CS447: Natural Language Processing

http://courses.engr.illinois.edu/cs447

Lecture 9: Neural Nets for NLP

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Part 1: Overview

What have we covered so far?

We have covered a broad overview of some basic techniques in NLP:

- N-gram language models
- Logistic regression
- Word embeddings

Today, we'll put all of these together
to create a (much better) neural language model!

Today's class: Intro to neural nets

Part 1: Overview

Part 2: What are **neural nets**?

What are **feedforward** networks?
What is an **activation function**?
Why do we want activation functions to be **nonlinear**?

Part 3: **Neural n-gram models**

How can we use neural nets to model n-gram models?

How many parameters does such a model have?
Is this better than traditional n-gram models? Why? Why not?

What is “deep learning”?

Neural networks, typically with several hidden layers

- (depth = # of hidden layers)
- Single-layer neural nets are linear classifiers
- Multi-layer neural nets are more expressive

Very impressive performance gains in computer vision (ImageNet), speech recognition and NLP over the last decade.

Neural nets have been around for many decades.

Why did they suddenly make a comeback?

- Fast computers (GPUs!) and (very) large datasets have made it possible to train these very complex models.

Why deep learning in NLP?

NLP was slower to catch on to deep learning than e.g. computer vision.

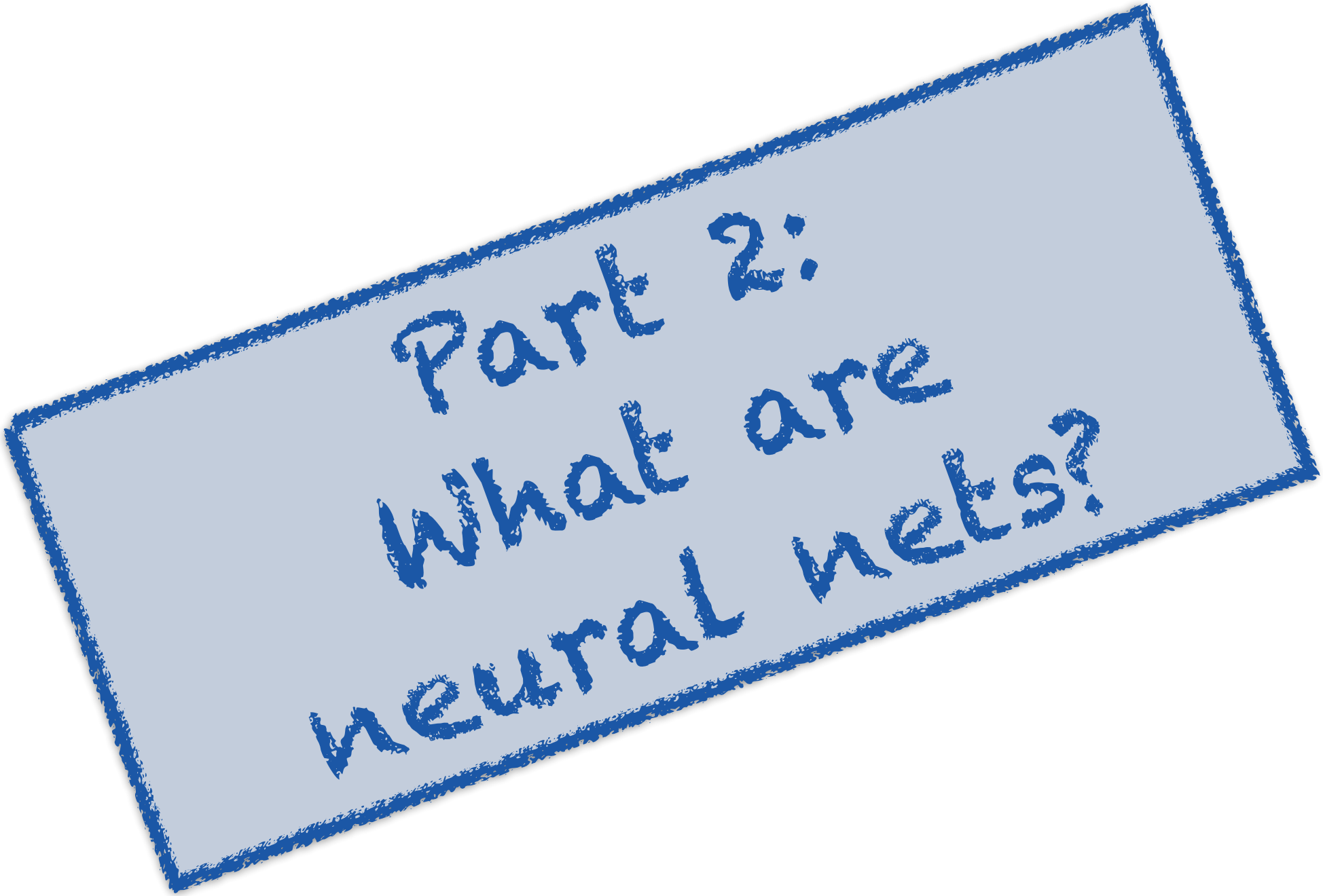
Language seems challenging for neural nets:

- Neural nets take (**real-valued**) **vectors as inputs**…
- … but language consists of **variable length sequences of discrete symbols**

But by now neural models have led to a similar fundamental paradigm shit in NLP.

We will talk about this a lot more later.

Today, we'll just cover some basics.


Part 2:
What are
neural nets?

What are neural networks?

A family of machine learning models that was originally inspired by how neurons (nerve cells) process information and learn.

In NLP, neural networks are now widely used, e.g. for

- **Classification**
  (e.g. sentiment analysis)
- **(Sequence) generation**
  (e.g. in machine translation, response generation for dialogue, etc.
- **Representation Learning** (neural embeddings)
  (word embeddings, sequence embeddings, graph embeddings,…)
- **Structure Prediction** (incl. sequence labeling)
  (e.g. part-of-speech tagging, named entity recognition, parsing,…)

The first computational neural networks: McCulloch & Pitts (1943)

Influential mathematical model of neural activity that aimed to capture the following assumptions:

- The neural system is a (directed) **network** of neurons
  (neurons = nerve cells)
- Neural activity consists of **electric impulses** that travel from neuron to neuron in this network
  Neurons receive input from other neurons
- Each neuron is **activated** (initiates an impulse)
  if the sum of the activations it receives (from other neurons)
  is above some **threshold ('all-or-none character')**
- This network of neurons may or may not have cycles
  (but the math is much easier without **cycles**)

The Perceptron (Rosenblatt 1958)

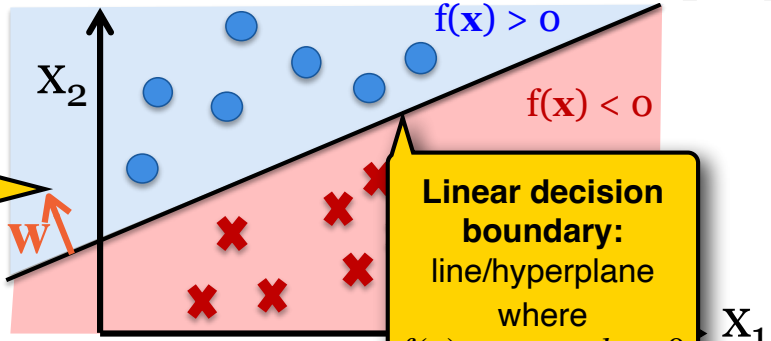
A **linear classifier** based on a **threshold activation** function:

Return $y = +1$ iff $f(\mathbf{x}) = \mathbf{w}\mathbf{x} + b > 0$

$y = -1$ iff $f(\mathbf{x}) = \mathbf{w}\mathbf{x} + b \leq 0$

$y \in \{-1, +1\}$ makes the **update rule** easier to write than $y \in \{0,1\}$

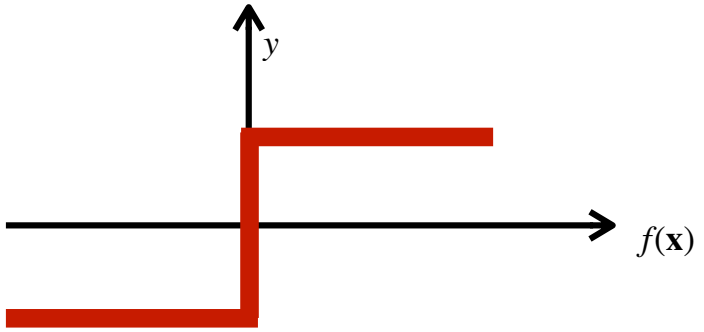
Linear classifier for $\mathbf{x} = (x_1, x_2)$



w is **orthogonal** to the **decision boundary**

Linear decision boundary: line/hyperplane where $f(\mathbf{x}) = \mathbf{w}\mathbf{x} + b = 0$

Threshold Activation



Threshold activation is inspired by the "all-or-none character" (McCulloch & Pitts, 1943) of how neurons process information

Perceptron update rule: (online stochastic gradient descent)

If the predicted $\hat{y}^{(i)} \neq y^{(i)}$: $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} + \eta y^{(i)} \mathbf{x}^{(i)}$

Increment w (**lower the slope** of the decision boundary) when y **should be** +1, decrement **w** when it should be -1)

Training: Change weights when the model makes a **mistake**

Notation for linear classifiers

Given N-dimensional inputs $\mathbf{x} = (x_1, \ldots, x_N)$:

With an explicit bias term b:

$$f(\mathbf{x}) = \mathbf{w}\mathbf{x} + b = \sum_{i=1}^{N} w_i x_i + b$$

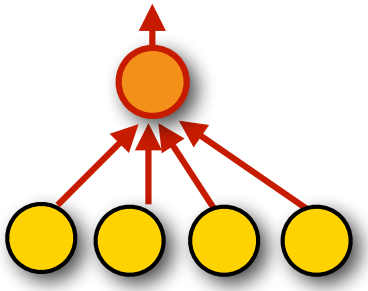
Without an explicit bias term b:

$$f(\mathbf{x}) = \mathbf{w}\mathbf{x} = \sum_{i=0}^{N} w_i x_i \qquad \text{where } x_0 = 1$$

(Decision boundary goes through origin of $(N+1)$-dimensional space)

From Perceptrons to (Feedforward) Neural Nets

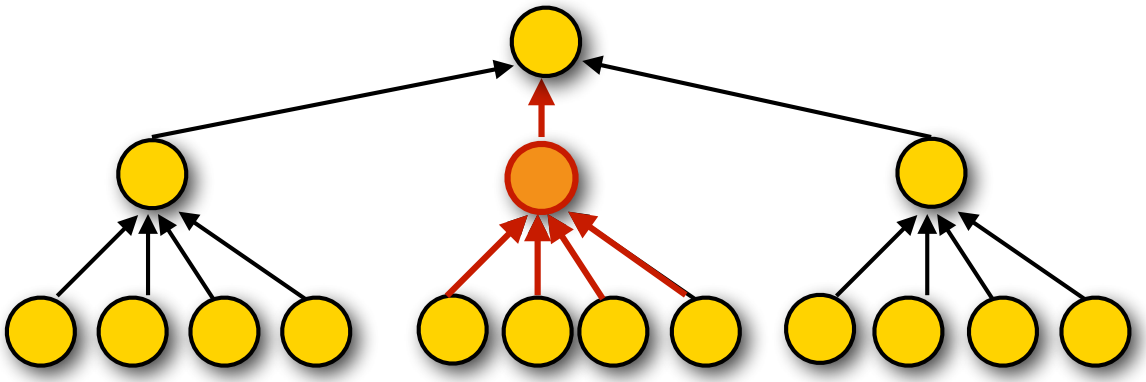
A **perceptron** can be seen as a single neuron (one output unit with a vector or **layer** of input units):

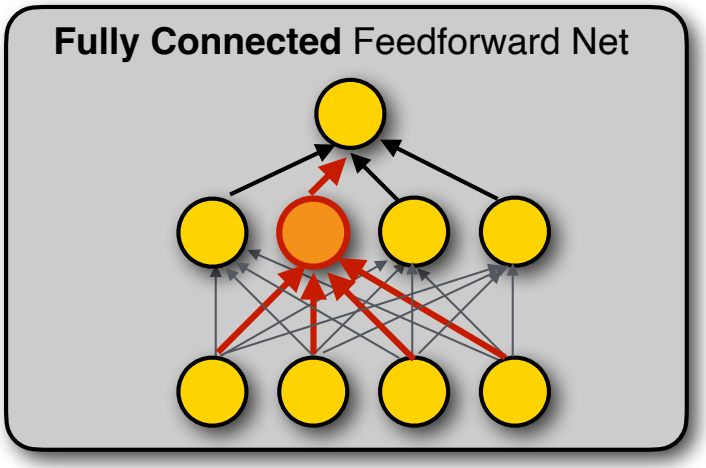


Output unit: scalar $y = f(\mathbf{x})$

Input layer: vector $\mathbf{x}$

But each element of the input can be a neuron itself:





From Perceptrons to (Feedforward) Neural Nets

Neural nets replace the Perceptron's linear threshold activation function with **non-linear activation functions** $g()$ …

$$y = g(\mathbf{w}\mathbf{x} + b)$$

… because **non-linear classifiers** are **more expressive** than linear classifiers (e.g. can represent XOR ["exclusive or"])

… because any multilayer network of linear perceptrons is **equivalent** to **a single linear perceptron**

… and because learning requires us to **set the weights** of each unit

Recall **Gradient descent** (e.g. for logistic regression):
Update the weights based on the gradient of the loss

In a multi-layer feedforward neural net, we need to pass the gradient of the loss back from the output through all layers (**backpropagation**):
We need **differentiable** activation functions

Nonlinear Activation Functions $g()$

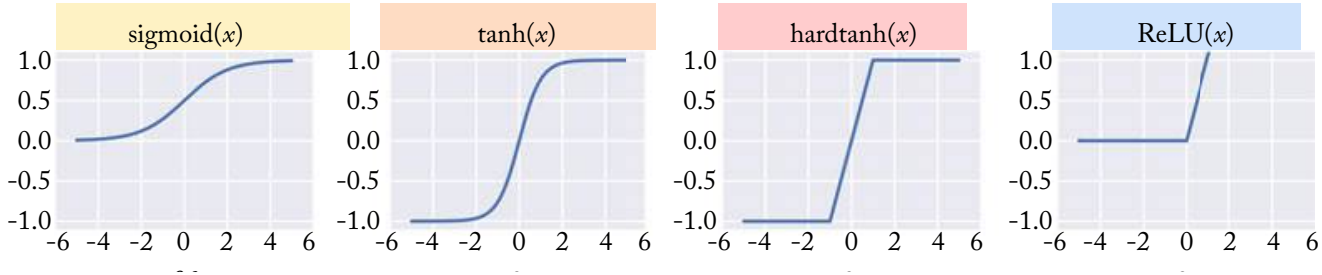


Fig.:Y. Goldberg (2017) Neural Network Methods for Natural Language Processing

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

Outputs in $[0,1]$ range. Useful for output units (**probabilities**), interpolation

Hyperbolic tangent: $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$

Outputs in $[-1,1]$ range. Useful for **internal** units

Hard tanh $\text{htanh}(x) = -1 \text{ for } x < -1,\ 1 \text{ for } x > 1,\ x \text{ otherwise}$

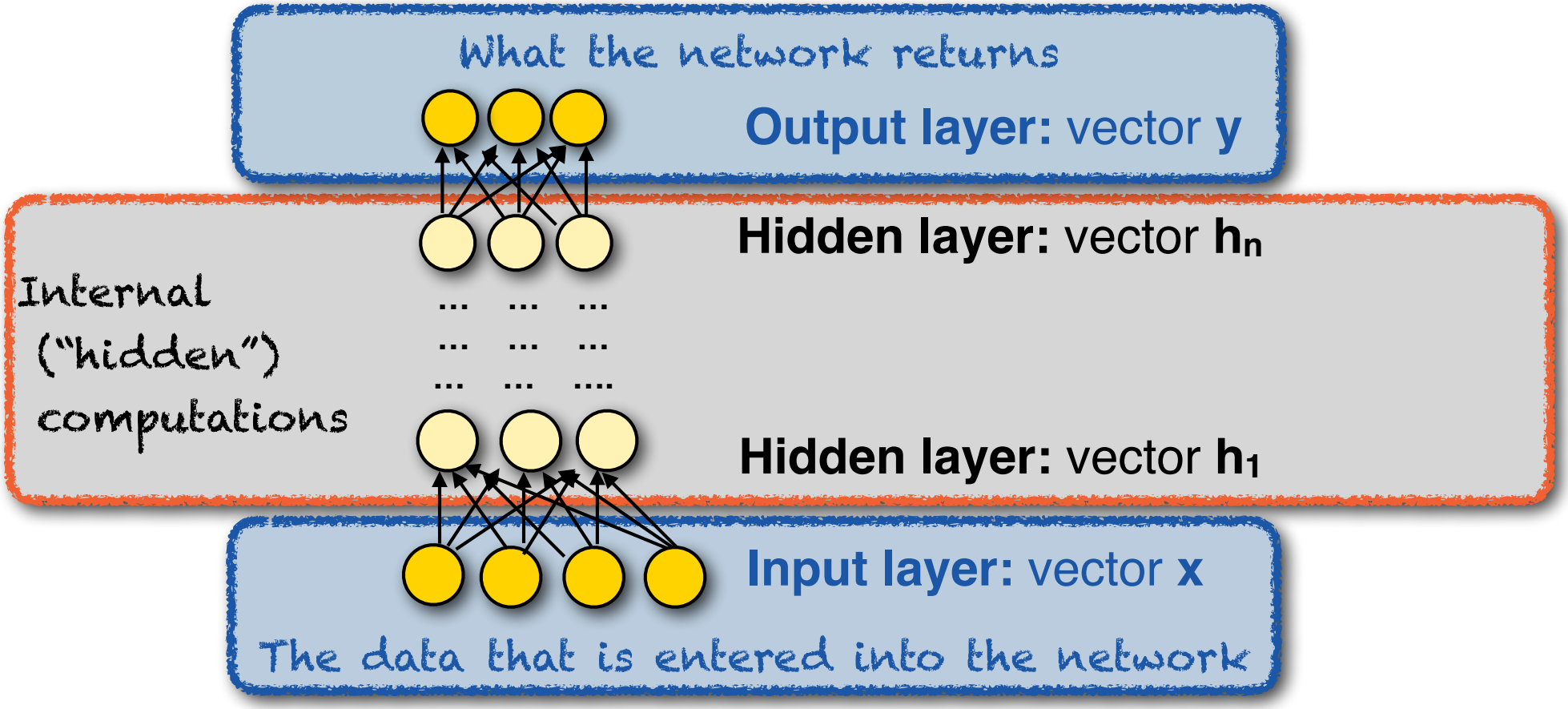
Outputs in $[-1,1]$ range. Approximates $\tanh$

Rectified Linear Unit: $ReLU(x) = \max(0,x)$

Outputs in $[0, +\infty]$. Works very well for **internal** units.

Multi-layer feedforward networks

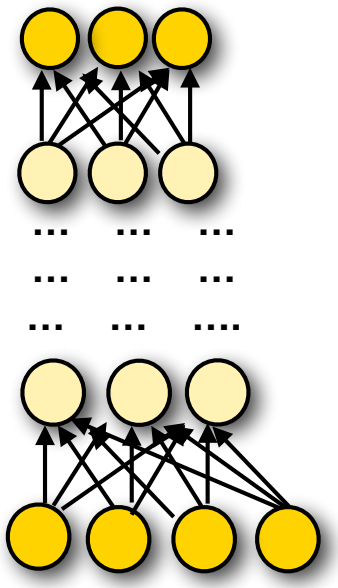
We typically assume feedforward networks are organized into layers:



Fully connected feedforward nets

Three kinds of layers,
arranged in sequence:

- **Input layer**
  (what's fed into the net)
- **Hidden layers**
  (intermediate computations)
- **Output layer**
  (what the net returns)



Output layer: vector **y**

Hidden layer: vector $\mathbf{h_n}$

Hidden layer: vector $\mathbf{h_1}$

Input layer: vector **x**

Each layer consists of a number of **units**.

- Each hidden/output unit computes a ***real-valued* activation**
- In a ***feedforward*** net, each (hidden/output) unit receives inputs from the units in the **immediately *preceding* layer**
- In a ***fully connected*** feedforward net, each unit receives inputs from ***all* units** in the immediately preceding layer

Additional "*Highway connections*" that skip layers can be useful

Feedforward computations

The **activation x_{ij} of unit *j* in layer *i*** is computed as

$$x_{ij} = g(\mathbf{w}_{ij} \cdot \mathbf{x}_{i-1} + b_{ij})$$

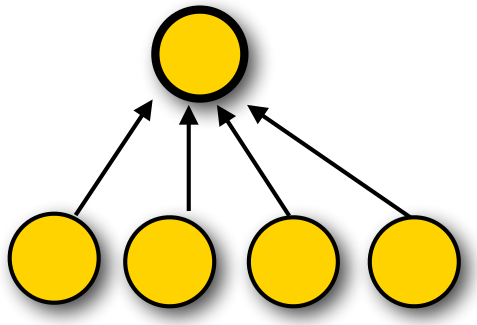
where

- $\mathbf{w}_{ij} = (w_{ij1}, \ldots, w_{ijK})$ is a (unit-specific) **weight vector**
  (K= #units in $(i-1)$-th layer, because each connection into unit j is associated with one real-valued weight for each unit in the preceding layer)
- b_{ij} is a (unit-specific) real-valued **bias term**
- $g()$ is a (layer-specific) **non-linear activation function**

Each **layer** is defined by its number of units, N,
a non-linear activation function $g()$ applied to all units in the layer,
a learned matrix of weights $\mathbf{W}$, and a learned bias vector $\mathbf{b}$.

Binary Classification
with a multilayer feedforward net

The **output layer** consists of a **single unit** with the **sigmoid** activation function

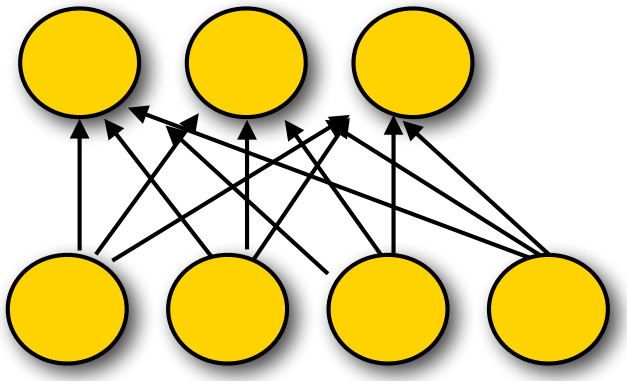


One output unit with **sigmoid** activation function

$$y = \sigma(\mathbf{w}\mathbf{x} + b) \in [0...1]$$

Multi-*Class* Classification with a multilayer feedforward net

With K output classes, the output layer has K **units** with a **softmax** activation function:



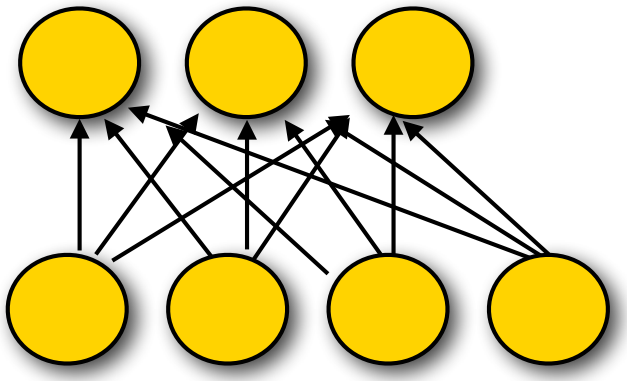
Output layer:
A vector $\mathbf{y} = (y_1, \ldots, y_K)$ where the *i*-th element corresponds to the probability that the input has class *i*:

$$y_i = \mathbf{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{k=1}^{K} \exp(z_k)}$$

such that we get a categorical distribution over all *K* classes

Multi-*Label* Classification
with a multilayer feedforward net

With K output classes, K output units
with K **sigmoid** activation functions:



Output layer:
A vector $\mathbf{y} = (y_1, \ldots, y_K)$ where the *i*-th element corresponds to the probability that the input does (or doesn't) have class *i*:

$$y_i = \mathbf{sigmoid}(\mathbf{w}_i \mathbf{x}_i + b_i)$$

We now have a separate probability for each possible class label.

Part 3: Neural n-gram models

Our first neural net for NLP: A neural n-gram model

Given a fixed-size vocabulary V, an *n*-gram model predicts the probability of the *n*-th word following the preceding *n*–1 words:

$$P(w^{(i)} \mid w^{(i-1)}, w^{(i-2)}, \ldots, w^{i-(n-1)})$$

How can we model this with a neural net?

— **Input layer:** concatenate *n*–1 word vectors

— **Output layer:** a softmax over |V| units

An n-gram model $P(w \mid w_1 \ldots w_k)$ as a feedforward net (naively)

Assumptions:

The **vocabulary** V contains V types (incl. UNK, BOS, EOS)
We want to condition each word on k preceding words

Our (naive) model:

— [Naive]
Each **input word** $w_i \in V$ is a ***V*-dimensional one-hot vector** $v(w)$
→ The **input layer** $\mathbf{x} = [v(w_1),\ldots,v(w_k)]$ has $V \times k$ elements

— We assume **one hidden layer h**

— The **output layer** is a softmax over V elements

$$P(w \mid w_1 \ldots w_k) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$$

An n-gram model $P(w \mid w_1 \ldots w_k)$ as a feedforward net (better)

Assumptions:

The **vocabulary** V contains V types (incl. UNK, BOS, EOS)
We want to condition each word on k preceding words

Our (better) model:

— [Better]
Each **input word** $w_i \in V$ is an ***n*-dimensional dense embedding vector** $v(w)$ (with $n \ll V$)
→ The **input layer** $\mathbf{x} = [v(w_1),\ldots,v(w_k)]$ has $n \times k$ elements

— We assume **one hidden layer h**

— The **output layer** is a softmax over V elements

$$P(w \mid w_1 \ldots w_k) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$$

Our neural n-gram models

Architecture:

Input Layer: $\mathbf{x} = [v(w_1)\ldots v(w_k)]$

Hidden Layer: $\mathbf{h} = g(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)$

Output Layer: $P(w \mid w_1 \ldots w_k) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$

How many parameters do we need? [# of weights and biases]:

Hidden layer with one-hot inputs: $\mathbf{W}^1 \in \mathbb{R}^{(k \cdot V) \times \dim(\mathbf{h})}$ $\mathbf{b}^1 \in \mathbb{R}^{\dim(\mathbf{h})}$

Hidden layer with dense inputs: $\mathbf{W}^1 \in \mathbb{R}^{(k \cdot n) \times \dim(\mathbf{h})}$ $\mathbf{b}^1 \in \mathbb{R}^{\dim(\mathbf{h})}$

Output layer (any inputs): $\mathbf{W}^2 \in \mathbb{R}^{\dim(\mathbf{h}) \times V}$ $\mathbf{b}^2 \in \mathbb{R}^{V}$

With V = 10K, n = 300 (word2vec), dim(h) = 300

k = 2 (trigram): $\mathbf{W}^1 \in \mathbb{R}^{20{,}000 \times 300}$ or $\mathbf{W}^1 \in \mathbb{R}^{600 \times 300}$ and $\mathbf{b}_1 \in \mathbb{R}^{300}$

k = 5 (six-gram): $\mathbf{W}^1 \in \mathbb{R}^{50{,}000 \times 300}$ or $\mathbf{W}^1 \in \mathbb{R}^{1500 \times 300}$ and $\mathbf{b}_1 \in \mathbb{R}^{300}$

$\mathbf{W}^2 \in \mathbb{R}^{300 \times 10{,}000}$ $\mathbf{b}^2 \in \mathbb{R}^{10{,}000}$

Six-gram model with one-hot inputs: 27,000,460,000 parameters,
with dense inputs: 3,460,000 parameters

Traditional six-gram model: $10^{4 \times 6} = 10^{24}$ parameters

Naive (one-hot input) neural n-gram model

Advantage over non-neural n-gram model:

— The hidden layer captures **interactions** among context words

— **Increasing the order** of the n-gram requires only a small **linear increase** in the number of parameters.

dim($\mathbf{W}^1$) goes from $(k \cdot \dim(V)) \cdot \dim(\mathbf{h})$ to $((k+1) \cdot \dim(V)) \cdot \dim(\mathbf{h})$

— **Increasing the vocabulary** also leads only to a linear increase in the number of parameters

But: With a one-hot encoding and dim(V) ≈ 10K or so, this model still requires a LOT of parameters to learn.

And: The Markov assumption still holds

Better (dense embeddings input) neural n-gram model

Advantage over non-neural n-gram model:

— Same as naive neural model, plus:

Advantages over naive neural n-gram model:

— We have far **fewer parameters** to learn

— **Better generalizations:** If similar input words have similar embeddings, the model will predict similar probabilities in similar contexts:

$P(w \mid \text{the doctor saw the}) \approx P(w \mid \text{a nurse sees her})$

But: This generalization only works if the contexts have similar words in the *same* position.

And: The Markov assumption still holds.

Neural n-gram models

Naive neural n-gram models (one-hot inputs) have similar shortcomings to standard n-gram models

- Models get very large (and sparse) as n increases
- We can't generalize across similar contexts
- Markov (independence) assumptions are too strict

Better neural n-gram models can be obtained with dense word embeddings:

- Models remain much smaller
- Embeddings may provide some (limited) generalization across similar contexts

Future lectures: CBOW neural nets as a language model, recurrent language models