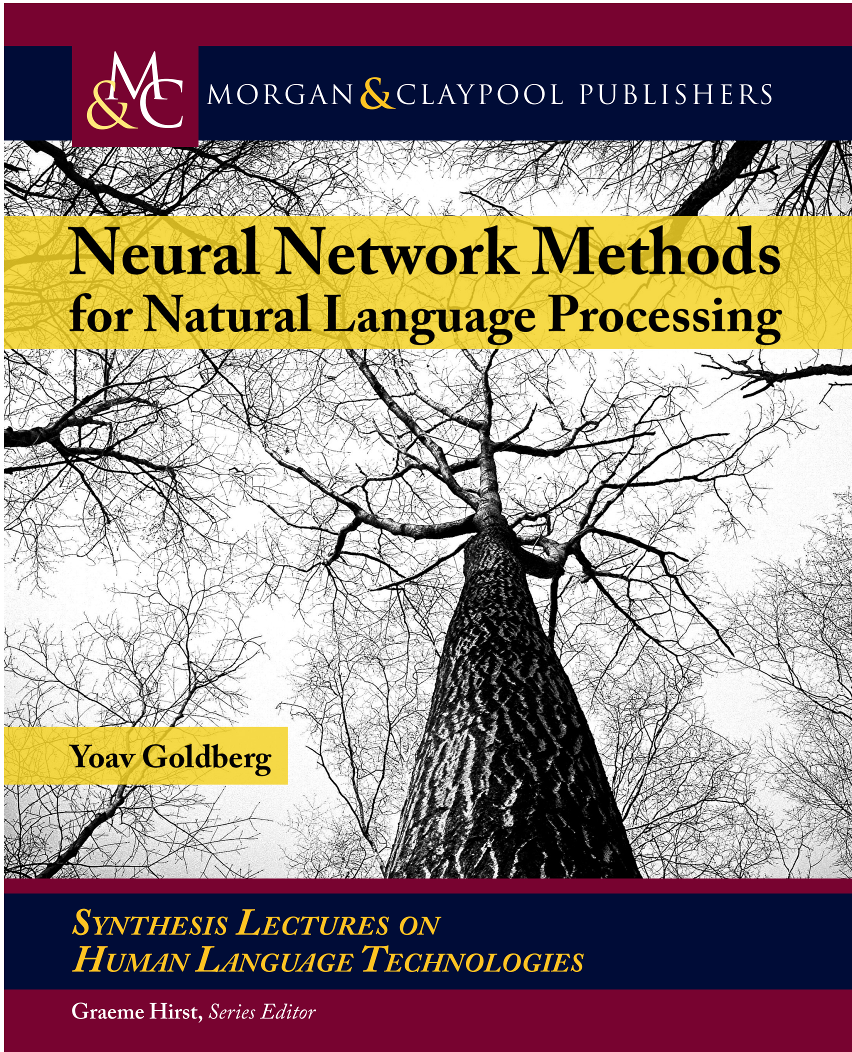
Neural Network LMs

READ CHAPTERS 5 AND 7 IN JURAFSKY AND MARTIN

READ CHAPTER 4 FROM YOAV GOLDBER'S BOOK **NEURAL NETWORKS METHODS FOR NLP**

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



Reminders

QUIZ IS DUE TONIGHT BY 11:59PM

HOMEWORK 5 IS DUE WEDNESDAY

Recap: Logistic Regression

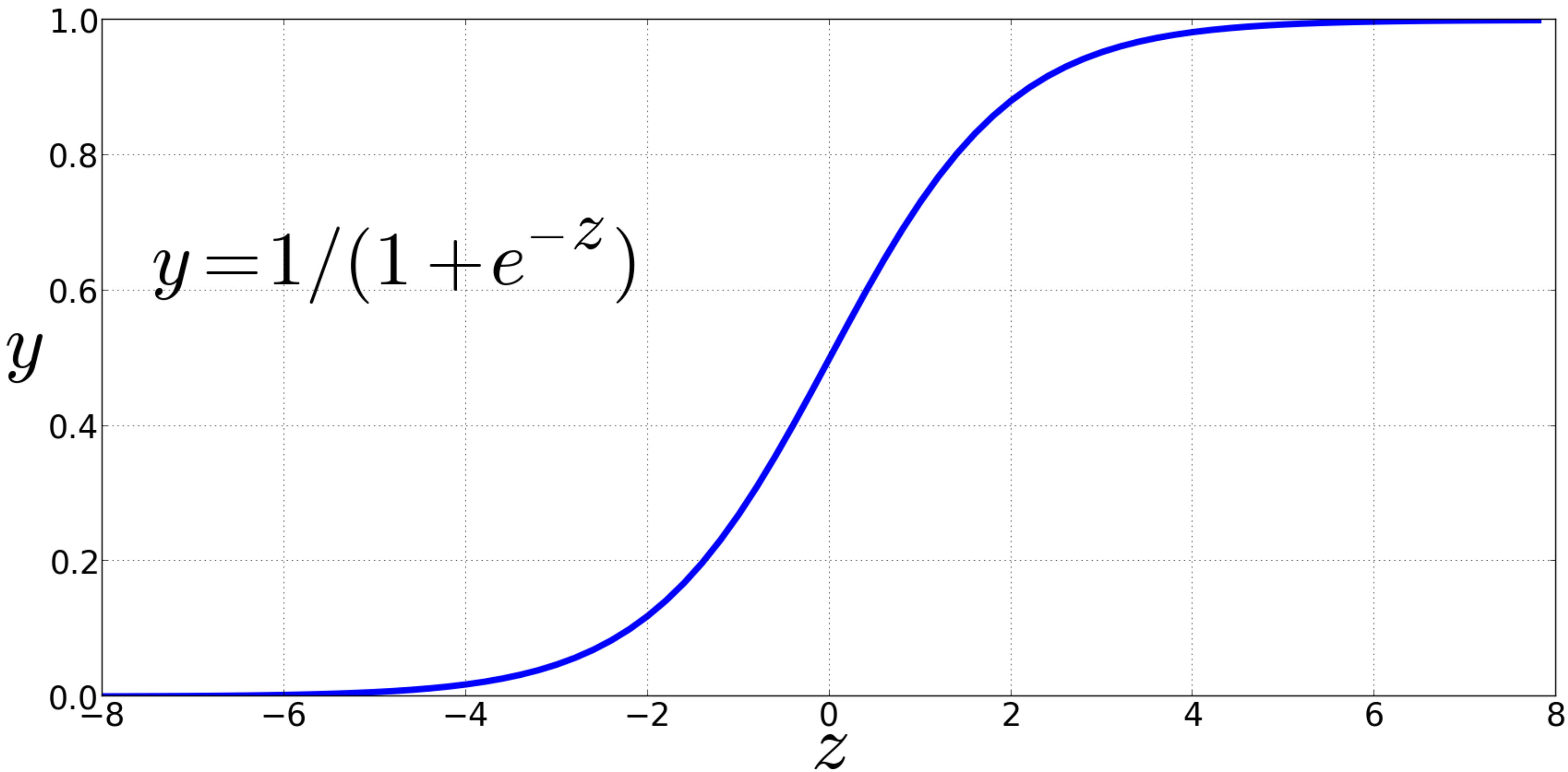
Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**.

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

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 $\mathrm{p}(\mathrm{y}|\mathrm{x}) = \hat{y}$.

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

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 $\hat{\boldsymbol{y}}$= σ ($w \cdot x$) + 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 $\hat{\boldsymbol{y}}$ (the closer it is to 1), the better the classifier; the lower $\hat{\boldsymbol{y}}$ 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).

Loss on all training examples

$$\log p(training\ labels) = \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)})$$

$$= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)})$$

$$= -\sum_{i=1}^{m} \mathrm{L}_{\mathrm{CE}}(\hat{y}^{(i)}|y^{(i)})$$

Finding good parameters

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

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

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.

Finding good parameters

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

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

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.

Gradient descent

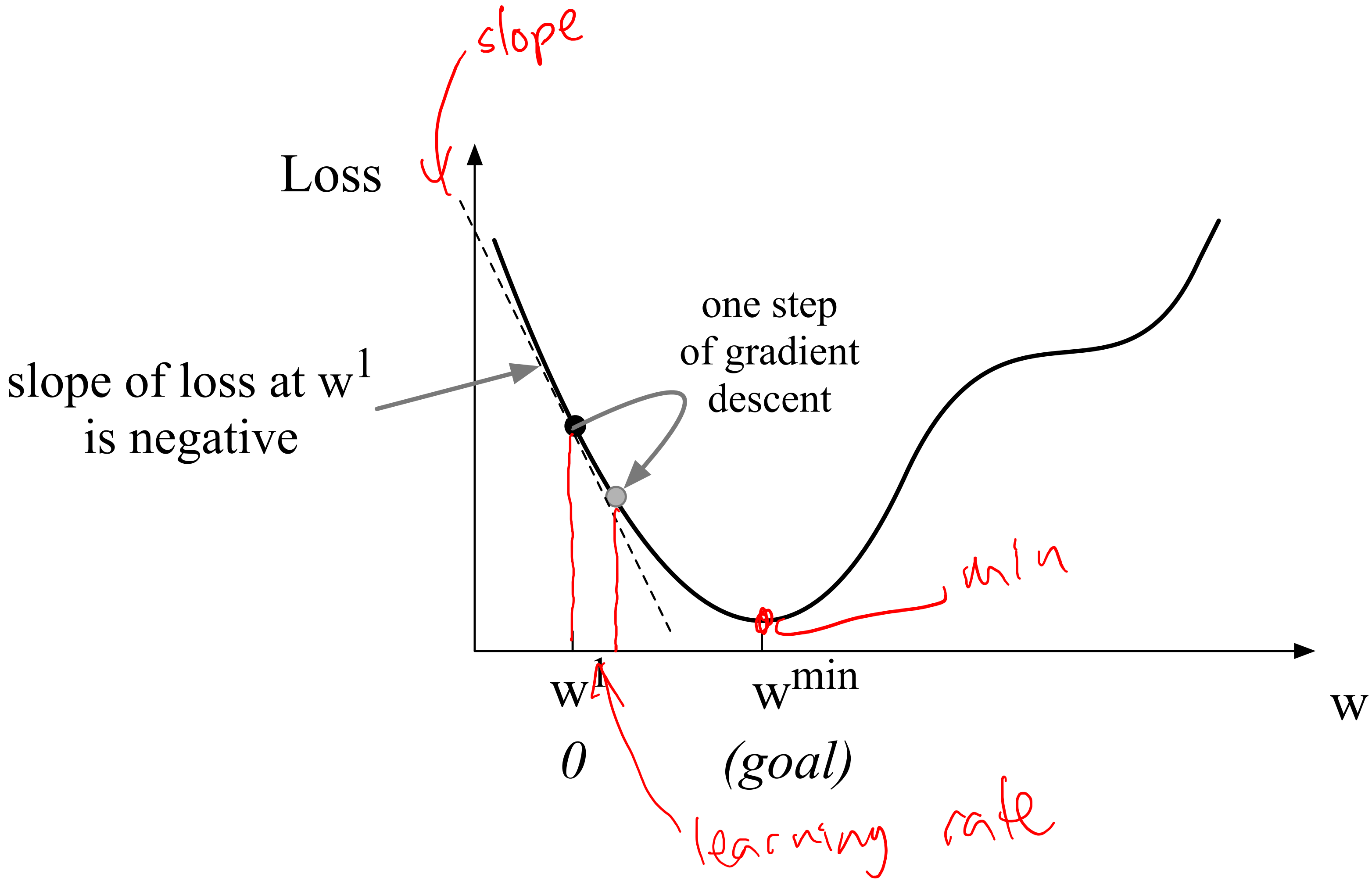
Global v. Local Minimums

For logistic regression, this loss function is conveniently **convex**.

A convex function has just **one minimum**, so there are no local minima to get stuck in.

So gradient descent starting from any point is guaranteed to find the minimum.

Iteratively find minimum



How much should we update the parameter by?

The magnitude of the amount to move in gradient descent is the value of the slope weighted by a learning rate η.

A higher/faster learning rate means that we should move *w* more on each step.

time t+1

"eta"

slope

$$w^{t+1} = w^t - \eta \frac{d}{dw} f(x;w)$$

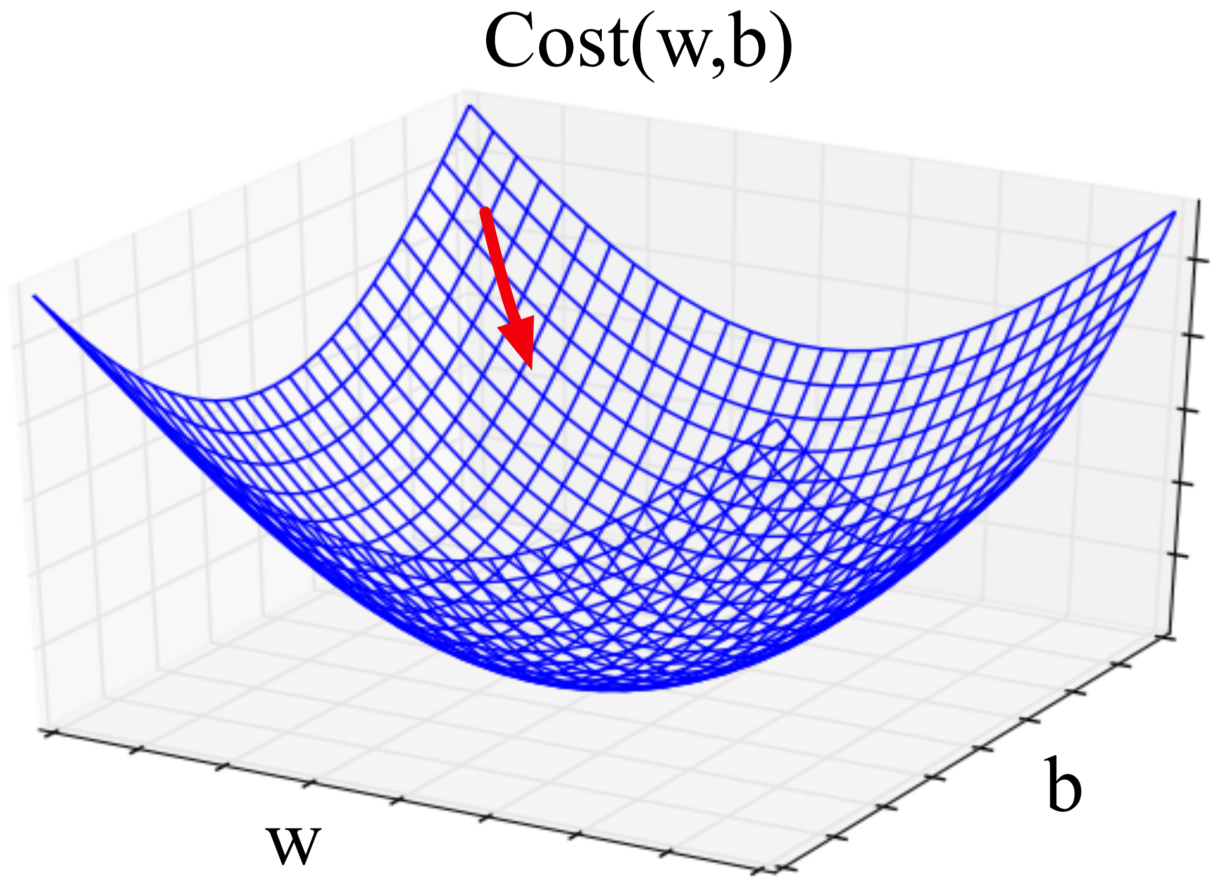
new weight

old weight

minus

learning rate * derivative of of fn.

Many dimensions



Updating each dimension w_i

$$\nabla_\theta L(\overbrace{f(x;\theta)}^{\hat{y}}, y)) \;=\; \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta), y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta), y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta), y) \end{bmatrix}$$

our model's prediction for input x given parameters θ

Learning rate

The final equation for updating θ based on the gradient is

$$\theta_{t+1} \;=\; \theta_t - \eta \nabla L(f(x;\theta), y)$$

The Gradient

To update θ, we need a definition for the gradient $\nabla L(f(x;\theta), y)$.

For logistic regression the cross-entropy loss function is:

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

The derivative of this function for one observation vector *x* for a single weight w_j is

$$\frac{\partial L_{CE}(w,b)}{\partial w_j} = [\underbrace{\sigma(w\cdot x+b)}_{\text{our model's prediction}} - \overbrace{y}^{\text{truth}}]x_j$$

⇐ * value of feature j

The gradient is a very intuitive value: the difference between the true *y* and our estimate for x, multiplied by the corresponding input value x_j .

Average Loss

$$Cost(w,b) = \frac{1}{m}\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)})$$

$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log\sigma\left(w\cdot x^{(i)} + b\right) + \left(1 - y^{(i)}\right)\log(1 - \sigma\left(w\cdot x^{(i)} + b\right))$$

This is what we want to minimize!!

The Gradient

The loss for a batch of data or an entire dataset is just the average loss over the *m* examples

$$Cost(w,b) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma\left(w \cdot x^{(i)} + b\right) + \left(1 - y^{(i)}\right) \log(1 - \sigma\left(w \cdot x^{(i)} + b\right))$$

The gradient for multiple data points is the sum of the individual gradients:

$$\frac{\partial Cost(w,b)}{\partial w_j} = \sum_{i=1}^{m} [\sigma\left(w \cdot x^{(i)} + b\right) - y^{(i)}] x_j^{(i)}$$

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 $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$

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

$\theta \leftarrow 0$

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

Multinomial logistic regression

Instead of binary classification, we often want more than two classes. For sentiment classification we might extend the class labels to be **positive, negative**, and **neutral**.

We want to know the probability of y for each class $c \in C$, $p(y = c|x)$.

To get a proper probability, we will use a **generalization of the sigmoid function** called the **softmax function.**

$$\text{softmax}(z_i) = \frac{e^{z_j}}{\sum_{j=1}^{k} e^{z_j}} \; 1 \leq i \leq k$$

Softmax

The softmax function takes in an input vector $z = [z_1, z_2, ..., z_k]$ and outputs a vector of values normalized into probabilities.

$$\text{softmax}(z) = [\frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}}, \frac{e^{z_2}}{\sum_{i=1}^{k} e^{z_i}}, \cdots, \frac{e^{z_k}}{\sum_{i=1}^{k} e^{z_i}}]$$

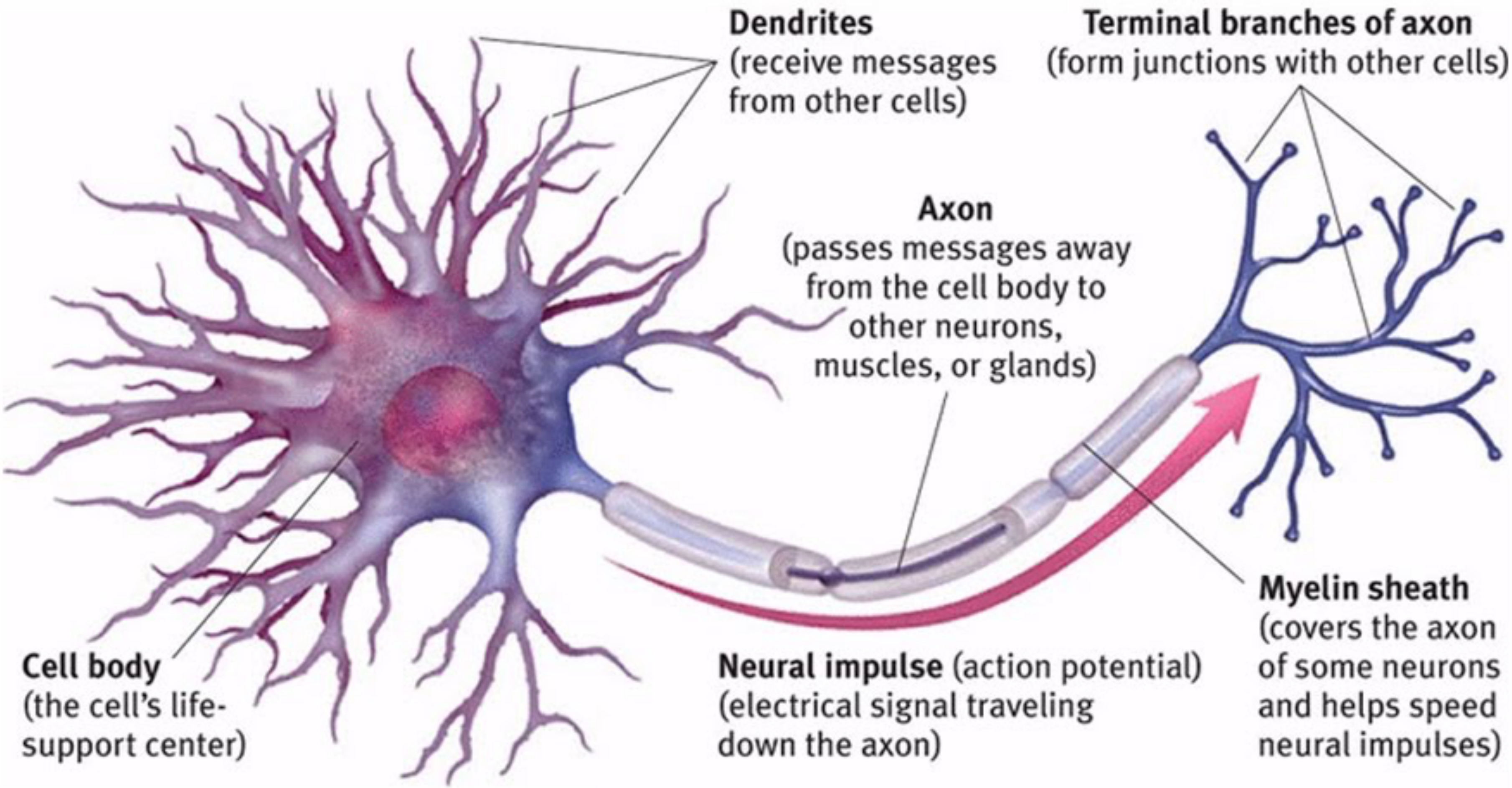
For example, for this input:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

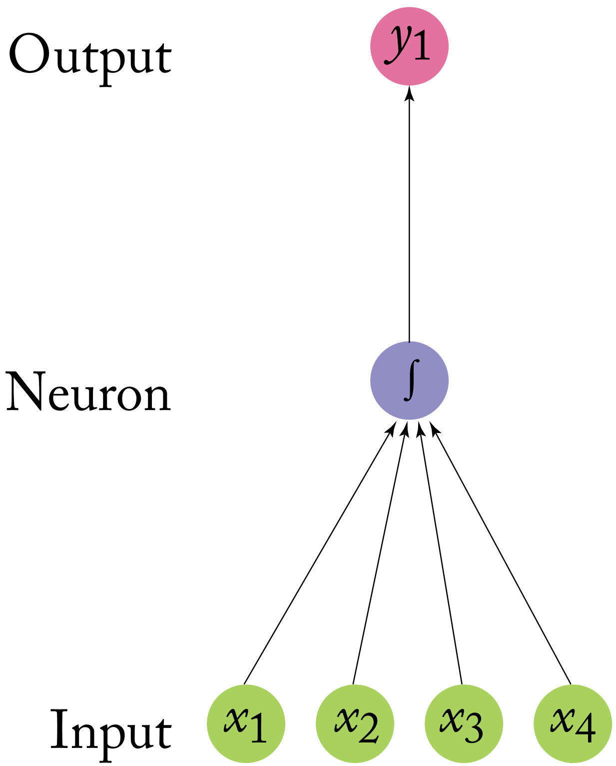
Softmax will output:

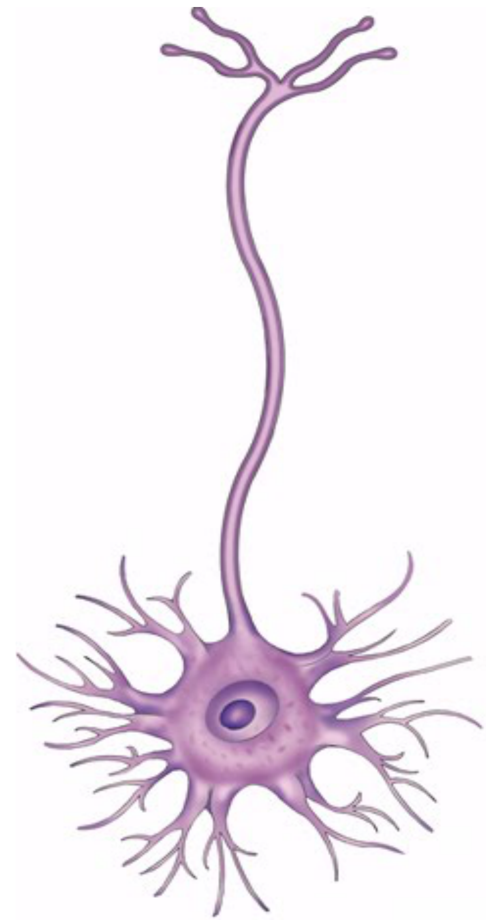
[0.056, 0.090, 0.007, 0.099, 0.74, 0.010]

Neural Networks: A brain-inspired metaphor

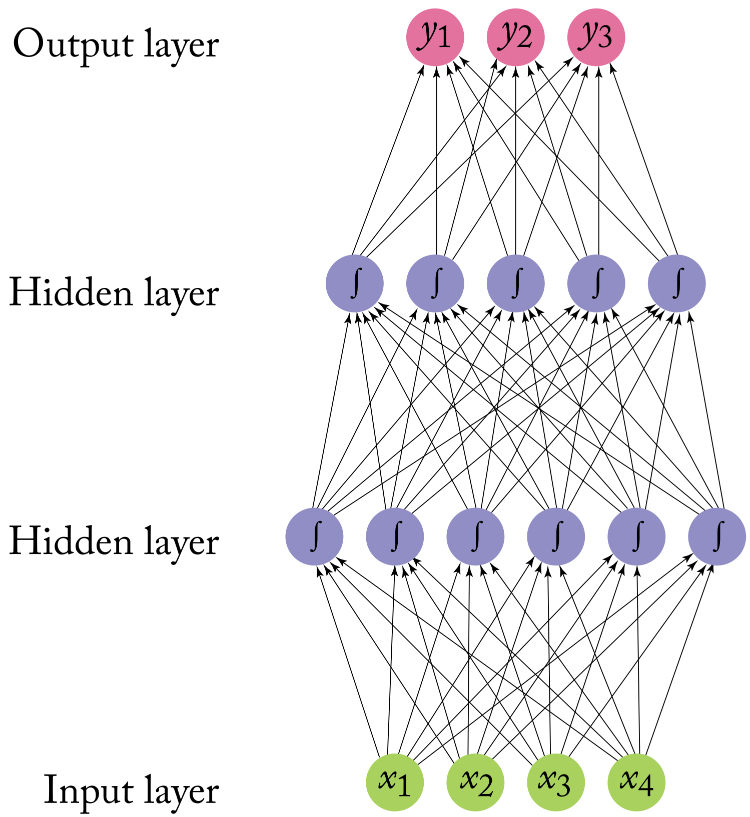


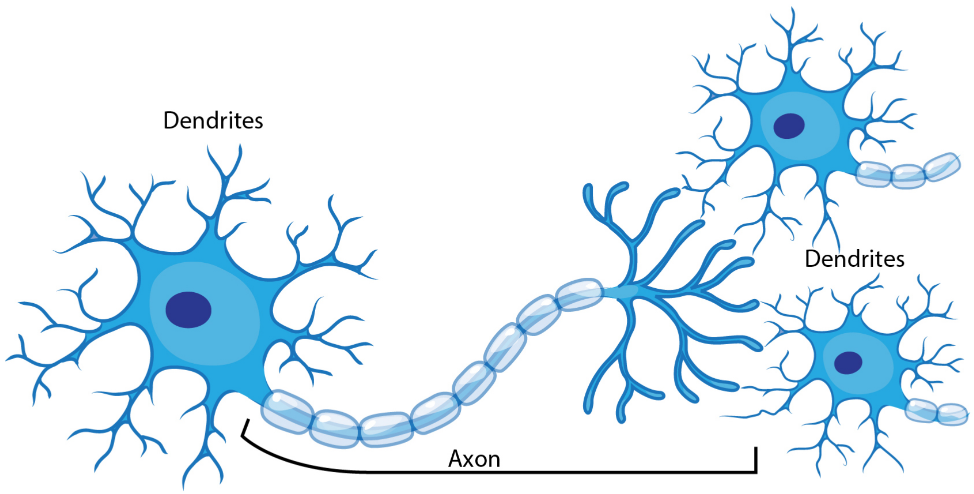
A single neuron





Neural networks





Mathematical Notation

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

$$\mathrm{NN}_{\mathrm{Perceptron}}(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$$

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

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.

$$\mathrm{NN}_{\mathrm{MLP1}}(\boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{W}^{\mathbf{1}} + \boldsymbol{b}^{\mathbf{1}})\boldsymbol{W}^{\mathbf{2}} + \boldsymbol{b}^{\mathbf{2}}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W}^{\mathbf{1}} \in \mathbb{R}^{d_{in} \times d_1}, \quad \boldsymbol{b}^{\mathbf{1}} \in \mathbb{R}^{d_1}, \quad \boldsymbol{W}^{\mathbf{2}} \in \mathbb{R}^{d_1 \times d_2}, \quad \boldsymbol{b}^{\mathbf{2}} \in \mathbb{R}^{d_2}$$

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

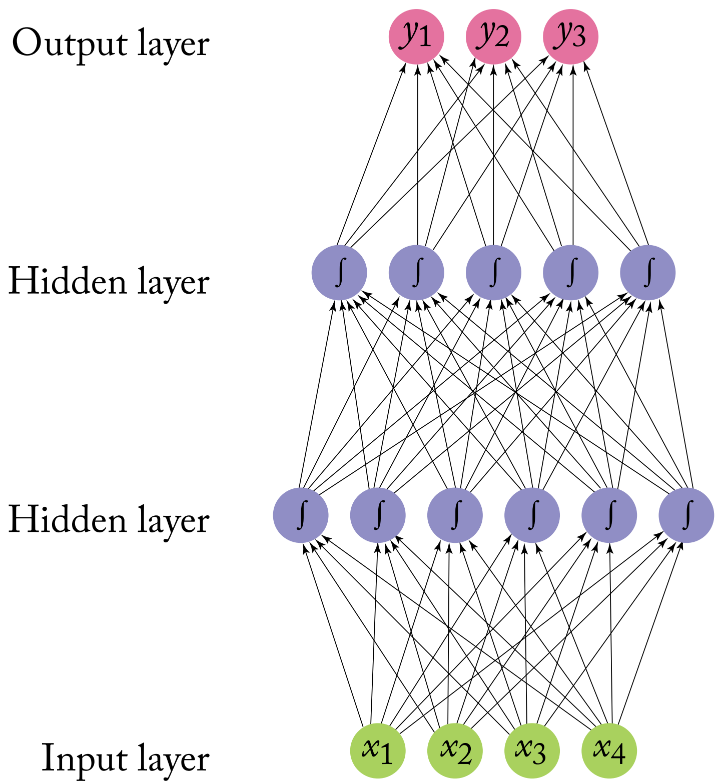
g is a nonlinear function (also an activation function),

$\mathbf{W}^2$ 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:

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



Same equation, but written with intermediary variables:

$$
\begin{aligned}
\mathrm{NN}_{\mathrm{MLP2}}(\boldsymbol{x}) &= \boldsymbol{y} \\
\boldsymbol{h}^1 &= g^1(\boldsymbol{x}\boldsymbol{W}^1 + \boldsymbol{b}^1) \\
\boldsymbol{h}^2 &= g^2(\boldsymbol{h}^1\boldsymbol{W}^2 + \boldsymbol{b}^2) \\
\boldsymbol{y} &= \boldsymbol{h}^2\boldsymbol{W}^3.
\end{aligned}
$$

Dimensions of the layers

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

$\mathbf{d}_{in}$ is the number of dimensions of the input vector

$\mathbf{d}_{out}$ is the number of dimensions of the output vector

A fully connected layer **l(x) = xW + b** with input size $\mathbf{d}_{in}$ and and output size $\mathbf{d}_{out}$ will have the following dimensions:

the dimensions of ***x*** are **1** x $\mathbf{d}_{in}$

the dimensions of ***W*** are $\mathbf{d}_{in}$ x $\mathbf{d}_{out}$

the dimensions of ***b*** are **1** x $\mathbf{d}_{out}$

Dimensions of the output layer

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

- Regression or scoring
- Binary classification

$\mathbf{d}_{out}$ = 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.

$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})$$

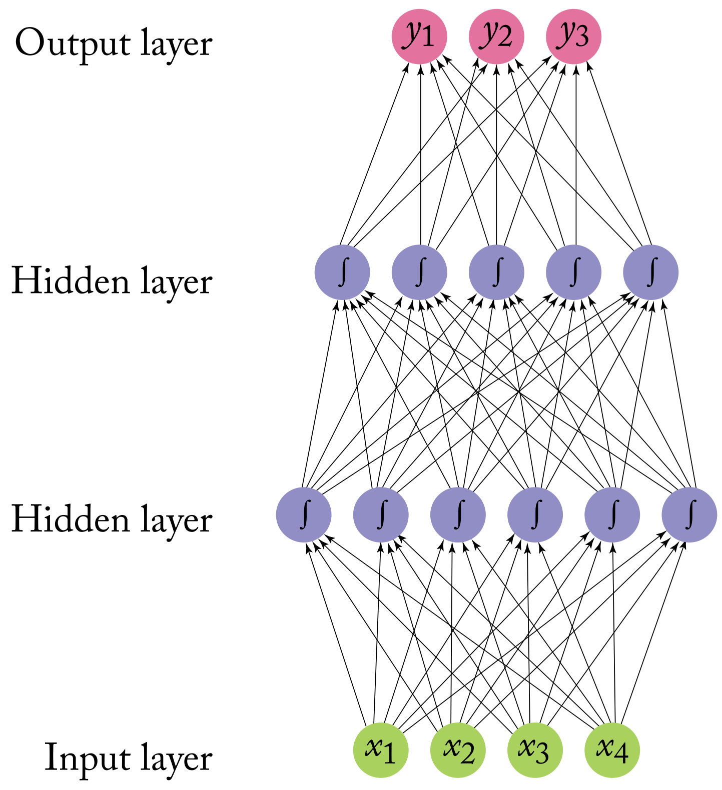
$$\hat{\boldsymbol{y}}_{[i]} = \frac{e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[i]}}}{\sum_j e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[j]}}}.$$

Representation Power

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

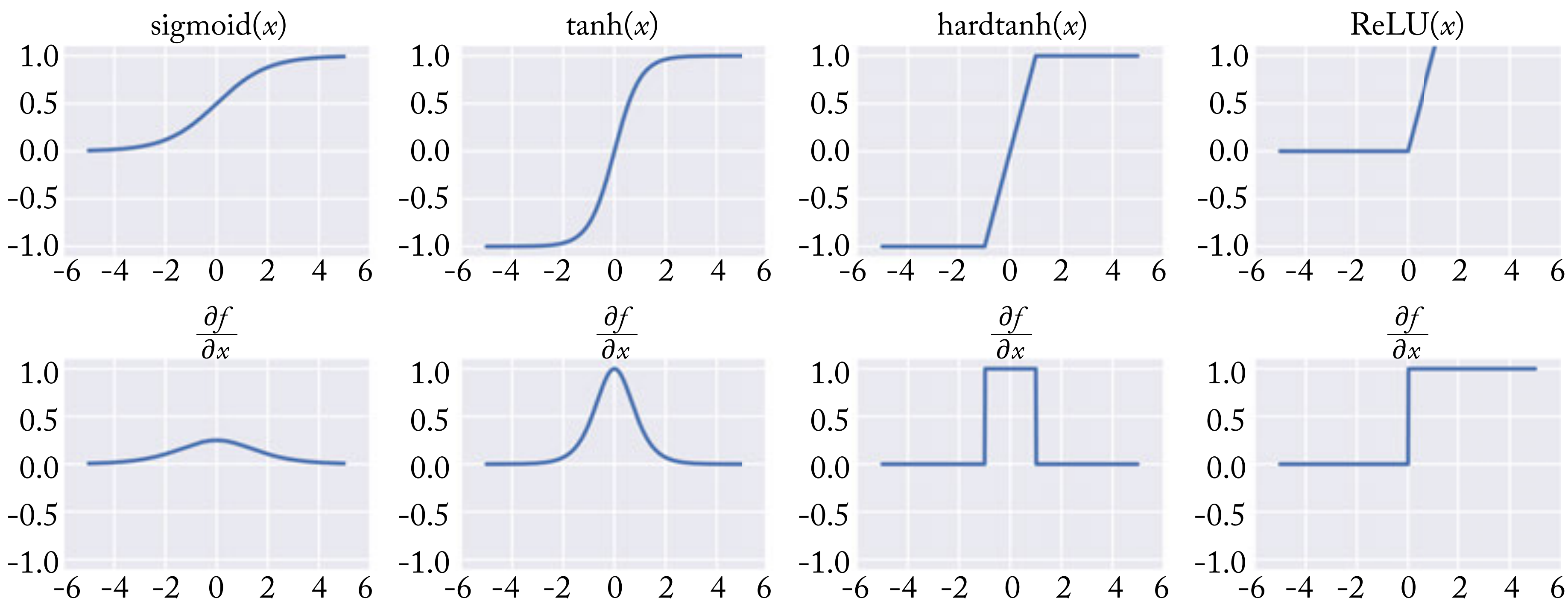
It can approximate a family of functions that includes all continuous functions on a closed and bounded subset of R^n

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.

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

Dropout attempts to avoid overfitting by randomly dropping (setting 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 $P(w_{i+1} | w_{k-i:i})$ from a large corpus.

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

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

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

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 $100{,}000^{10} = 10^{50}$ -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, <u>*using a multi-layer neural network.*</u>

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 **x** just concatenates v(w) for each of the k words:

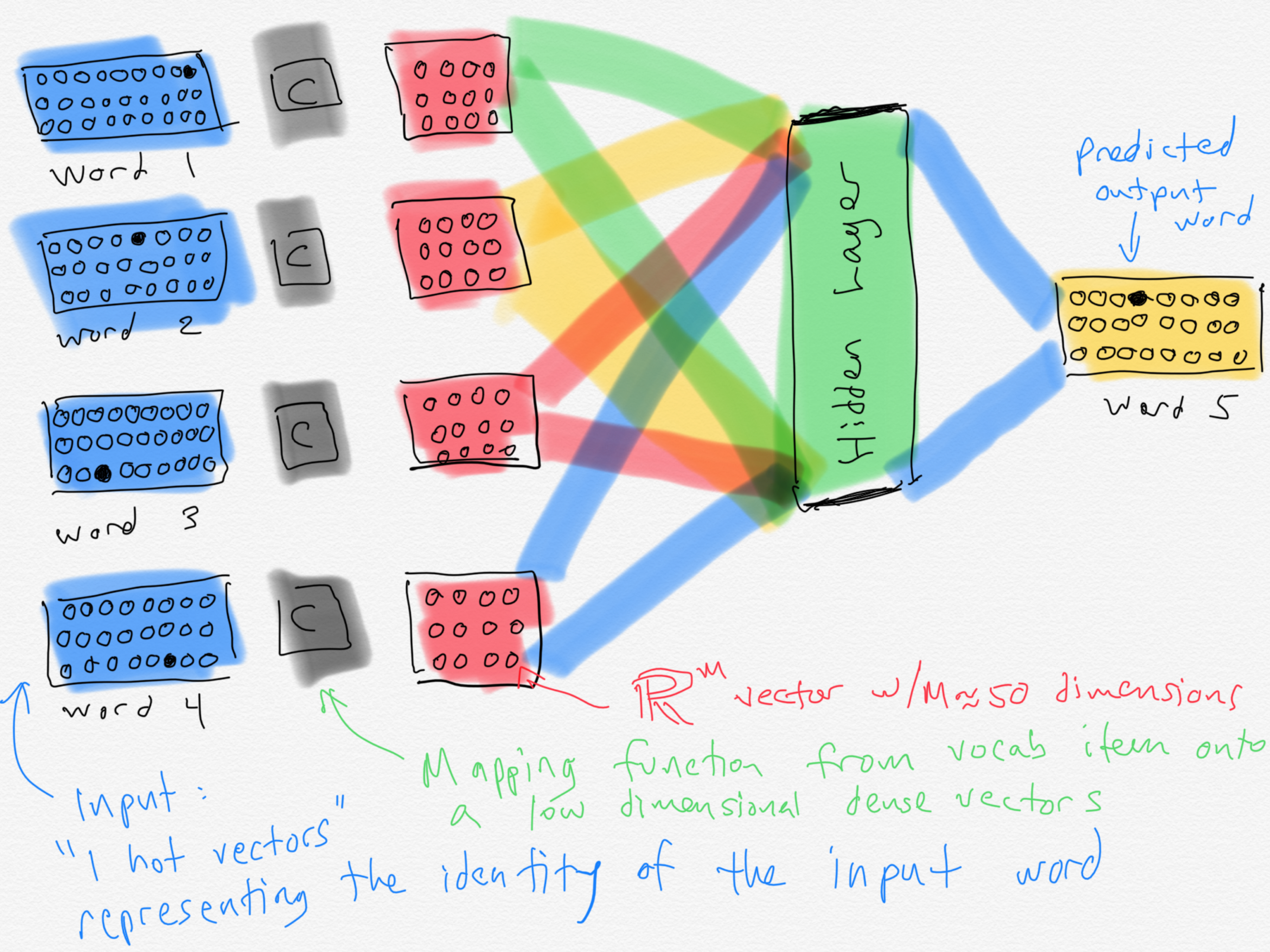
$$\boldsymbol{x} = [v(w_1); v(w_2); \ldots; v(w_k)]$$

Neural LMs

The input **x** is fed into a neural network with 1 or more hidden layers:

$$
\begin{aligned}
\hat{\boldsymbol{y}} = P(w_i | w_{1:k}) = LM(w_{1:k}) &= \mathrm{softmax}(\boldsymbol{h}\boldsymbol{W}^{\mathbf{2}} + \boldsymbol{b}^{\mathbf{2}}) \\
\boldsymbol{h} &= g(\boldsymbol{x}\boldsymbol{W}^{\mathbf{1}} + \boldsymbol{b}^{\mathbf{1}}) \\
\boldsymbol{x} &= [v(w_1); v(w_2); \ldots; v(w_k)] \\
v(w) &= \boldsymbol{E}_{[w]}
\end{aligned}
$$

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


word 1
word 2
word 3
word 4
C
C
C
C
Hidden Layer
predicted output word
word 5
$\mathbb{R}^m$ vector w/ $m \approx 50$ dimensions
Mapping function from vocab item onto a low dimensional dense vectors
input: "1 hot vectors" representing the identity of the input word

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 we often use alternative loss functions or approximations.

Advantages of NN LMs

Better results. They achieve better preplexity 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 in training, because it did observe *blue car* and *red car*.

A by-product of training are word embeddings!

Language Modeling

Goal: Learn a **function** that returns the joint probability

Primary difficulty:

1. There are too many parameters to accurately estimate. This is sometimes called the "curse of dimensionality"
2. In n-gram-based models we fail to generalize to related words / word sequences that we <u>have</u> observed.

Curse of dimensionality / sparse statistics

Suppose we want a joint distribution over 10 words.
Suppose we have a vocabulary of size 100,000.

$$100{,}000^{10} = 10^{50} \text{ parameters}$$

This is too high to estimate from data.

Chain rule

In LMs we user chain rule to get the conditional probability of the next word in the sequence given all of the previous words:

$$P(w_1 w_2 w_3 \ldots w_t) = \prod_{t=1}^{T} P(w_t | w_1 \ldots w_{t-1})$$

What assumption do we make in n-gram LMs to simplify this?

The probability of the next word only depends on the previous *n*-1 words.

A small *n* makes it easier for us to get an estimate of the probability from data.

Probability tables

We construct tables to look up the probability of seeing a word given a history.

curse of	$P(w_t \mid w_{t-n} \dots w_{t-1})$
dimensionality	
azure	
knowledge	
oak	

The tables only store observed sequences.

What happens when we have a new (unseen) combination of n words?

Unseen sequences

What happens when we have a new (unseen) combination of n words?

1. Back-off
2. Smoothing / interpolation

We are basically just stitching together short sequences of observed words.

Alternate idea

Let's try **generalizing**.

Intuition: Take a sentence like

The cat is walking in the bedroom

And use it when we assign probabilities to similar sentences like

The dog is running around the room

A Neural Probabilistic LM

Bengio et al NIPS 2003

1. Use a vector space model where the words are vectors with real values $\mathbb{R}^m$. m=30, 60, 100. This gives a way to compute word similarity.
2. Define a function that returns a joint probability of words in a sequence based on a sequence of these vectors.
3. Simultaneously learn the word representations **and** the probability function from data.

Seeing one of the cat/dog sentences allows them to increase the probability for that sentence **and** its combinatorial # of **"neighbor" sentences** in vector space.

A Neural Probabilistic LM

Given:

A training set $w_1 \ldots w_t$ where $w_t \in V$

Learn:

$f(w_1 \ldots w_t) = P(w_t | w_1 \ldots w_{t-1})$

Subject to giving a high probability to an unseen text/dev set (e.g. minimizing the perplexity)

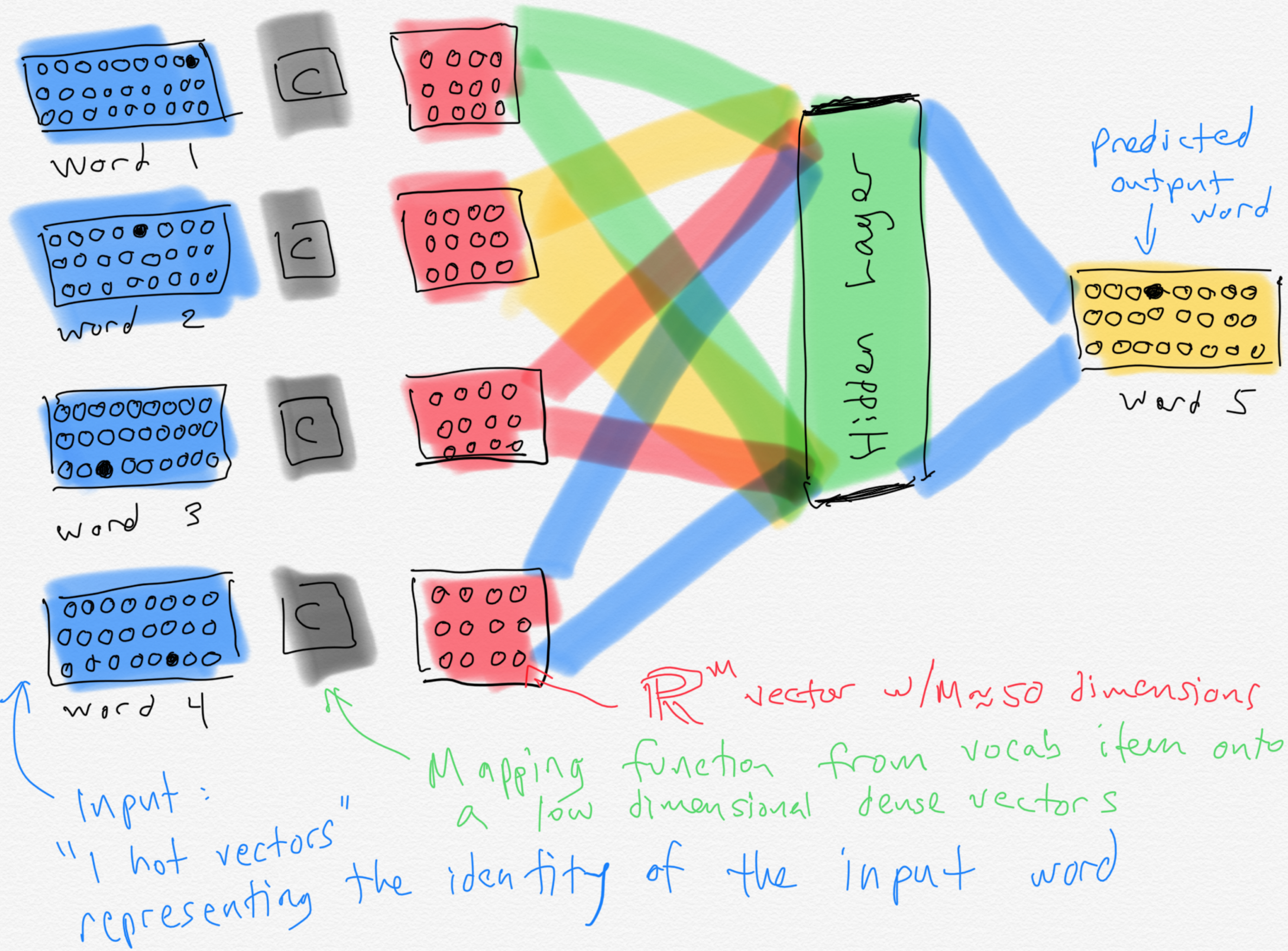
Constraint:

Create a proper probability distribution (e.g. sums to 1) so that we can take the product of conditional probabilities to get the joint probability of a sentence

A Neural Probabilistic LM

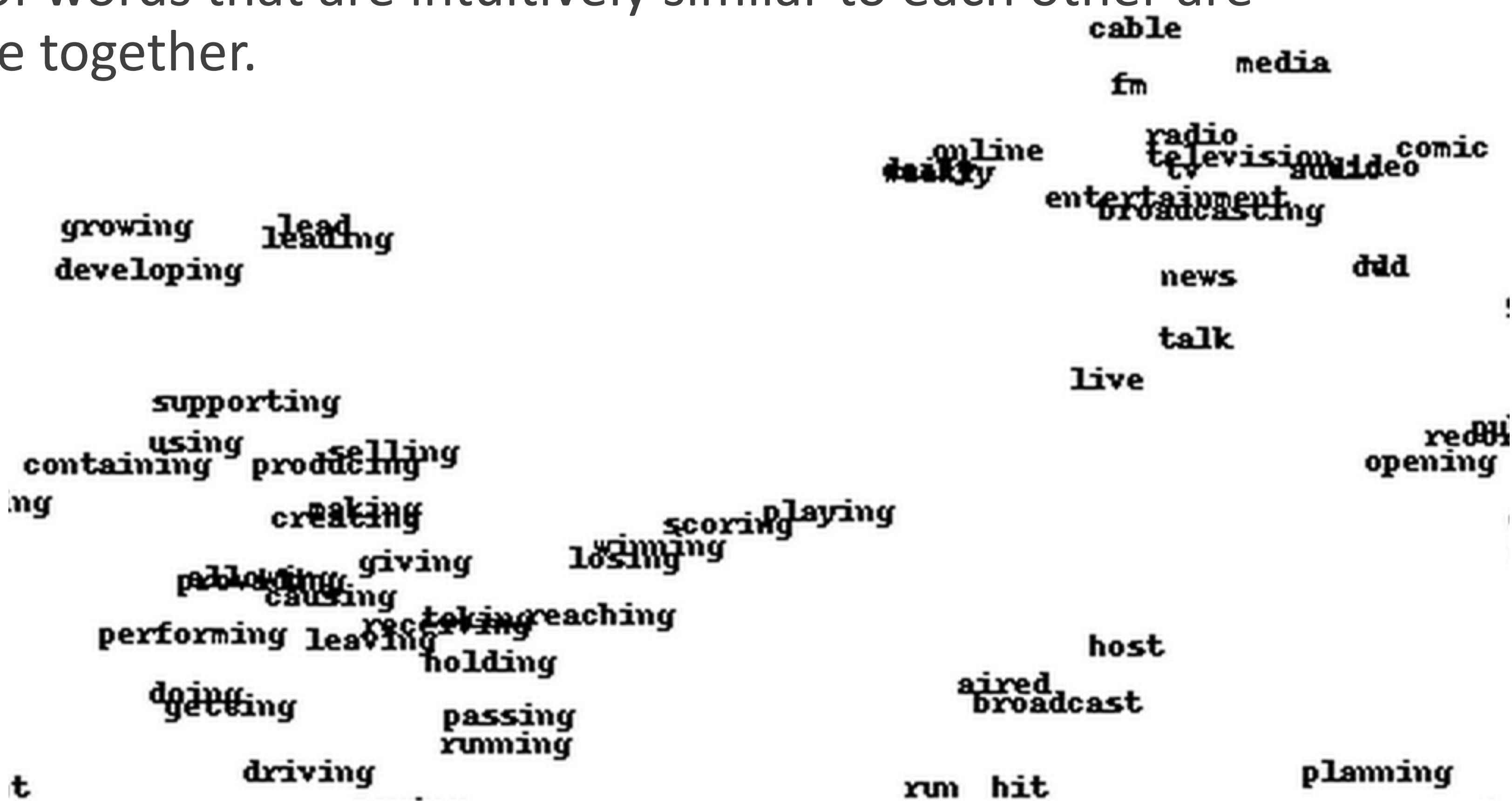
1. Create a mapping function C from any word in V onto $\mathbb{R}^M$. Store this in a V-by-M matrix. Initialize it with singular value decomposition (SVD).
2. The neural architecture: a function *g* maps sequence of word vectors onto a probability distribution over the vocabulary V

$$g(C(w_{t-n}) \ldots C(w_{t-1})) = P(w_t | w_{t-n} \ldots w_{t-1})$$


word 1
word 2
word 3
word 4
C
C
C
C
Hidden Layer
predicted output word
word 5
ℝ^m vector w/ M≈50 dimensions
Mapping function from vocab item onto a low dimensional dense vectors
input: "1 hot vectors" representing the identity of the input word

Word embeddings

When the ~50 dimensional vectors that result from training a neural LM are projected down to 2-dimensions, we see a lot of words that are intuitively similar to each other are close together.



Current state of the art neural LMs

[ELMo]

[GPT]

[BERT]

[GPT-2]