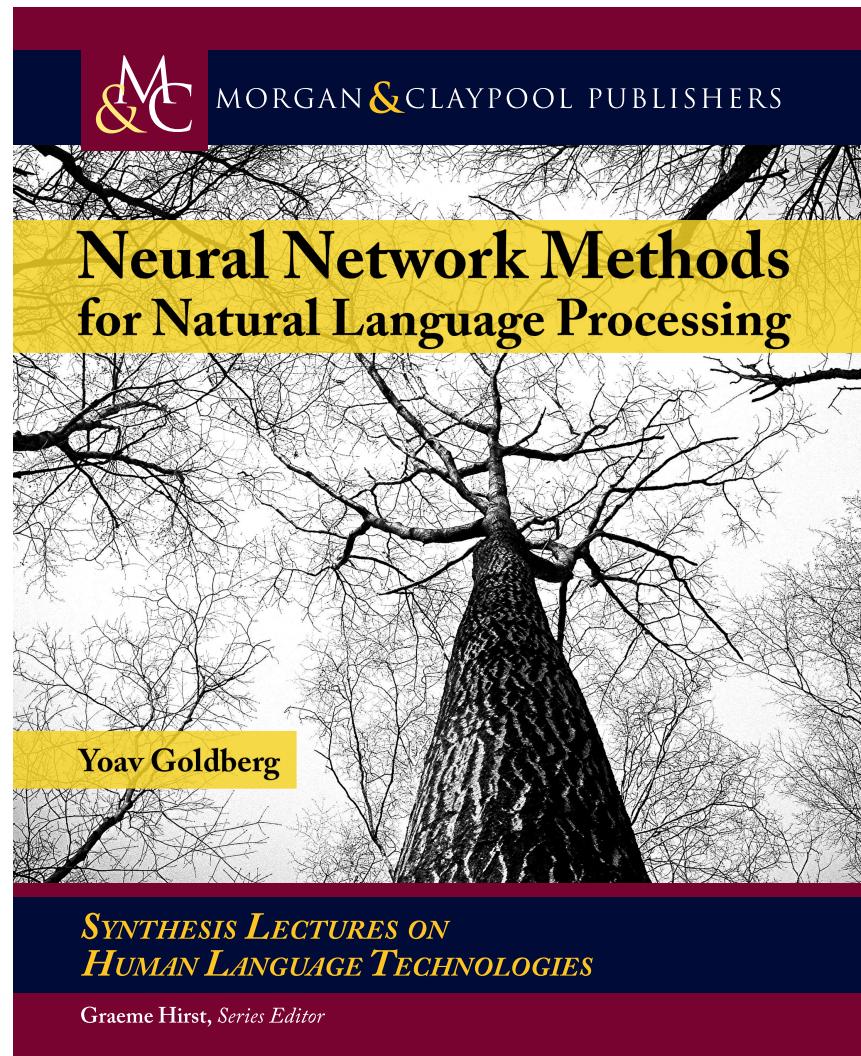


Neural Network LMs

READ CHAPTER 4 AND
CHAPTER 9 FROM

YOAV GOLDBER'S BOOK
NEURAL NETWORKS
METHODS FOR NLP

(IT'S FREE TO DOWNLOAD
FROM PENN'S CAMPUS!)



Recap: Logistic Regression

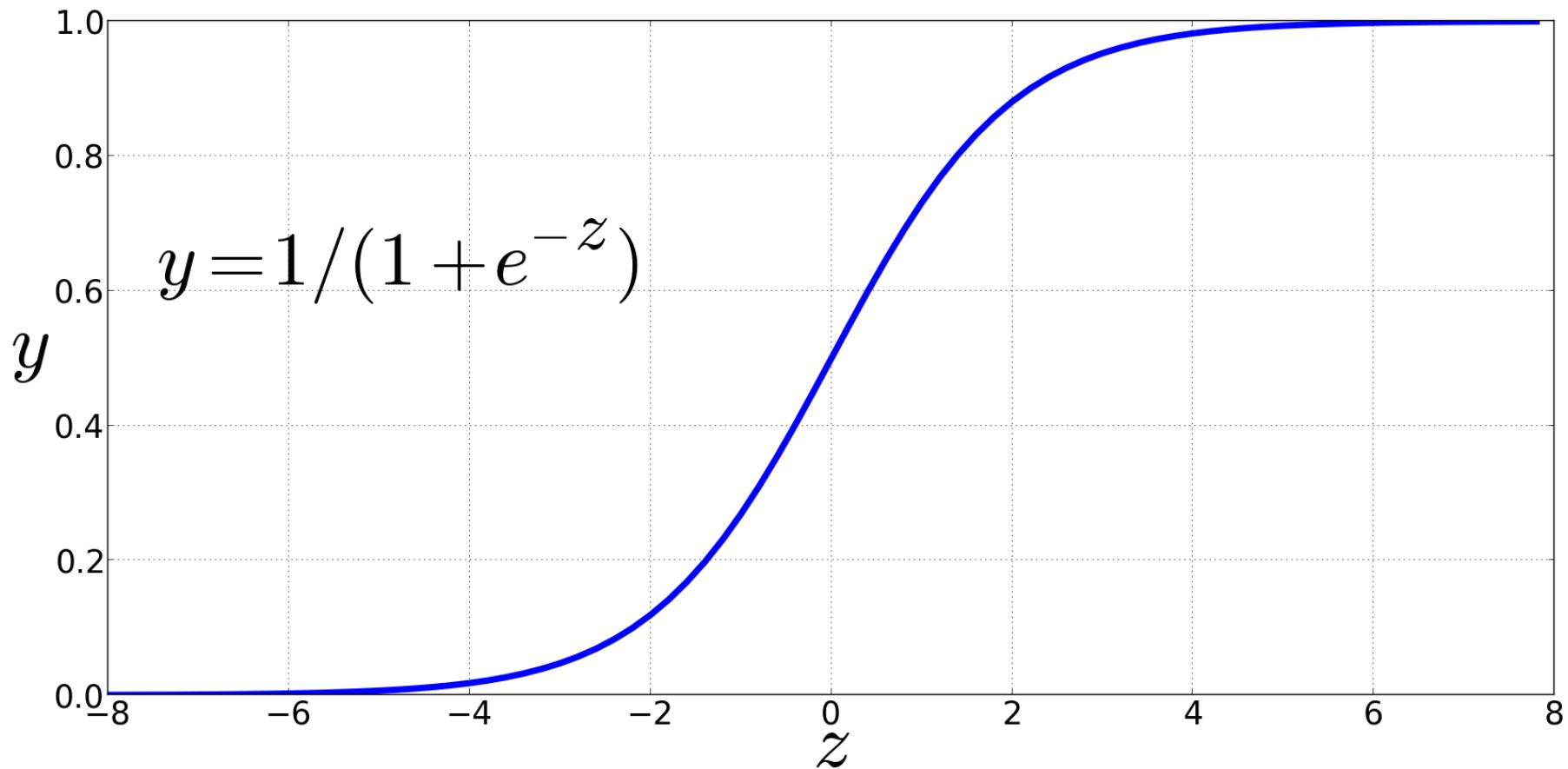
Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**.

$$z = \left(\sum_{i=1}^n w_i x_i \right) + b$$

We can also write this as a dot product:

$$z = w \cdot x + b$$

Recap: Sigmoid function



Recap: Probabilities

$$\begin{aligned} P(y = 1) &= \sigma(w \cdot x + b) \\ &= \frac{1}{1 + e^{-(w \cdot x + b)}} \end{aligned}$$

$$P(y = 0) = 1 - \sigma(w \cdot x + b)$$

Recap: Loss functions

We need to determine for some observation x how close the classifier output ($\hat{y} = \sigma(w \cdot x + b)$) is to the correct output (y , which is 0 or 1).

$$L(\hat{y}, y) = \text{how much } \hat{y} \text{ differs from the true } y$$

Recap: Loss functions

For one observation x , let's **maximize** the probability of the correct label $p(y|x)$.

$$p(y|x) = \hat{y}^y(1 - \hat{y})^{1-y}$$

If $y = 1$, then $p(y|x) = \hat{y}$.

If $y = 0$, then $p(y|x) = 1 - \hat{y}$.

Recap: Cross-entropy loss

The result is cross-entropy loss:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

Finally, plug in the definition for $\hat{y} = \sigma(w \cdot x + b)$

$$L_{CE}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log(1 - \sigma(w \cdot x + b))]$$

Recap: Cross-entropy loss

Why does minimizing this negative log probability do what we want?

A perfect classifier would assign probability 1 to the correct outcome ($y=1$ or $y=0$) and probability 0 to the incorrect outcome.

That means the higher \hat{y} (the closer it is to 1), the better the classifier; the lower \hat{y} is (the closer it is to 0), the worse the classifier.

The negative log of this probability is a convenient loss metric since it goes from 0 (negative log of 1, no loss) to infinity (negative log of 0, infinite loss).

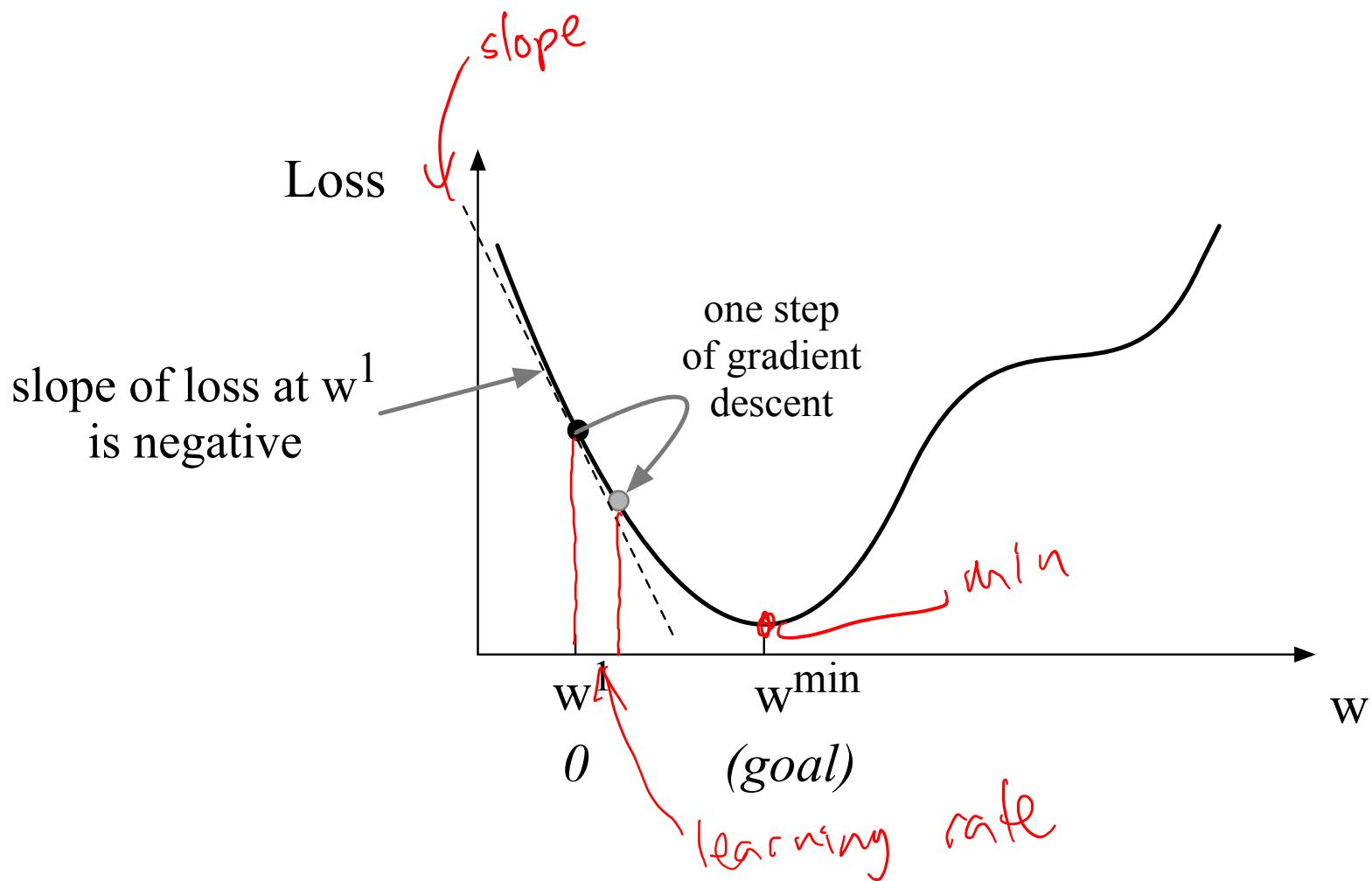
Recap: Finding good parameters

We use gradient descent to find good settings for our weights and bias by minimizing the loss function.

$$\hat{\theta} = \operatorname{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^m L_{CE}(y^{(i)}, x^{(i)}; \theta)$$

Gradient descent is a method that finds a minimum of a function by figuring out in which direction (in the space of the parameters θ) the function's slope is rising the most steeply, and moving in the opposite direction.

Iteratively find minimum



Stochastic gradient descent algorithm

•

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x , y) **returns** θ

where: L is the loss function

f is a function parameterized by θ

x is the set of training inputs $x^{(1)}$, $x^{(2)}$, ..., $x^{(n)}$

y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$, ..., $y^{(n)}$

$\theta \leftarrow 0$

repeat T times

For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)

Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output \hat{y} ?

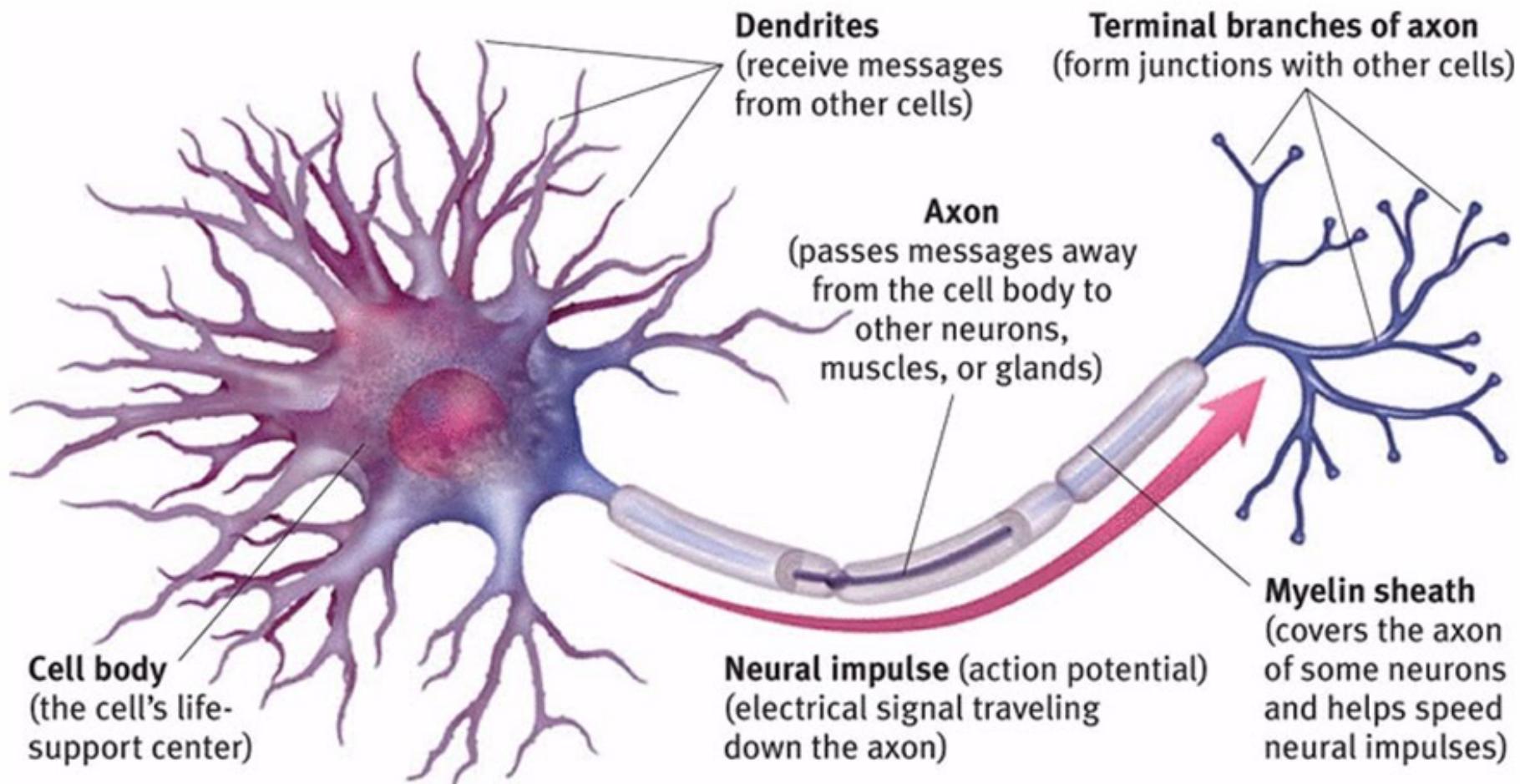
Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?

$g \leftarrow \nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss ?

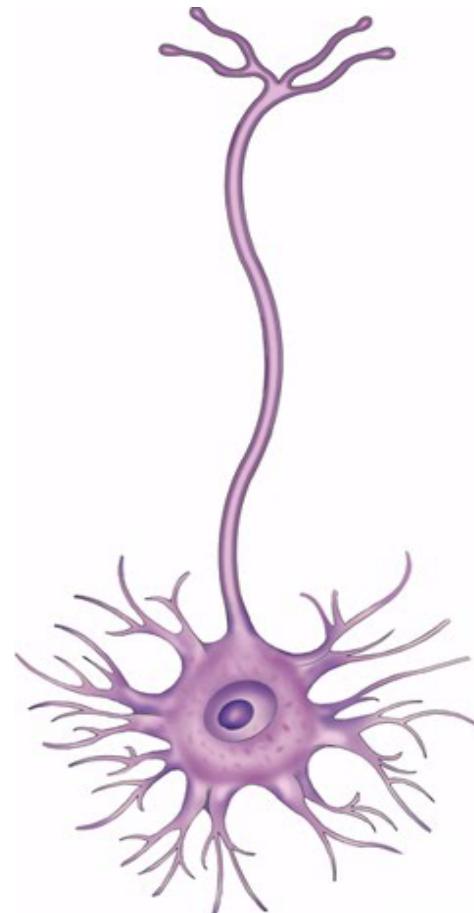
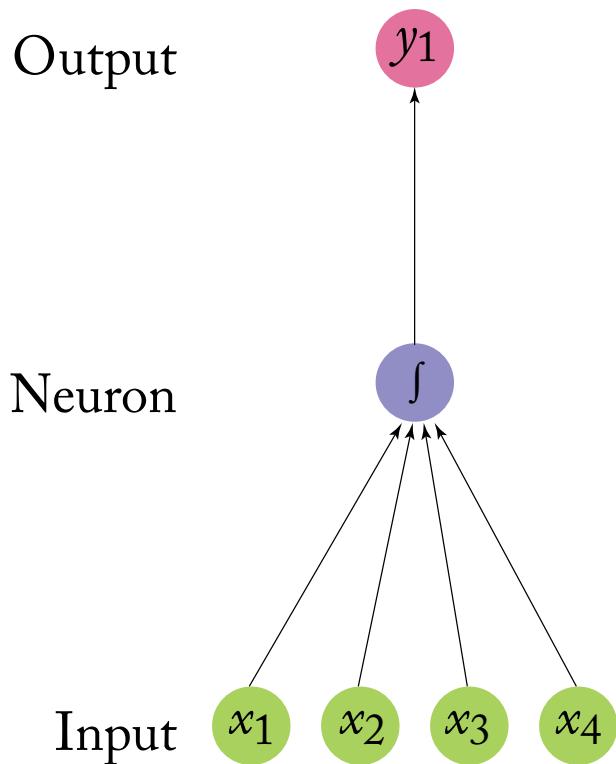
$\theta \leftarrow \theta - \eta g$ # go the other way instead

return θ

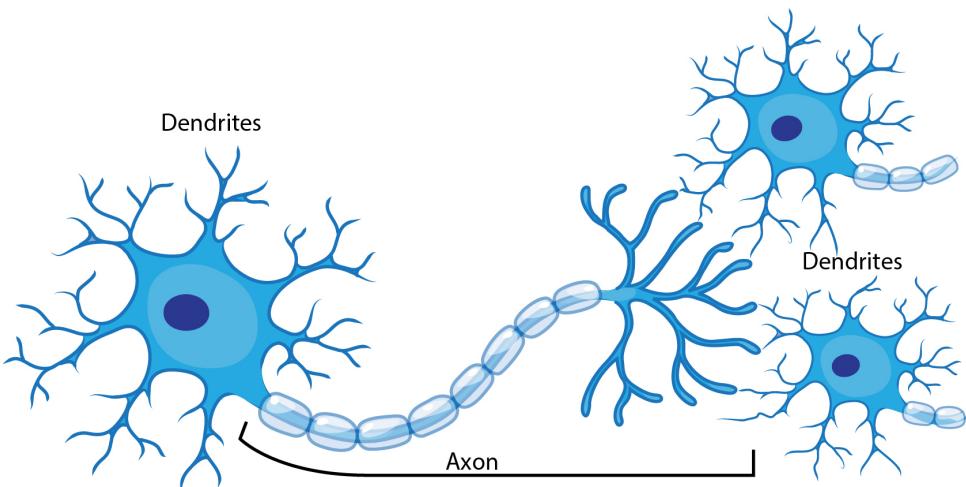
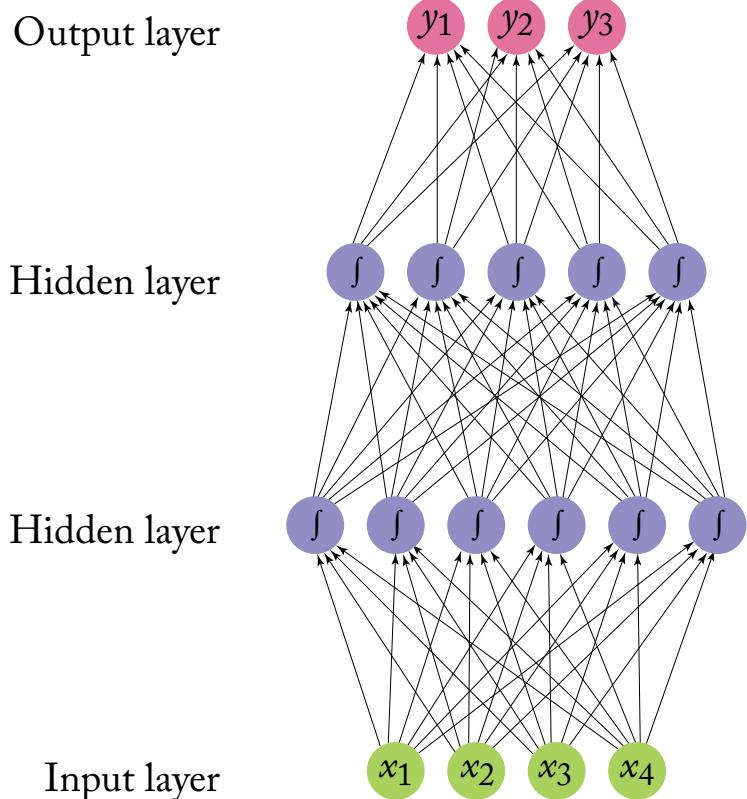
Neural Networks: A brain-inspired metaphor



A single neuron



Neural networks



Mathematical Notation

The simplest neural network is called a perceptron. It is simply a linear model:

$$\text{NN}_{\text{Perceptron}}(\mathbf{x}) = \mathbf{x} \mathbf{W} + \mathbf{b}$$

$$\mathbf{x} \in \mathbb{R}^{d_{in}}, \quad \mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}}, \quad \mathbf{b} \in \mathbb{R}^{d_{out}}$$

where W is the weight matrix and b is a bias term.

Mathematical Notation

To go beyond linear function, we introduce a non-linear hidden layer. The result is called a Multi-Layer Perceptron with one hidden layer.

$$\text{NN}_{\text{MLP1}}(\mathbf{x}) = g(\mathbf{x} \mathbf{W^1} + \mathbf{b^1}) \mathbf{W^2} + \mathbf{b^2}$$
$$\mathbf{x} \in \mathbb{R}^{d_{in}}, \quad \mathbf{W^1} \in \mathbb{R}^{d_{in} \times d_1}, \quad \mathbf{b^1} \in \mathbb{R}^{d_1}, \quad \mathbf{W^2} \in \mathbb{R}^{d_1 \times d_2}, \quad \mathbf{b^2} \in \mathbb{R}^{d_2}$$

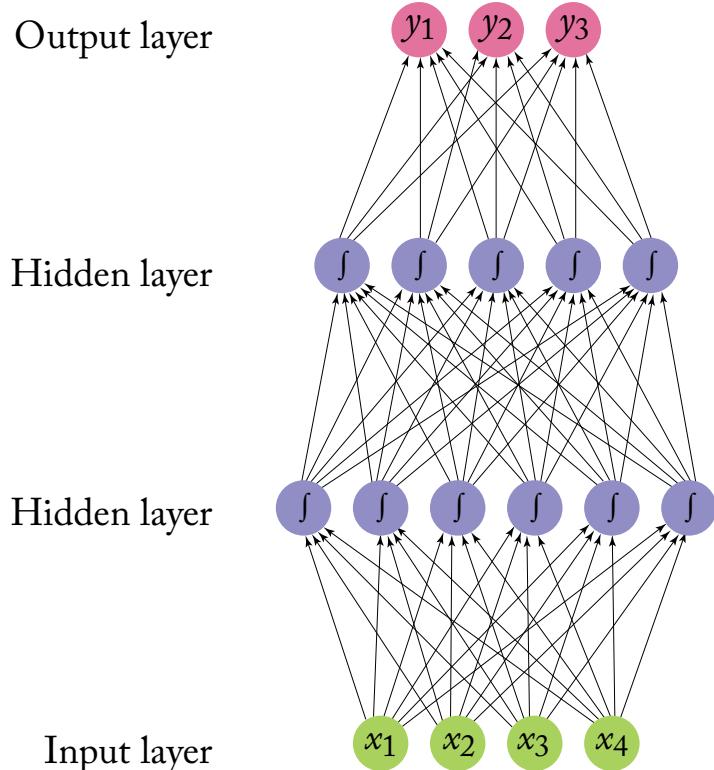
Here $\mathbf{W^1}$ and $\mathbf{b^1}$ are a matrix and a bias for the **first** linear transformation of the input \mathbf{x} ,

\mathbf{g} is a nonlinear function (also an activation function),
 $\mathbf{W^2}$ and $\mathbf{b^2}$ are the matrix and bias term for a **second** linear transform.

Mathematical Notation

We can add additional linear transformations and nonlinearities, resulting with a MLP with two hidden layers:

$$\text{NN}_{\text{MLP2}}(\mathbf{x}) = (g^2(g^1(\mathbf{x} \mathbf{W}^1 + \mathbf{b}^1) \mathbf{W}^2 + \mathbf{b}^2)) \mathbf{W}^3.$$



Same equation, but written with intermediary variables:

$$\text{NN}_{\text{MLP2}}(\mathbf{x}) = \mathbf{y}$$

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

$$\mathbf{h}^2 = g^2(\mathbf{h}^1 \mathbf{W}^2 + \mathbf{b}^2)$$

$$\mathbf{y} = \mathbf{h}^2 \mathbf{W}^3.$$

Dimensions of the layers

A neural network can be described by the dimensions of its layers and of its input.

d_{in} is the number of dimensions of the input vector

d_{out} is the number of dimensions of the output vector

A fully connected layer $I(x) = xW + b$ with input size d_{in} and output size d_{out} will have the following dimensions:

the dimensions of x are $1 \times d_{in}$

the dimensions of W are $d_{in} \times d_{out}$

the dimensions of b are $1 \times d_{out}$

Dimensions of the output layer

$d_{out} = 1$ means the neural networks output is a scalar. Such networks can be used for

- Regression or scoring
- Binary classification

$d_{out} = k > 1$ can be used for k-class classification.

- Associate each dimension with a class, and look for the dimension with maximal value.
- If the output vector entries are positive and sum to one, the output can be interpreted as a distribution over class assignments.

The **softmax** forces the values in an output layer to be positive and sum to 1, making them interpretable as a probability distribution.

$$\hat{y} = \text{softmax}(xW + b)$$

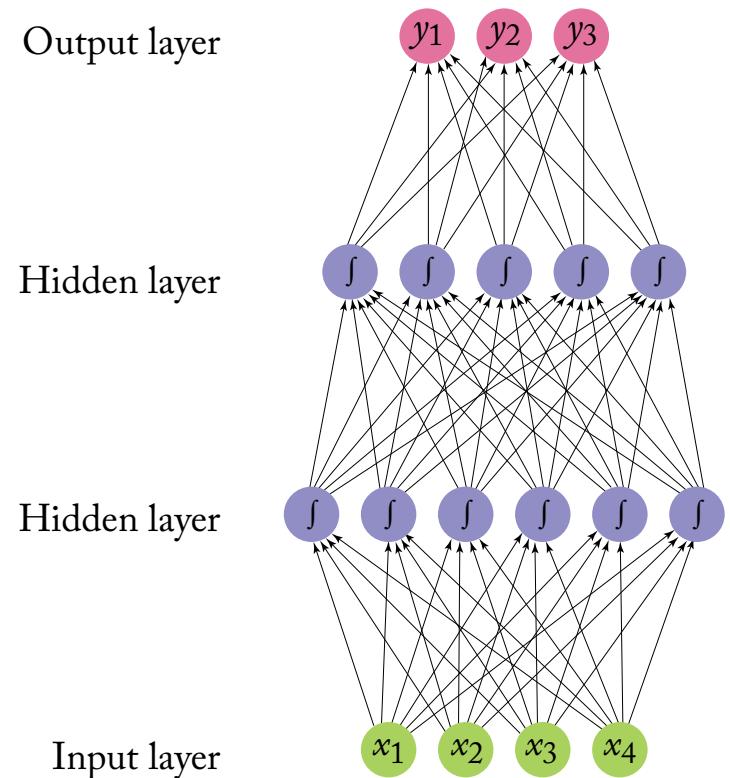
$$\hat{y}_{[i]} = \frac{e^{(xW + b)_{[i]}}}{\sum_j e^{(xW + b)_{[j]}}}.$$

Representation Power

A Multi-Layer Perceptron with one hidden layer is a “universal approximator”.

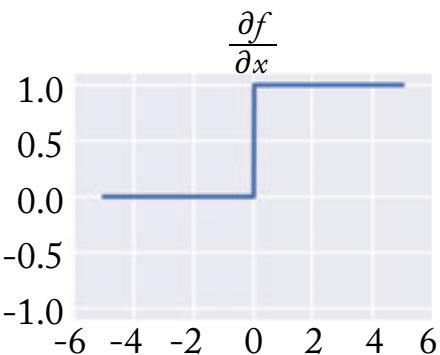
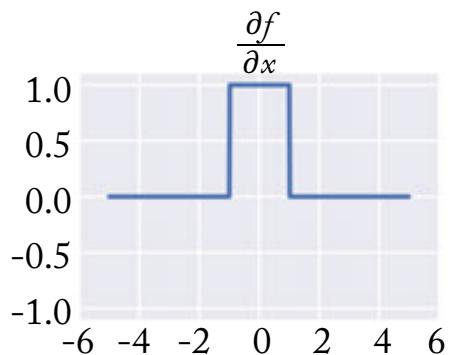
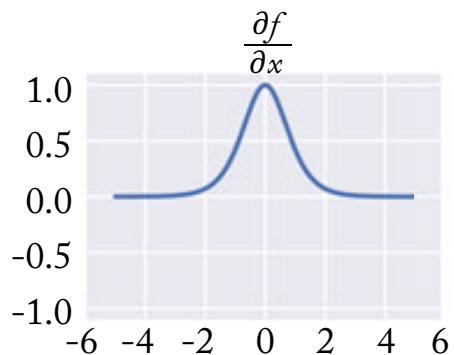
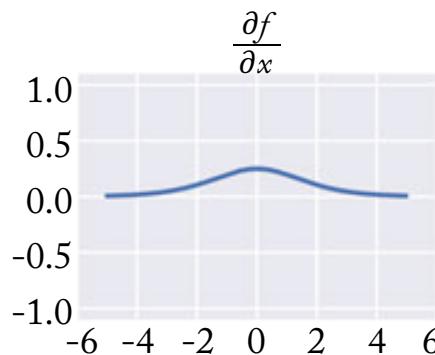
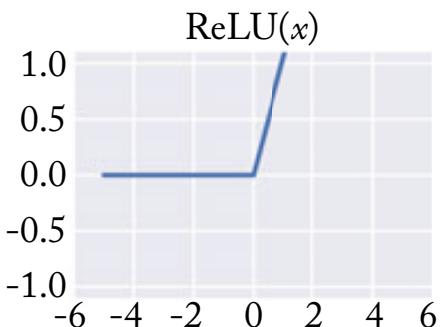
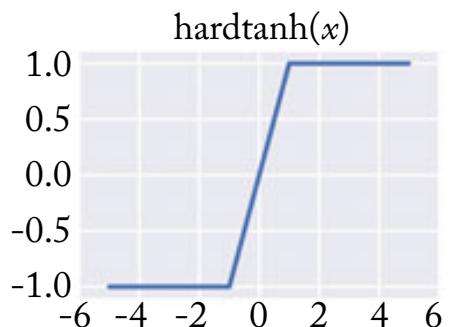
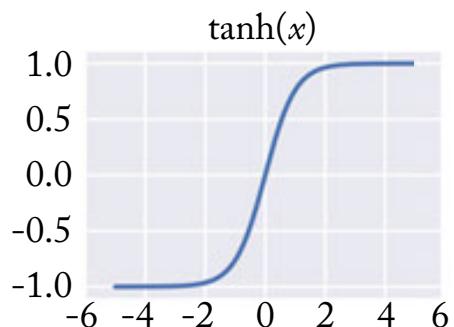
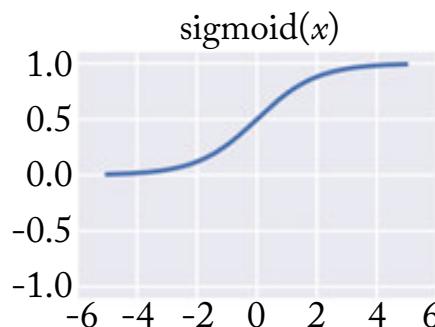
It can approximate a family of functions that includes all continuous functions on a closed and bounded subset of \mathbb{R}^n

It can approximate any function mapping from any finite dimensional discrete space to another.



So why use multiple layers?

Common Nonlinearities



Training concerns

Loss functions. Much like training a logistic regression classifier, we define a loss function

$$L(\hat{y}, y) = \text{how much } \hat{y} \text{ differs from the true } y$$

Loss functions like *cross-entropy loss* are relevant for neural nets too.

Regularization. To avoid overfitting, we often add a regularization term alongside our loss function when we search for the best parameters.

$$\begin{aligned}\hat{\Theta} &= \operatorname*{argmin}_{\Theta} \mathcal{L}(\Theta) + \lambda R(\Theta) \\ &= \operatorname*{argmin}_{\Theta} \frac{1}{n} \sum_{i=1}^n L(f(\mathbf{x}_i; \Theta), \mathbf{y}_i) + \lambda R(\Theta)\end{aligned}$$

Dropout attempts to avoid overfitting by randomly dropping (m 3/13 to 0) half of the neurons in the network in each training example in SGD.

Language Models

Estimate the probability of a sentence consisting of word sequence $w_{1:n}$

$$P(w_{1:n}) \approx \prod_{i=1}^n P(w_i \mid w_{i-k:i-1})$$

We need to estimate the probability of $P(w_{i+1} \mid w_{k-i:i})$ from a large corpus.

$$\hat{p}_{\text{MLE}}(w_{i+1} = m \mid w_{i-k:i}) = \frac{\#(w_{i-k:i+1})}{\#(w_{i-k:i})}$$

$$\hat{p}_{\text{add-}\alpha}(w_{i+1} = m \mid w_{i-k:i}) = \frac{\#(w_{i-k:i+1}) + \alpha}{\#(w_{i-k:i}) + \alpha |V|}$$

$$\hat{p}_{\text{int}}(w_{i+1} = m \mid w_{i-k:i}) = \lambda_{w_{i-k:i}} \frac{\#(w_{i-k:i+1})}{\#(w_{i-k:i})} + (1 - \lambda_{w_{i-k:i}}) \hat{p}_{\text{int}}(w_{i+1} = m \mid w_{i-(k-1):i}).$$

Limitations of LMs

The “curse of dimensionality”. If we want to model the full joint distribution of 10 consecutive words with a vocabulary V of size 100,000, there are potentially $100,000^{10} - 1 = 10^{50} - 1$ free parameters.

In n-gram LMs, we simplify this to predict the next word given a limited context. We construct conditional probabilities table for n given n-1.

Only those combinations of successive words that actually occur in our training corpus are recorded in the table.

Having observed *black car* and *blue car* does not influence our estimates of *red car*.

A lot of what we do is language modelling (smoothing, backoff, etc) is trying to deal with the unobserved entries.

Neural LMs (Bengio et al 2003)

1. Associate each word in the vocabulary with a vector-representation, thereby creating a notion of similarity between words.
2. Express the joint probability *function* of a word sequence in terms of the word vectors for the words in that sequence.
3. Simultaneously learn the word vectors and the parameters of the *function*.

The word vectors are low-dimensional ($d=30$ to $d=100$) dense vectors, like we've seen before.

The probability function is expressed the product of conditional probabilities of the next word given the previous word, using a multi-layer neural network.

Neural LMs

The input to the neural network is a k-gram of words $w_{1:k}$.

The output is a probability distribution over the next word.

The k context words are treated as a word window. Each word is associated with an embedding vector:

$$v(w) \in \mathbb{R}^{d_w}$$

The input vector \mathbf{x} just concatenates $v(w)$ for each of the k words:

$$\mathbf{x} = [v(w_1); v(w_2); \dots; v(w_k)]$$

Neural LMs

The input \mathbf{x} is fed into a neural network with 1 or more hidden layers:

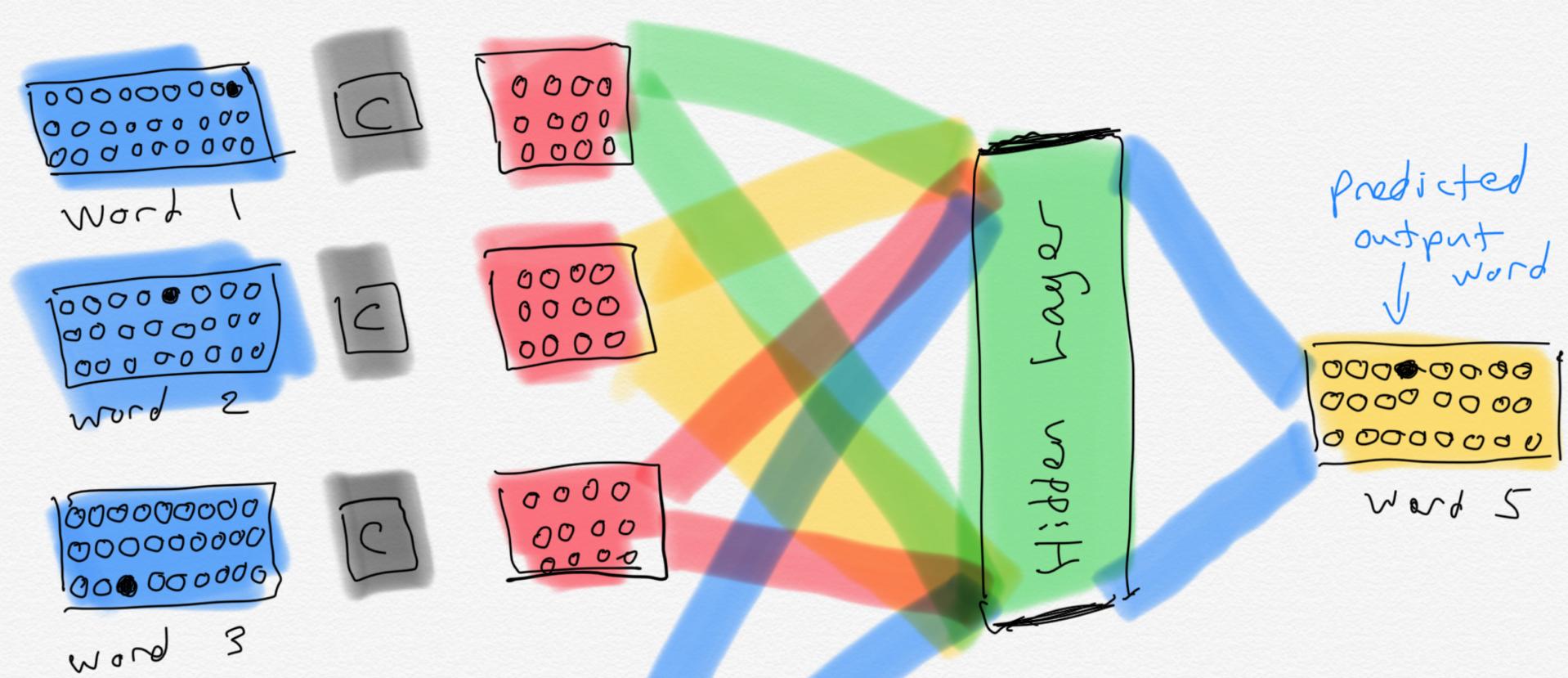
$$\hat{y} = P(w_i | w_{1:k}) = LM(w_{1:k}) = \text{softmax}(\mathbf{h} \mathbf{W^2} + \mathbf{b^2})$$

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

$$\mathbf{x} = [v(w_1); v(w_2); \dots; v(w_k)]$$

$$v(w) = \mathbf{E}_{[w]}$$

$$w_i \in V \quad \mathbf{E} \in \mathbb{R}^{|V| \times d_w} \quad \mathbf{W^1} \in \mathbb{R}^{k \cdot d_w \times d_{\text{hid}}} \quad \mathbf{b^1} \in \mathbb{R}^{d_{\text{hid}}} \quad \mathbf{W^2} \in \mathbb{R}^{d_{\text{hid}} \times |V|} \quad \mathbf{b^2} \in \mathbb{R}^{|V|}$$



Input:
"1 hot vectors"
representing the identity of the input word

\mathbb{R}^m vector w/ $M \approx 50$ dimensions

Mapping a low dimensional dense vector

function from vocab item onto

Training

The training examples are simply word kgrams from the corpus

The identities of the first $k+1$ words are used as features, and the last word is used as the target label for the classification.

Conceptually, the model is trained using cross-entropy loss.

Working with cross entropy loss works very well, but requires the use of a costly softmax operation which can be prohibitive for very large vocabularies, we often use alternative loss functions or approximations.

Advantages of NN LMs

Better results. They achieve better perplexity scores than SOTA n-gram LMs.

Larger N. NN LMs can scale to much larger orders of n. This is achievable because parameters are associated only with individual words, and not with n-grams.

They generalize across contexts. For example, by observing that the words *blue*, *green*, *red*, *black*, etc. appear in similar contexts, the model will be able to assign a reasonable score to the *green car* even though it never observed it in training, because it did observe *blue car* and *red car*.

A by-product of training are word embeddings!

$$\hat{\boldsymbol{y}} = \boldsymbol{P}(\boldsymbol{w}_i | \boldsymbol{w}_{1:k}) = LM(\boldsymbol{w}_{1:k}) = \text{softmax}(\boldsymbol{h}\boldsymbol{W^2} + \boldsymbol{b^2})$$

$$\hat{\boldsymbol{y}} = \boldsymbol{P}(\boldsymbol{w}_i | \boldsymbol{w}_{1:k}) = LM(\boldsymbol{w}_{1:k}) = \text{softmax}(\boldsymbol{h}\boldsymbol{W^2} + \boldsymbol{b^2})$$

$$~$$

$$\hat{\boldsymbol{y}} = \boldsymbol{P}(\boldsymbol{w}_i | \boldsymbol{w}_{1:k}) = LM(\boldsymbol{w}_{1:k}) = \text{softmax}(\boldsymbol{h}\boldsymbol{W^2} + \boldsymbol{b^2})$$

$$\boldsymbol{h}=g(\boldsymbol{x}\boldsymbol{W^1}+\boldsymbol{b^1})$$

$$\boldsymbol{x}=[v(w_1); v(w_2); \ldots; v(w_k)]$$

$$v(w) = E_{[w]}$$

$$w_i\in V \quad \boldsymbol{E} \in \mathbb{R}^{|V|\times d_w} \quad \boldsymbol{W^1} \in \mathbb{R}^{k\cdot d_w \times d_\text{hid}} \quad \boldsymbol{b^1} \in \mathbb{R}^{d_\text{hid}} \quad \boldsymbol{W^2} \in \mathbb{R}^{d_\text{hid} \times |V|} \quad \boldsymbol{b^2} \in \mathbb{R}^{|V|}$$