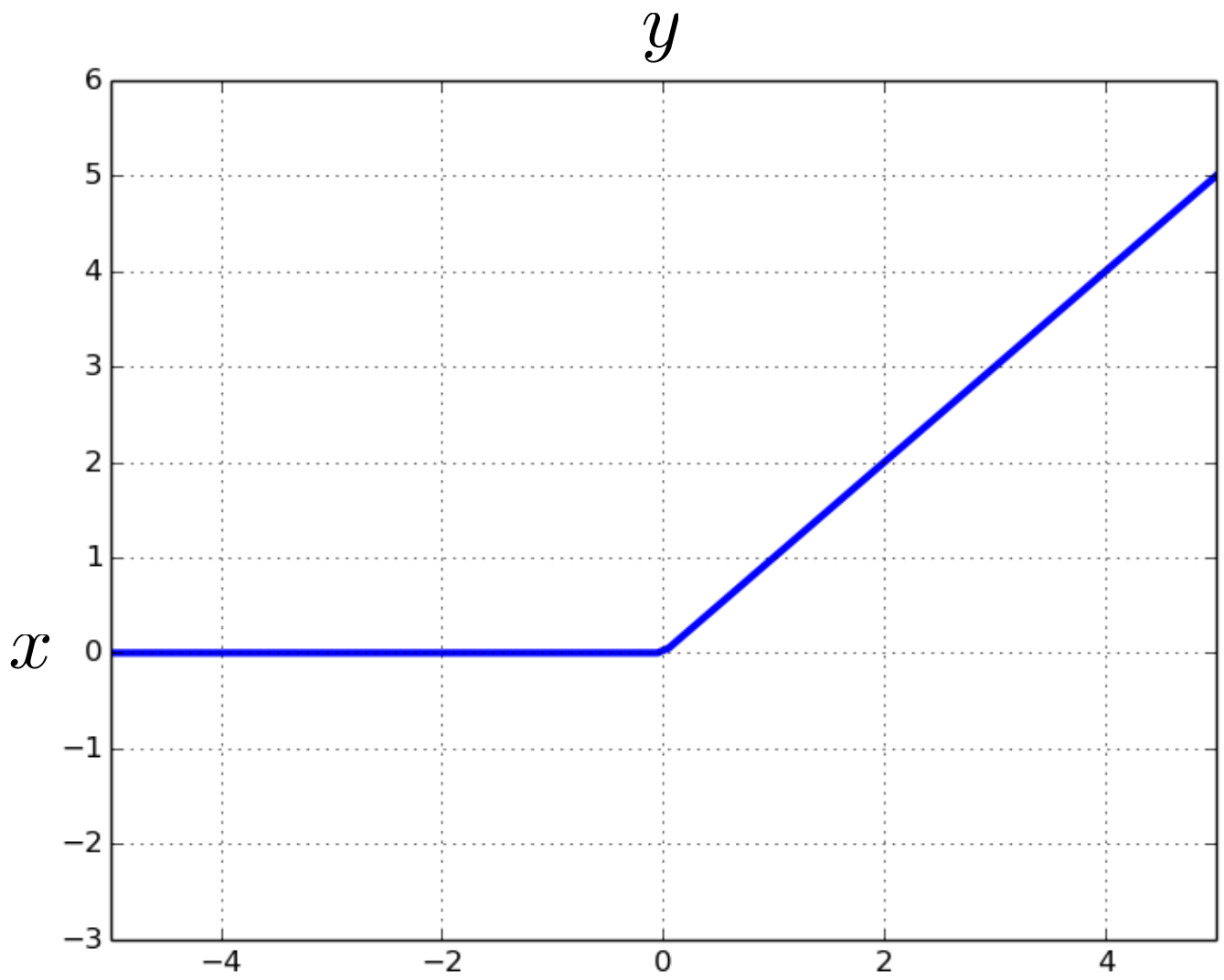
tanh: $y = \tanh(x)$



(logistic) sigmoid: $$y = \frac{1}{1 + \exp\{-x\}}$$



rectified linear unit (ReLU): $y = \max(0, x)$



# 2-layer network

$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

$$\boldsymbol{s} = g\left(W^{(1)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(1)}\right)$$

vector of label scores

- this is a 2-layer neural network
- input vector is $\boldsymbol{x}$
- output vector is $\boldsymbol{s}$

# 2-layer neural network for sentiment classification

$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

$$\boldsymbol{s} = g\left(W^{(1)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

Use softmax function to convert scores into probabilities

$$\text{softmax}(\boldsymbol{s}) = \begin{bmatrix} \frac{\exp\{s_1\}}{\sum_i \exp\{s_i\}} \\ \cdots \\ \frac{\exp\{s_d\}}{\sum_i \exp\{s_i\}} \end{bmatrix}$$

$$\boldsymbol{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

$$\boldsymbol{p} = \text{softmax}(\boldsymbol{s}) = \begin{bmatrix} \frac{\exp\{\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta})\}}{Z} \\ \frac{\exp\{\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta})\}}{Z} \end{bmatrix}$$

$$Z = \exp\{\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta})\} + \exp\{\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta})\}$$

# Why nonlinearities?

**2-layer network:**

$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

$$\boldsymbol{s} = g\left(W^{(1)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(1)}\right)$$

**written in a single equation:**

$$\boldsymbol{s} = g\left(W^{(1)}g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}\right)$$

- if g is linear, then we can rewrite the above as a single affine transform
- can you prove this? (use distributivity of matrix multiplication)

# Universal approximation theorem

From Wikipedia, the free encyclopedia

In the mathematical theory of artificial neural networks, the **universal approximation theorem** states[1] that a feed-forward network with a single hidden layer containing a finite number of neurons (i.e., a multilayer perceptron), can approximate continuous functions on compact subsets of $\mathbf{R}^n$, under mild assumptions on the activation function. The theorem thus states that simple neural networks can *represent* a wide variety of interesting functions when given appropriate parameters; however, it does not touch upon the algorithmic learnability of those parameters.

One of the first versions of the theorem was proved by George Cybenko in 1989 for sigmoid activation functions.[2]

Kurt Hornik showed in 1991[3] that it is not the specific choice of the activation function, but rather the multilayer feedforward architecture itself which gives neural networks the potential of being universal approximators. The output units are always assumed to be linear. For notational convenience, only the single output case will be shown. The general case can easily be deduced from the single output case.

# Understanding the score function

$$\boldsymbol{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

entry 2 of bias vector

$$\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) = s_1 = g\left(W^{(1)}_{1,*} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}_1\right)$$

$$\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) = s_2 = g\left(W^{(1)}_{2,*} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}_2\right)$$

row vector corresponding to row 2 of $W^{(1)}$

# Parameter sharing

$$\boldsymbol{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

**parameters NOT shared between labels**

$$\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) = s_1 = g\left(W_{1,*}^{(1)} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}_1^{(1)}\right)$$

$$\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) = s_2 = g\left(W_{2,*}^{(1)} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}_2^{(1)}\right)$$

**parameters shared between labels**

# Observation

- with linear models:
  - when using linear models for, say, sentiment classification, every feature included a label
  - **no** parameters were shared between labels
- with neural networks
  - we now have parameters shared across labels!
  - we still have some parameters that are devoted to particular labels
  - to define *x*, we design features that **only** look at the input (not at the labels)

# Defining input features

- say we're doing sentiment classification and we want to use a neural network
- what should ***x*** be?
  - it has to be independent of the label
  - it has to be a **fixed-length** vector

# Empirical Risk Minimization with Surrogate Loss Functions

- given training data: $\mathcal{T} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_{i=1}^{|\mathcal{T}|}$
  where each $y^{(i)} \in \mathcal{L}$ is a label
- we want to solve the following:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{|\mathcal{T}|} \operatorname{loss}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

many possible loss functions to consider optimizing

# Loss Functions

| name | loss | where used |
|---|---|---|
| cost ("0-1") | $\text{cost}(y, \text{classify}(\boldsymbol{x}, \boldsymbol{\theta}))$ | intractable, but underlies "direct error minimization" |
| perceptron | $-\text{score}(\boldsymbol{x}, y, \boldsymbol{\theta}) + \max_{y' \in \mathcal{L}} \text{score}(\boldsymbol{x}, y', \boldsymbol{\theta})$ | perceptron algorithm (Rosenblatt, 1958) |
| hinge | $-\text{score}(\boldsymbol{x}, y, \boldsymbol{\theta}) + \max_{y' \in \mathcal{L}} (\text{score}(\boldsymbol{x}, y', \boldsymbol{\theta}) + \text{cost}(y, y'))$ | support vector machines, other large-margin algorithms |
| log | $-\log p_{\boldsymbol{\theta}}(y \mid \boldsymbol{x})$ $= \text{score}(\boldsymbol{x}, y, \boldsymbol{\theta}) + \log \sum_{y' \in \mathcal{L}} \exp\{\text{score}(\boldsymbol{x}, y', \boldsymbol{\theta})\}$ | logistic regression, conditional random fields, maximum entropy models |

# (Sub)gradients of Losses for **Linear Models**

| name | entry *j* of (sub)gradient of loss for linear model |
|---|---|
| cost ("0-1") | not subdifferentiable in general |
| perceptron | $-f_j(\boldsymbol{x}, y) + f_j(\boldsymbol{x}, \hat{y}),\ \text{where } \hat{y} = \text{classify}(\boldsymbol{x}, \boldsymbol{\theta})$ |
| hinge | $-f_j(\boldsymbol{x}, y) + f_j(\boldsymbol{x}, \tilde{y}),\ \text{where } \tilde{y} = \text{costClassify}(\boldsymbol{x}, y, \boldsymbol{\theta})$ |
| log | $-f_j(\boldsymbol{x}, y) + \mathbb{E}_{p_{\boldsymbol{\theta}}(\cdot\|\boldsymbol{x})}[f_j(\boldsymbol{x}, \cdot)]$ |

# Learning with Neural Networks

$$\text{classify}(\boldsymbol{x}, \boldsymbol{\theta}) = \underset{\boldsymbol{y} \in \mathcal{L}}{\text{argmax}} \ \text{score}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$$

$$\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) = s_1 = g\left(W^{(1)}_{1,*} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}_1\right)$$

$$\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) = s_2 = g\left(W^{(1)}_{2,*} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}_2\right)$$

- we can use any of our loss functions from before, as long as we can compute (sub)gradients
- algorithm for doing this efficiently: **backpropagation**
- it's basically just the chain rule of derivatives

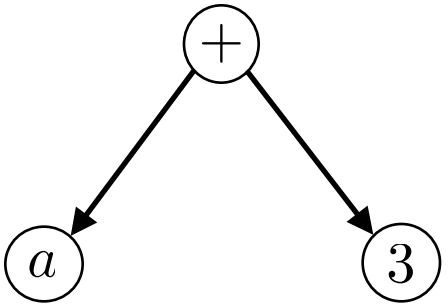
# Computation Graphs

- a useful way to represent the computations performed by a neural model (or any model!)
- why useful? makes it easy to implement automatic differentiation (backpropagation)
- many neural net toolkits let you define your model in terms of computation graphs (Theano, Torch, TensorFlow, CNTK, CNN, PENNE, etc.)

# Backpropagation

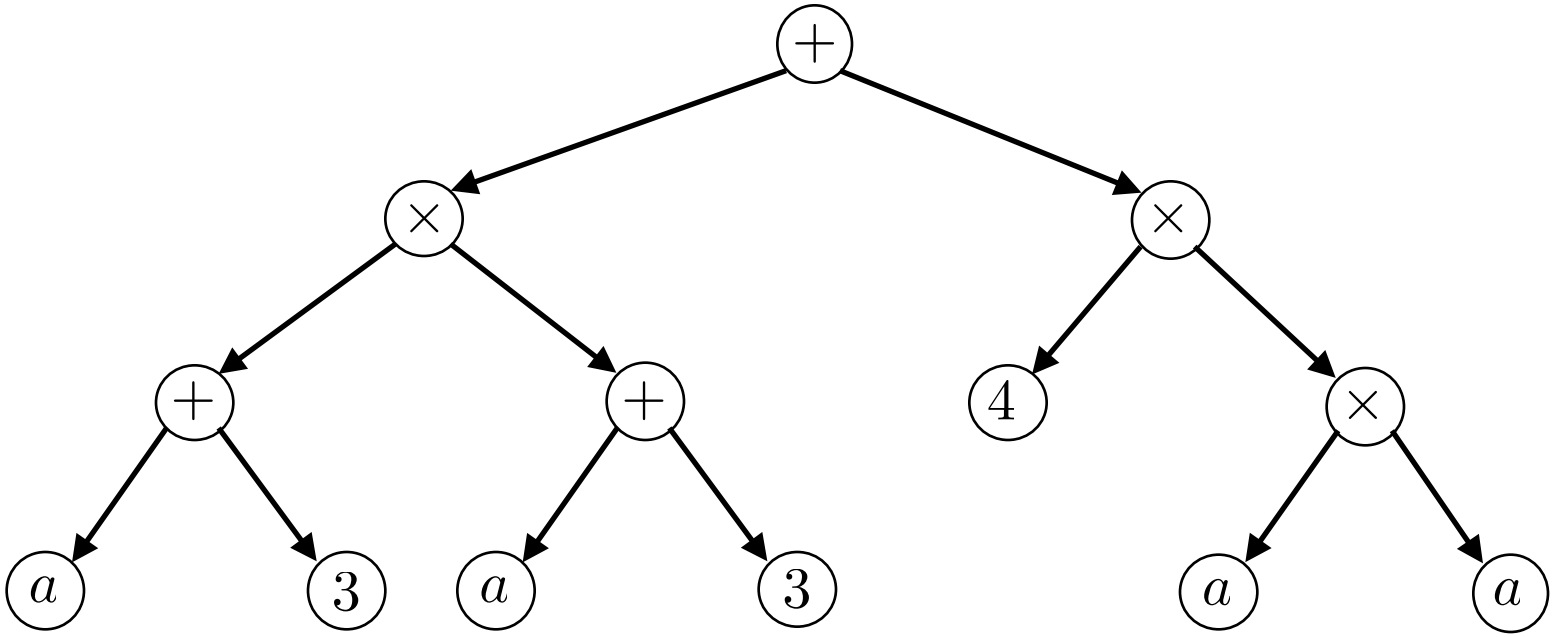
- backpropagation has become associated with neural networks, but it's much more general
- I also use backpropagation to compute gradients in **linear** models for structured prediction

# A simple computation graph:



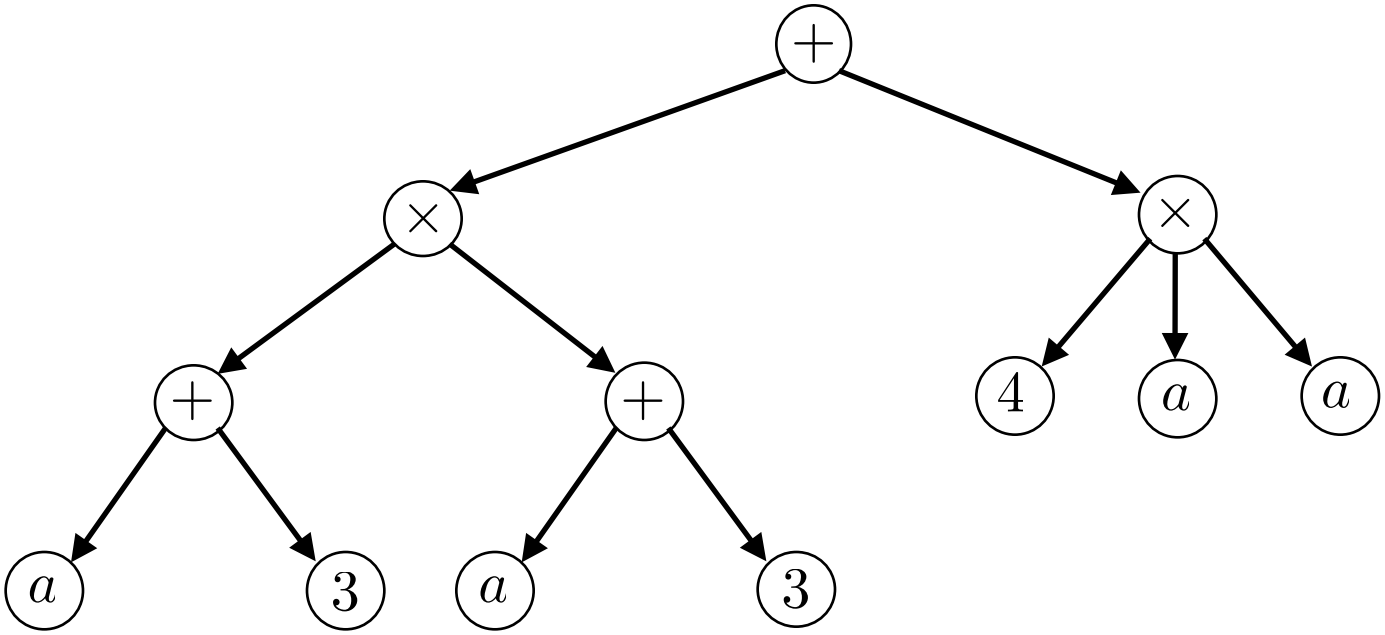

- represents expression “a + 3”

# A slightly bigger computation graph:

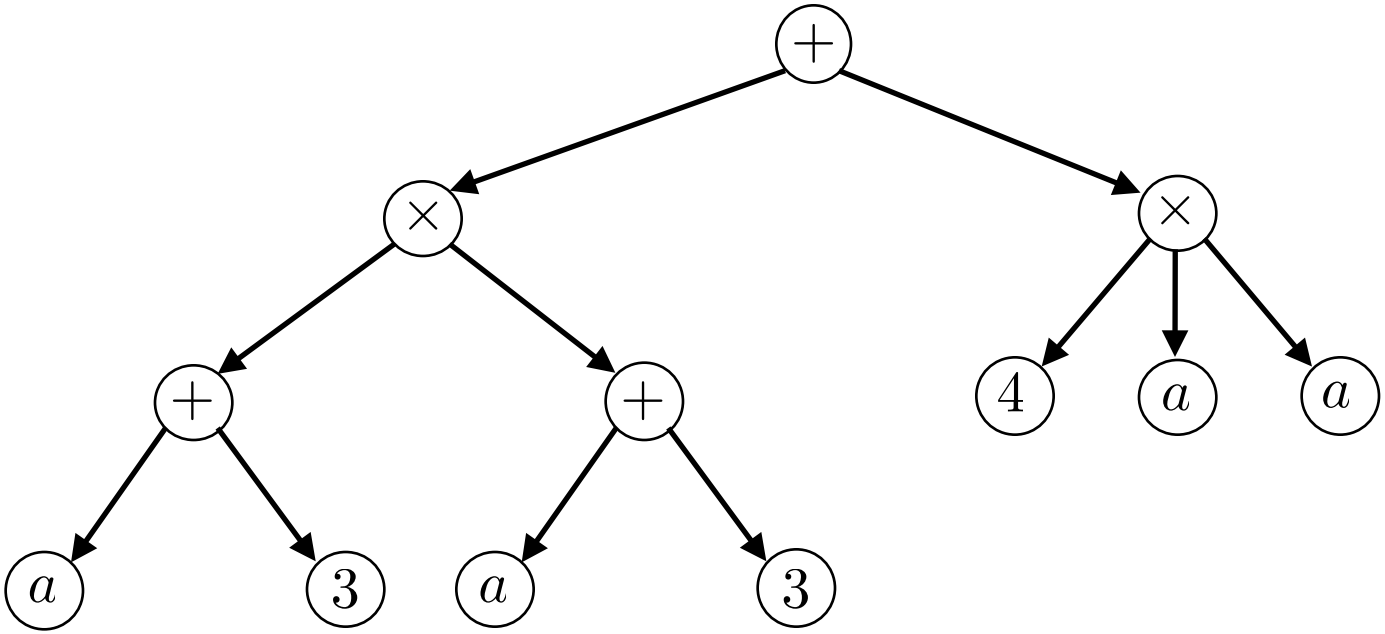



- represents expression "$(a + 3)^2 + 4a^2$"

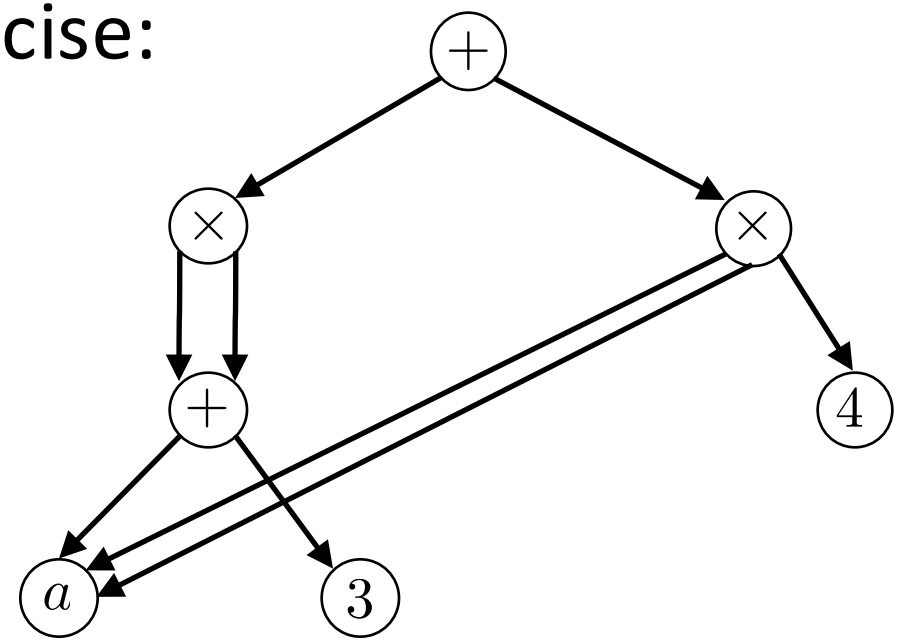
# Operators can have more than 2 operands:




- still represents expression "$(a + 3)^2 + 4a^2$"




- more concise:



# Overfitting & Regularization

- when we can fit any function, **overfitting** becomes a big concern
- overfitting: learning a model that does well on the training set but doesn't generalize to new data
- there are many strategies to reduce overfitting (we'll use the general term **regularization** for any such strategy)
- you used **early stopping** in Assignment 1, which is one kind of regularization

# Empirical Risk Minimization

- given training data: $\mathcal{T} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_{i=1}^{|\mathcal{T}|}$
  where each $y^{(i)} \in \mathcal{L}$ is a label
- we want to solve the following:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{|\mathcal{T}|} \operatorname{loss}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

# Regularized Empirical Risk Minimization

- given training data: $\mathcal{T} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_{i=1}^{|\mathcal{T}|}$

  where each $y^{(i)} \in \mathcal{L}$ is a label

- we want to solve the following:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{|\mathcal{T}|} \operatorname{loss}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) + \lambda R(\boldsymbol{\theta})$$

**regularization strength**

**regularization term**

# Regularization Terms

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{|\mathcal{T}|} \operatorname{loss}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) + \lambda R(\boldsymbol{\theta})$$

- most common: penalize large parameter values
- intuition: large parameters might be instances of overfitting
- examples:

**$L_2$ regularization**: $R_{\mathrm{L2}}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_2^2 = \sum_i \theta_i^2$

(also called Tikhonov regularization or ridge regression)

**$L_1$ regularization**: $R_{\mathrm{L1}}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_1 = \sum_i |\theta_i|$

(also called basis pursuit or LASSO)

# Regularization Terms

**$L_2$ regularization**: $R_{\mathrm{L2}}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_2^2 = \sum_i \theta_i^2$

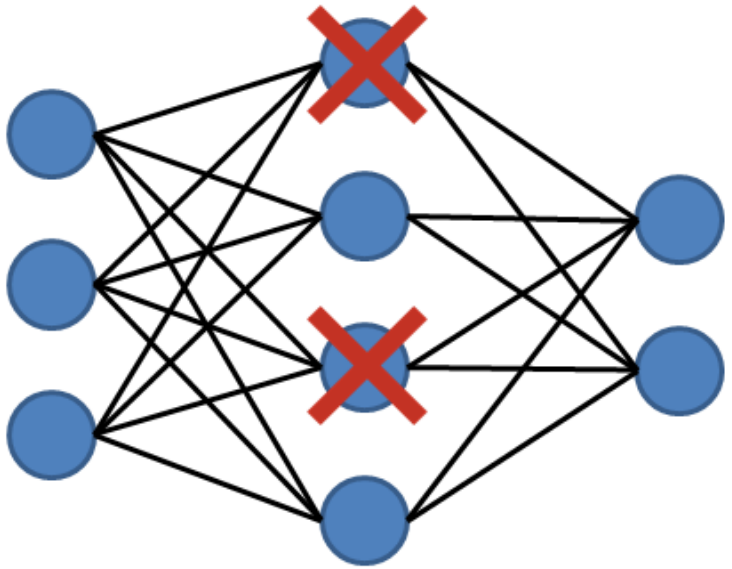
differentiable, widely-used

**$L_1$ regularization**: $R_{\mathrm{L1}}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_1 = \sum_i |\theta_i|$

not differentiable (but is subdifferentiable)
leads to sparse solutions (many parameters become zero!)

# Dropout

- popular regularization method for neural networks
- randomly “drop out” (set to zero) some of the vector entries in the layers



# Optimization Algorithms

- you used stochastic gradient descent (SGD) in Assignment 1
- but there are many other choices:
  - AdaGrad
  - AdaDelta
  - Adam
  - SGD with momentum
- we don't have time to go through these in class, but you should try using them! (most toolkits have implementations of these and others)