text classification 3: neural networks

CS 585, Fall 2018

Introduction to Natural Language Processing

http://people.cs.umass.edu/~miyyer/cs585/

Mohit Iyyer

College of Information and Computer Sciences

University of Massachusetts Amherst

some slides adapted from Jordan Boyd-Graber and Richard Socher

questions from last time…

see https://stats.stackexchange.com/questions/81659/mutual-information-versus-correlation

- PMI vs covariance matrix?
- why do we have two embedding matrices (**W** and **C**) in word2vec?

[2] Throughout this note, we assume that the words and the contexts come from distinct vocabularies, so that, for example, the vector associated with the word *dog* will be different from the vector associated with the context *dog*. This assumption follows the literature, where it is not motivated. One motivation for making this assumption is the following: consider the case where both the word *dog* and the context *dog* share the same vector v. Words hardly appear in the contexts of themselves, and so the model should assign a low probability to $p(dog|dog)$, which entails assigning a low value to $v \cdot v$ which is impossible.

Goldberg & Levy, 2014

- what distribution do we draw negative samples from? unigram ^ 0.75. why? *shrug*
- HW 1 encoding issues?

Summary: How to learn word2vec (skip-gram) embeddings

Start with V random 300-dimensional vectors as initial embeddings

Use logistic regression, the second most basic classifier used in machine learning after naïve bayes

- Take a corpus and take pairs of words that co-occur as positive examples
- Take pairs of words that don't co-occur as negative examples
- Train the classifier to distinguish these by slowly adjusting all the embeddings to improve the classifier performance
- Throw away the classifier code and keep the embeddings.

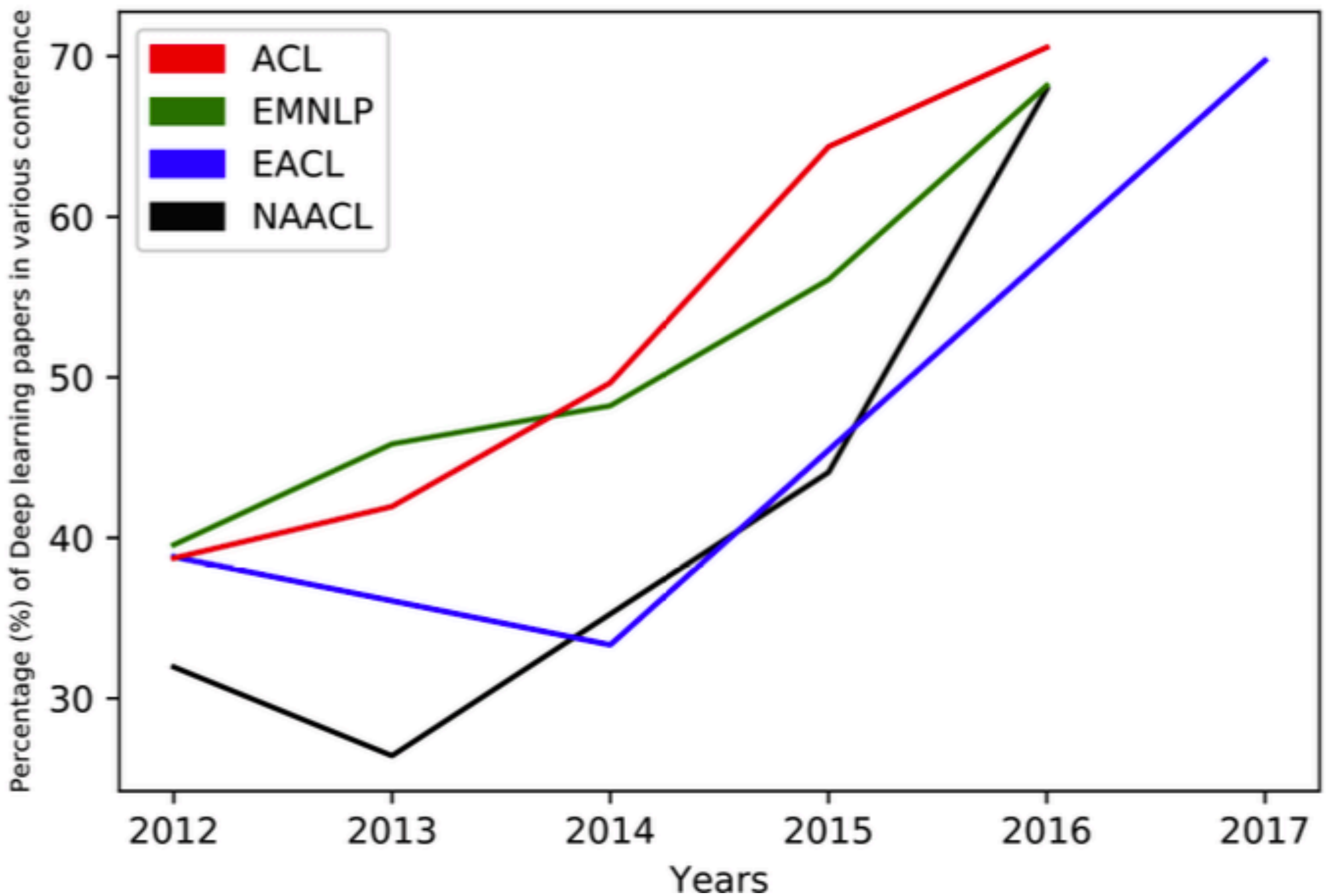
qualitatively evaluating word embeddings: nearest neighbors demo

https://projector.tensorflow.org/

text classification

- input: some text $\mathbf{x}$ (e.g., sentence, document)
- output: a label $\mathbf{y}$ (from a finite label set)
- goal: learn a mapping function f from $\mathbf{x}$ to $\mathbf{y}$

the rise of deep learning in natural language processing



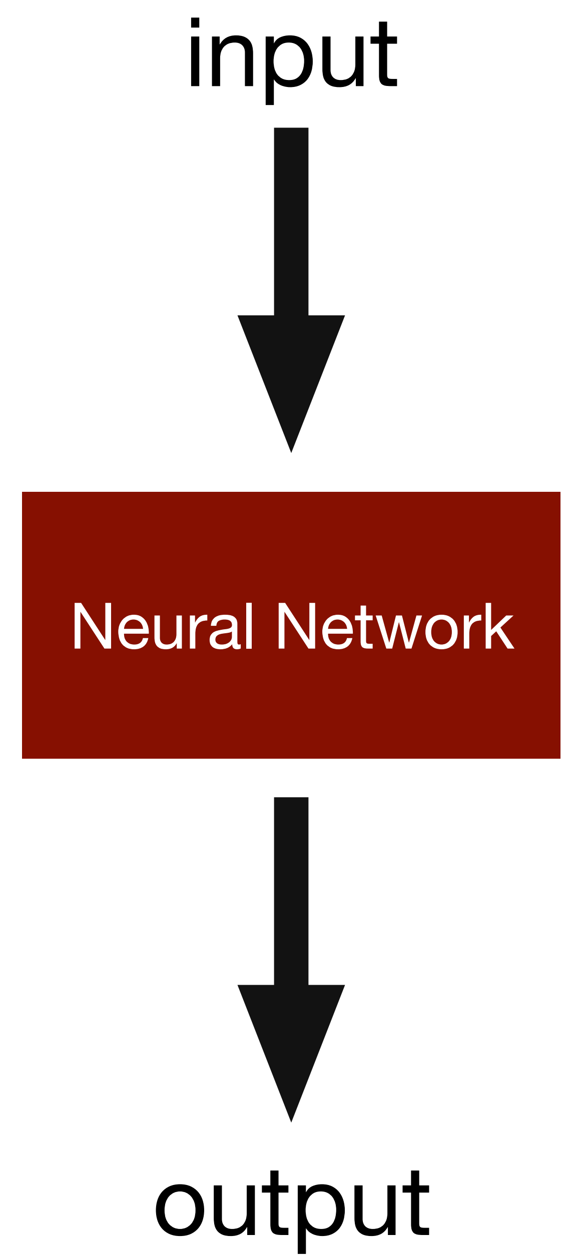
neural classification

- goal: avoid feature engineering… why?
- general model architectures that work well for many different datasets (and tasks!)
- for medium-to-large datasets, deep learning methods generally outperform naive Bayes / feature-based logistic regression

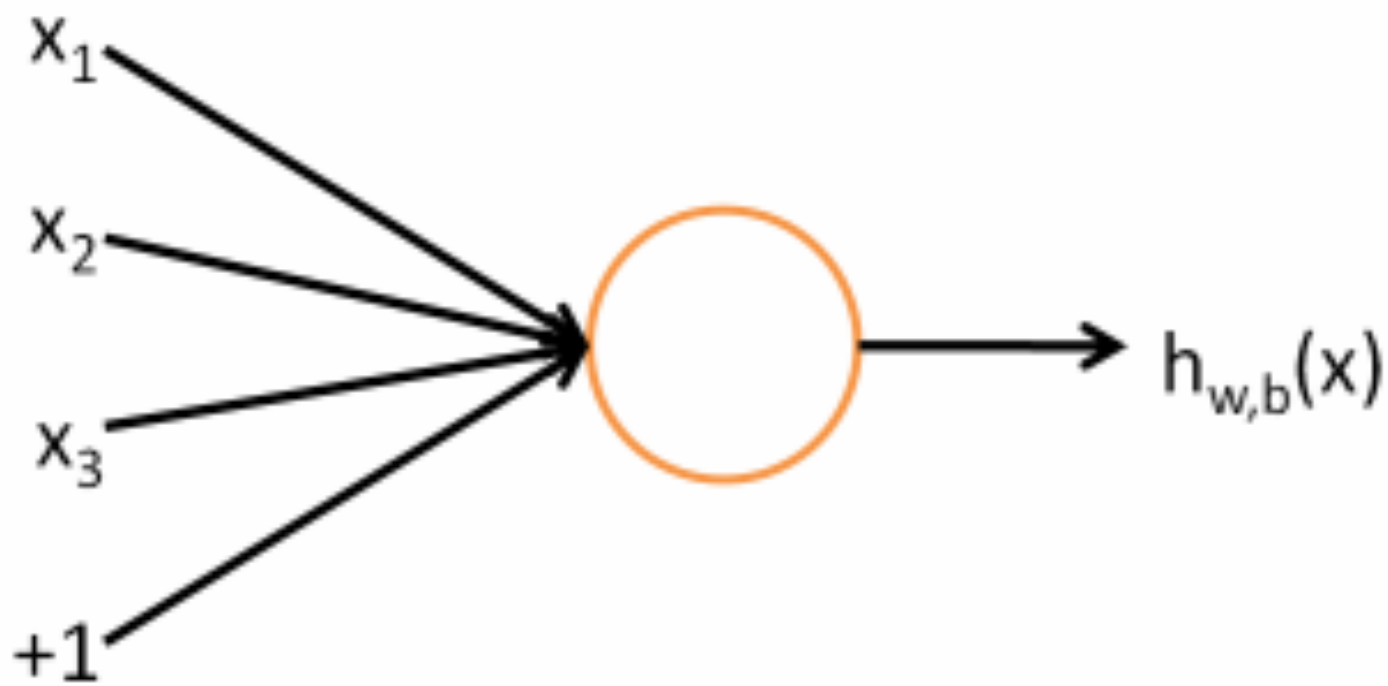
what is **deep learning**?

$$f(\text{input}) = \text{output}$$

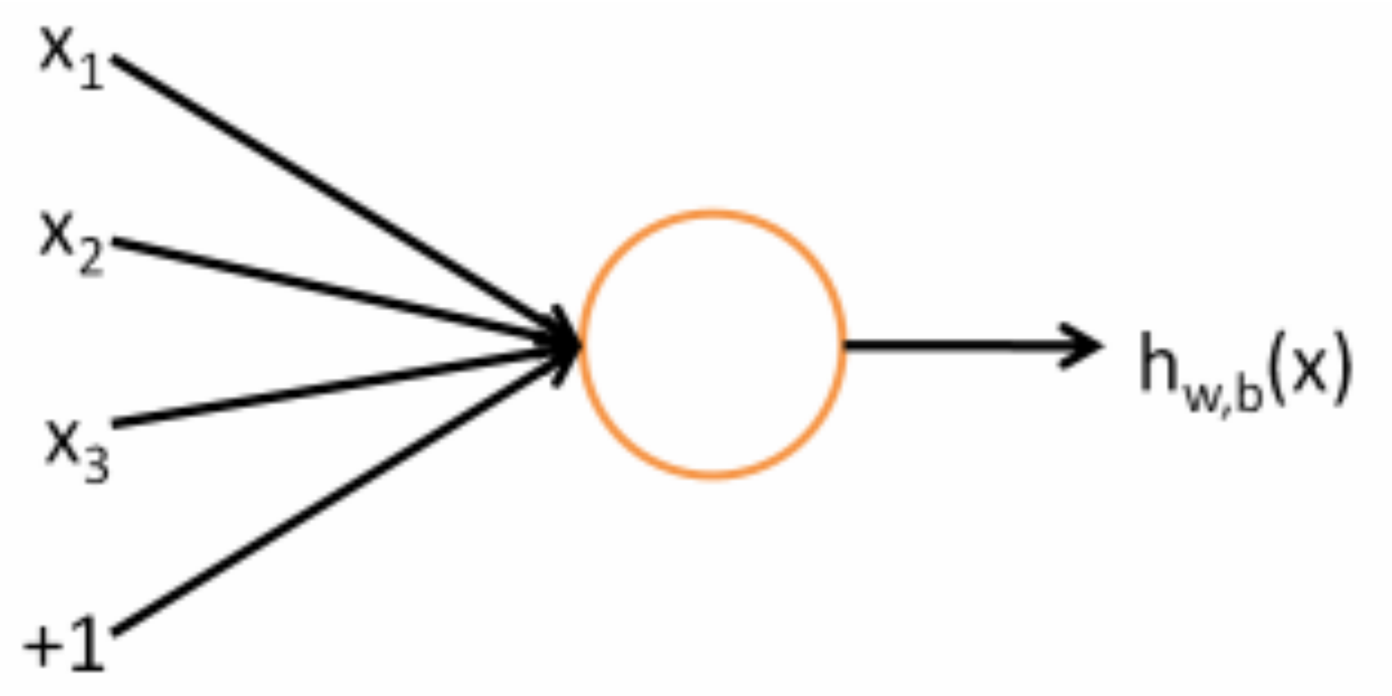
what is **deep learning**?



Logistic Regression by Another Name: Map inputs to output



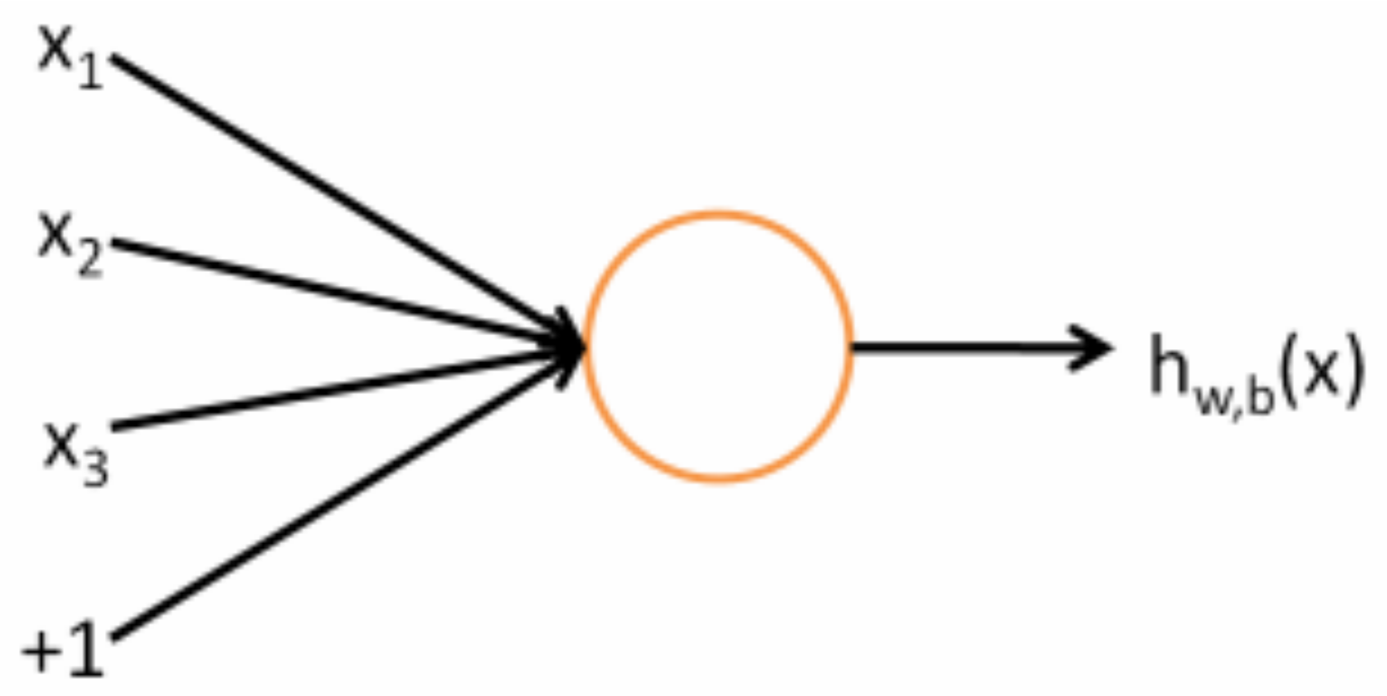
Logistic Regression by Another Name: Map inputs to output



Input
Vector $x_1 \ldots x_d$

inputs encoded as
real numbers

Logistic Regression by Another Name: Map inputs to output



Output

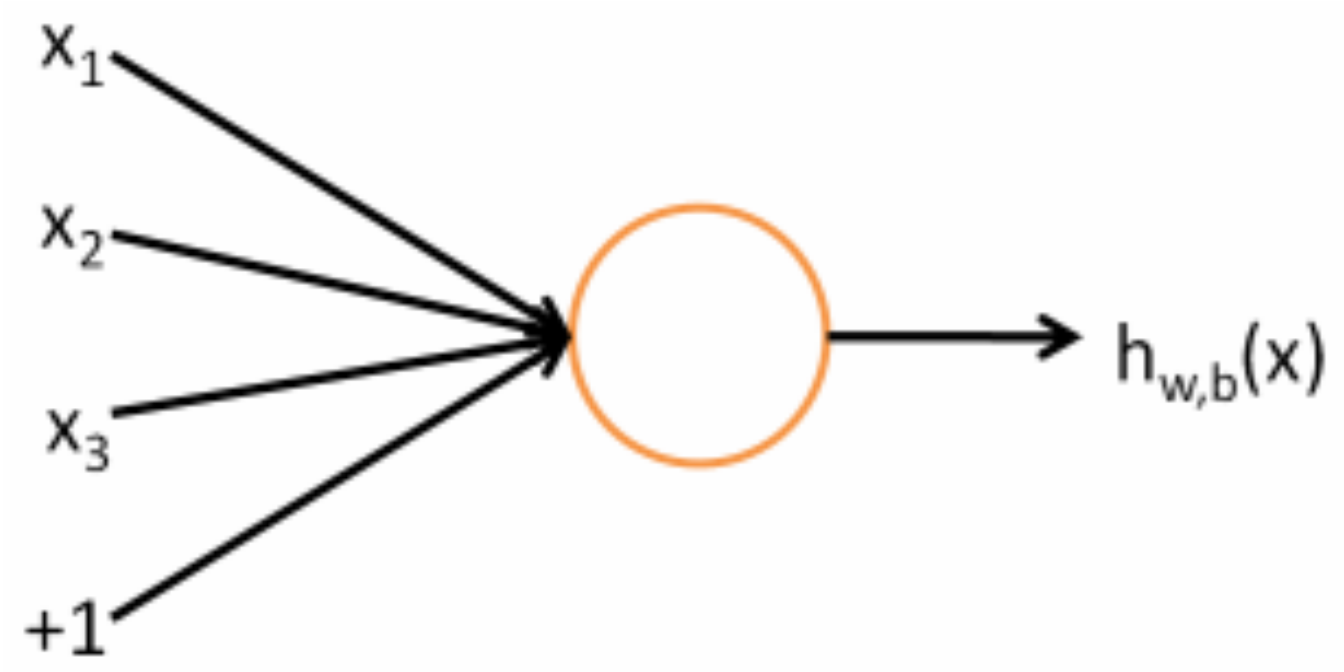
$$f\left(\sum_i W_i x_i + b\right)$$

Input

Vector $x_1 \ldots x_d$

multiply inputs by

Logistic Regression by Another Name: Map inputs to output



Output

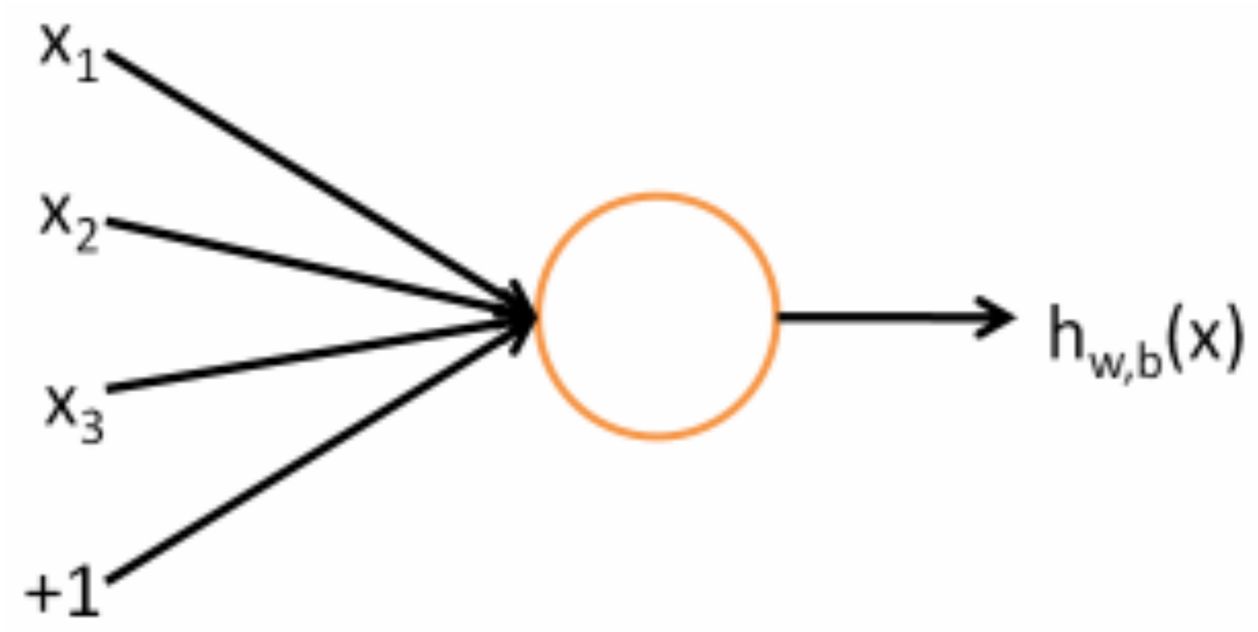
$$f\left(\sum_i W_i x_i + b\right)$$

Input

Vector $x_1 \ldots x_d$

add bias

Logistic Regression by Another Name: Map inputs to output



Input

Vector $x_1 \dots x_d$

Output

$$f\left(\sum_i W_i x_i + b\right)$$

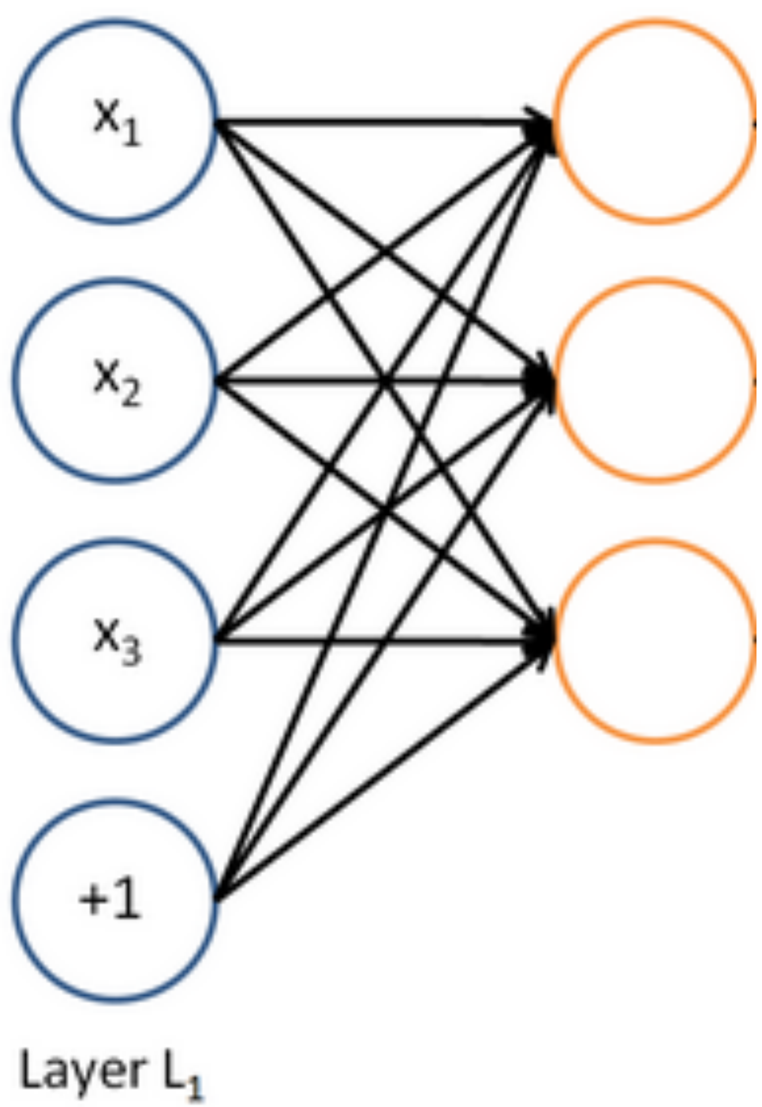
Activation

$$f(z) \equiv \frac{1}{1 + \exp(-z)}$$

pass through nonlinear sigmoid

A neural network
= running several logistic regressions at the same time

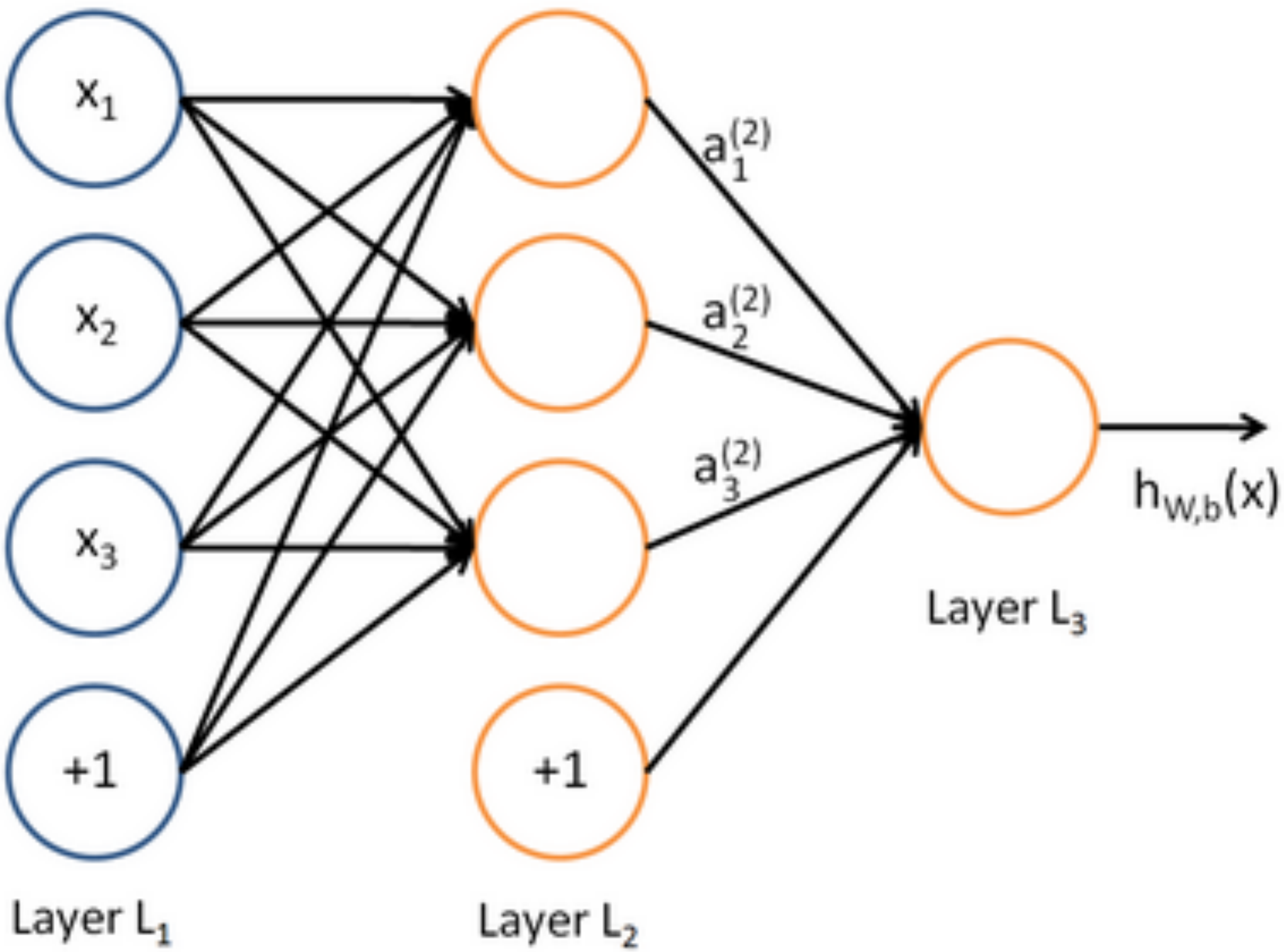
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

A neural network
= running several logistic regressions at the same time

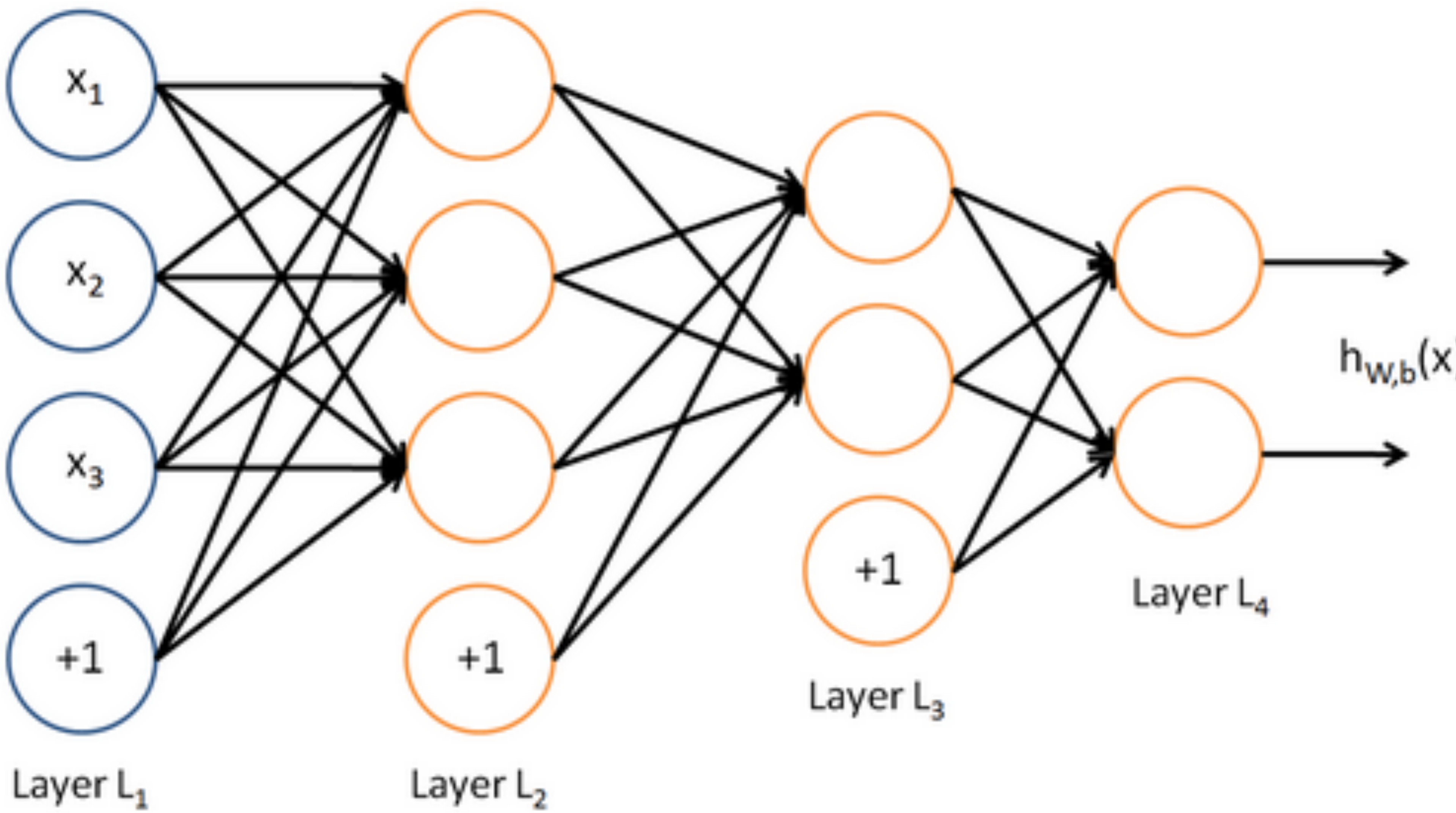
… which we can feed into another logistic regression function



It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

A neural network
= running several logistic regressions at the same time

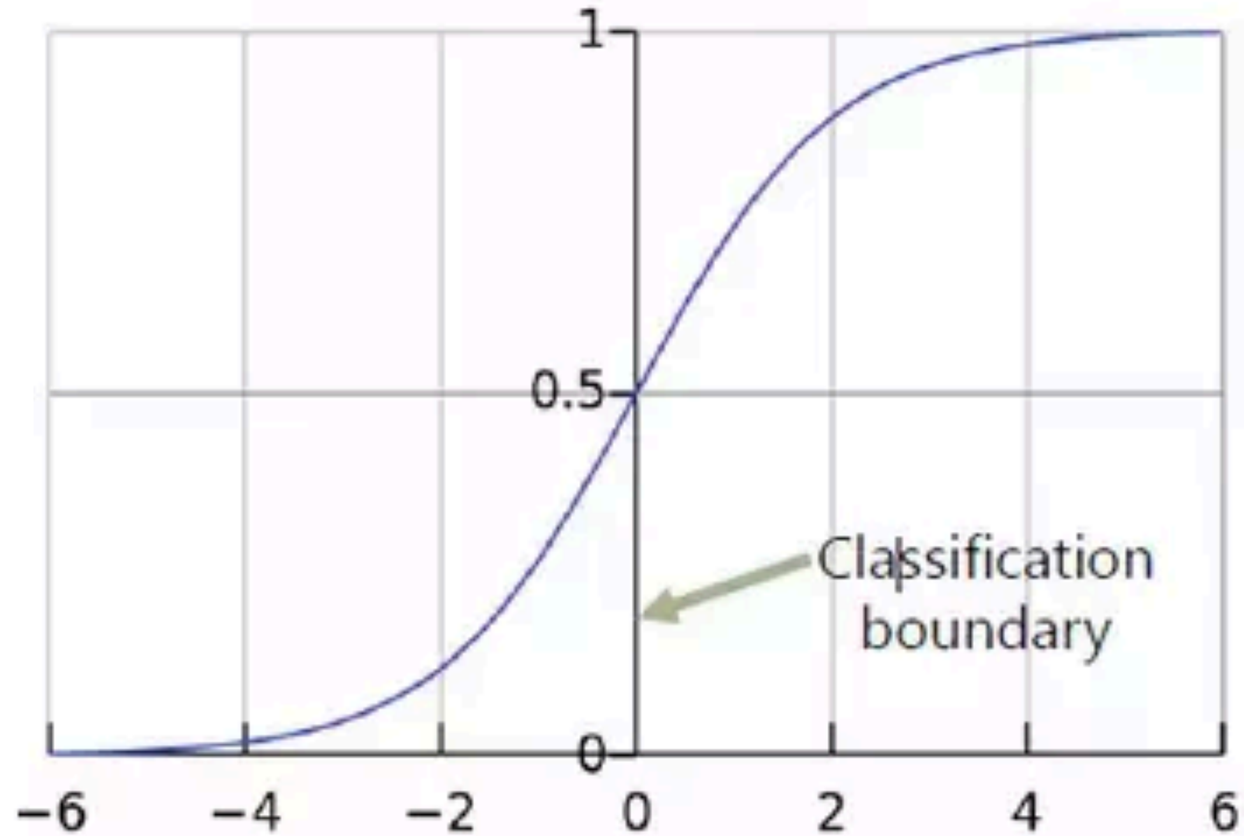
Before we know it, we have a multilayer neural network….



logistic regression is a *linear* classifier…
its decision boundary is linear in **x**

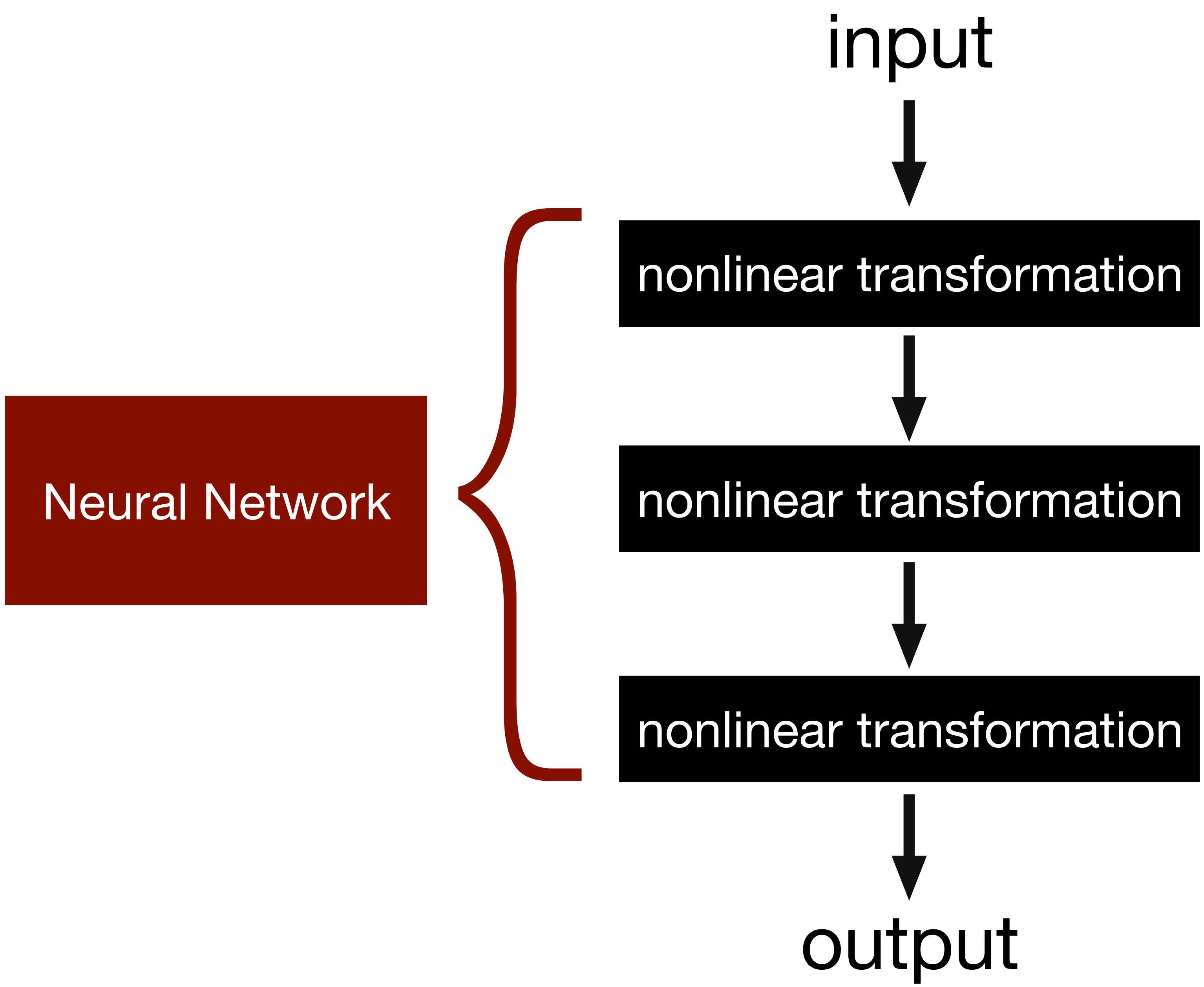
sigmoid function

$$\frac{1}{1 + e^{-\beta x}}$$

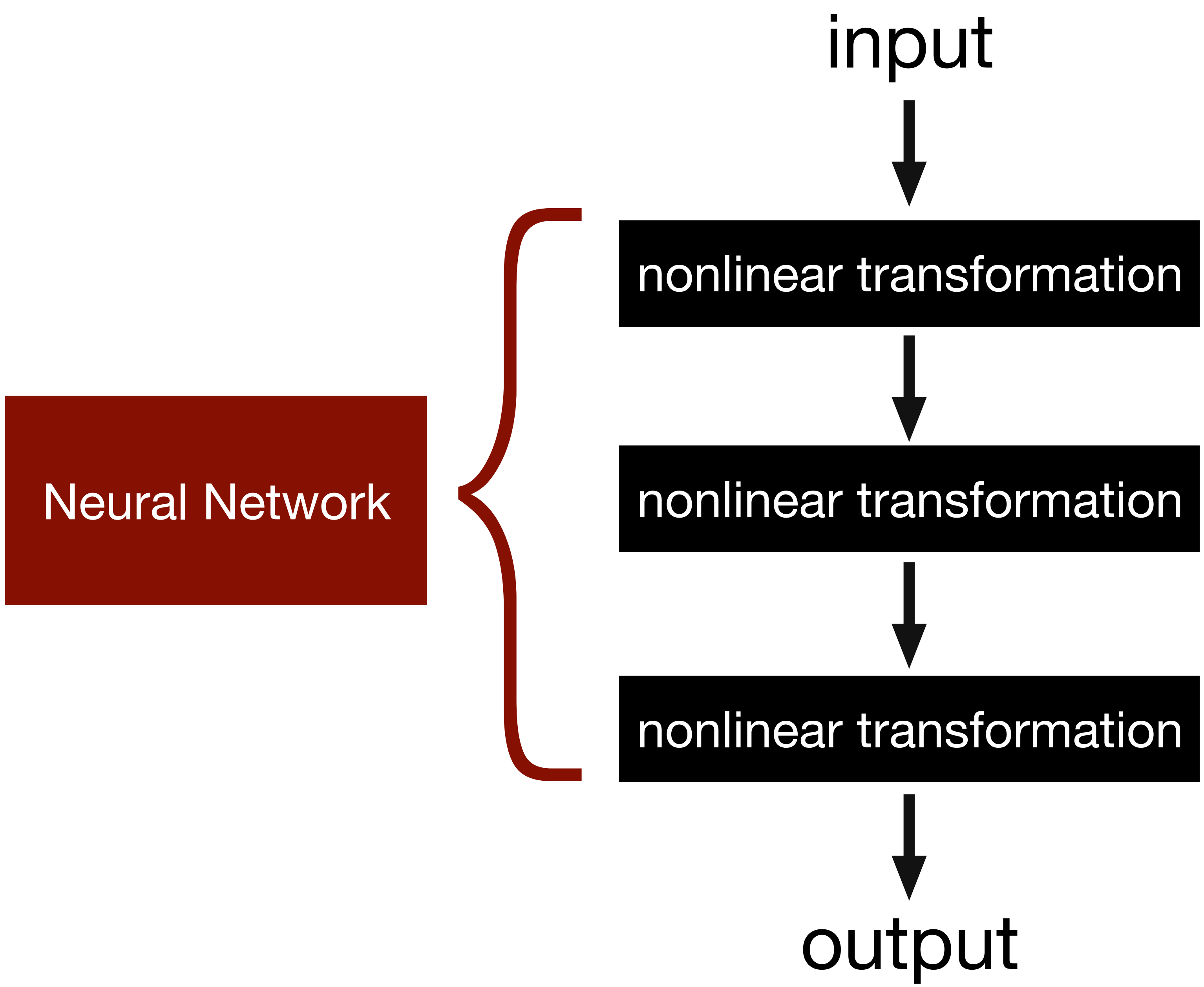


$$\beta x$$

what is deep learning?

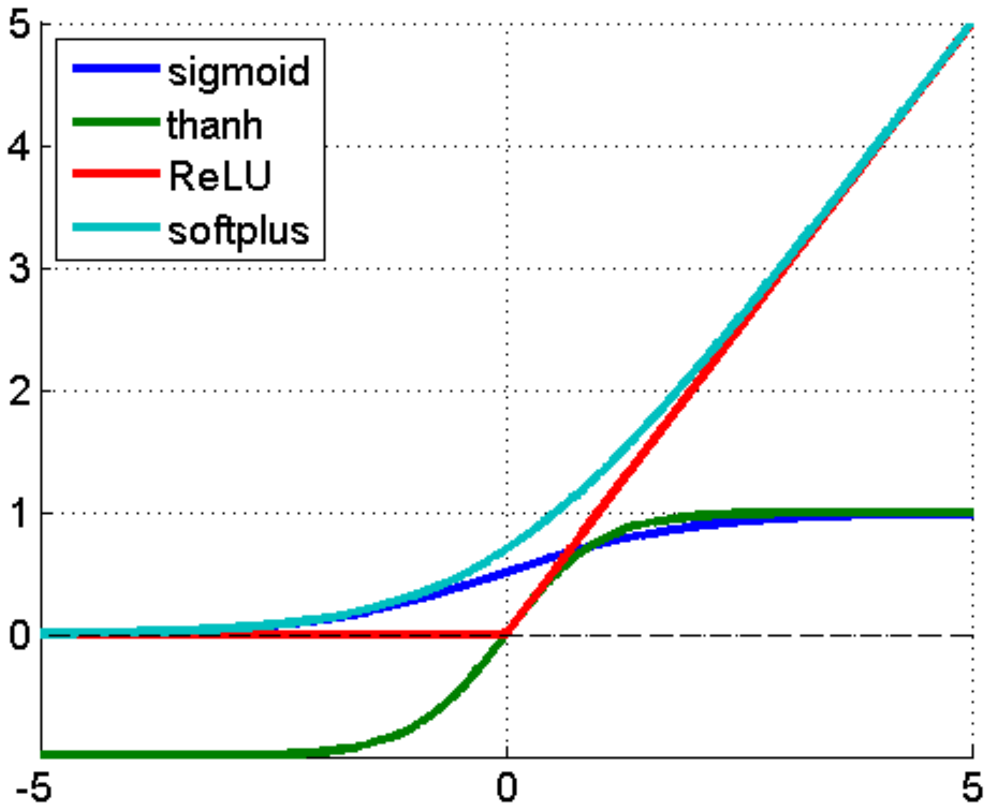


what is deep learning?



$$h_n = f(Wh_{n-1} + b)$$

Better name: non-linearity



- Logistic / Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$

- tanh

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

- ReLU

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$

- SoftPlus: $f(x) = \ln(1 + e^x)$

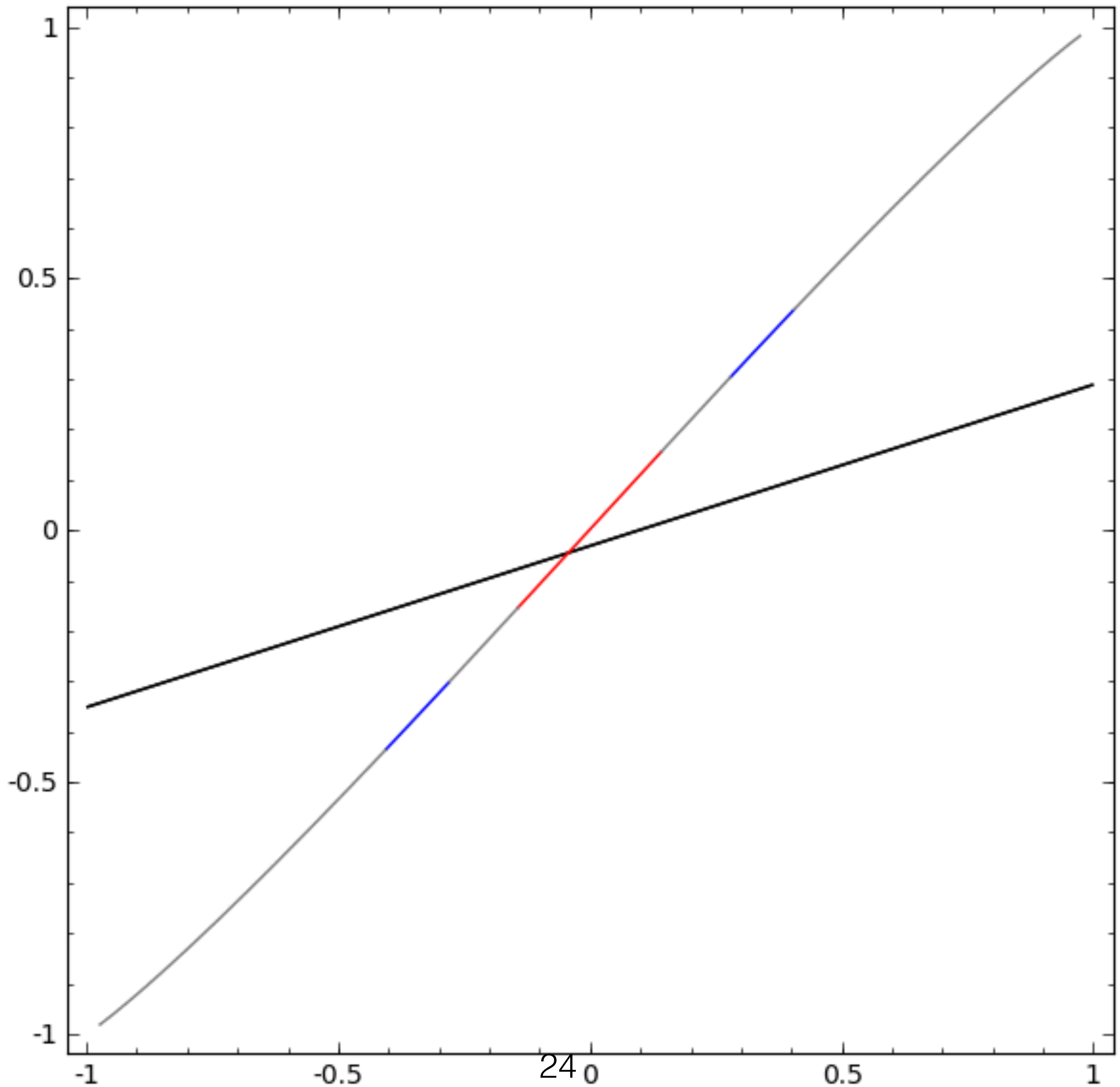
is a multi-layer neural network with no nonlinearities
(i.e., f is the identity $f(\mathbf{x}) = \mathbf{x}$)
more powerful than a one-layer network?

is a multi-layer neural network with no nonlinearities
(i.e., f is the identity $f(\mathbf{x}) = \mathbf{x}$)
more powerful than a one-layer network?

No! You can just compile all of the layers into a single transformation!

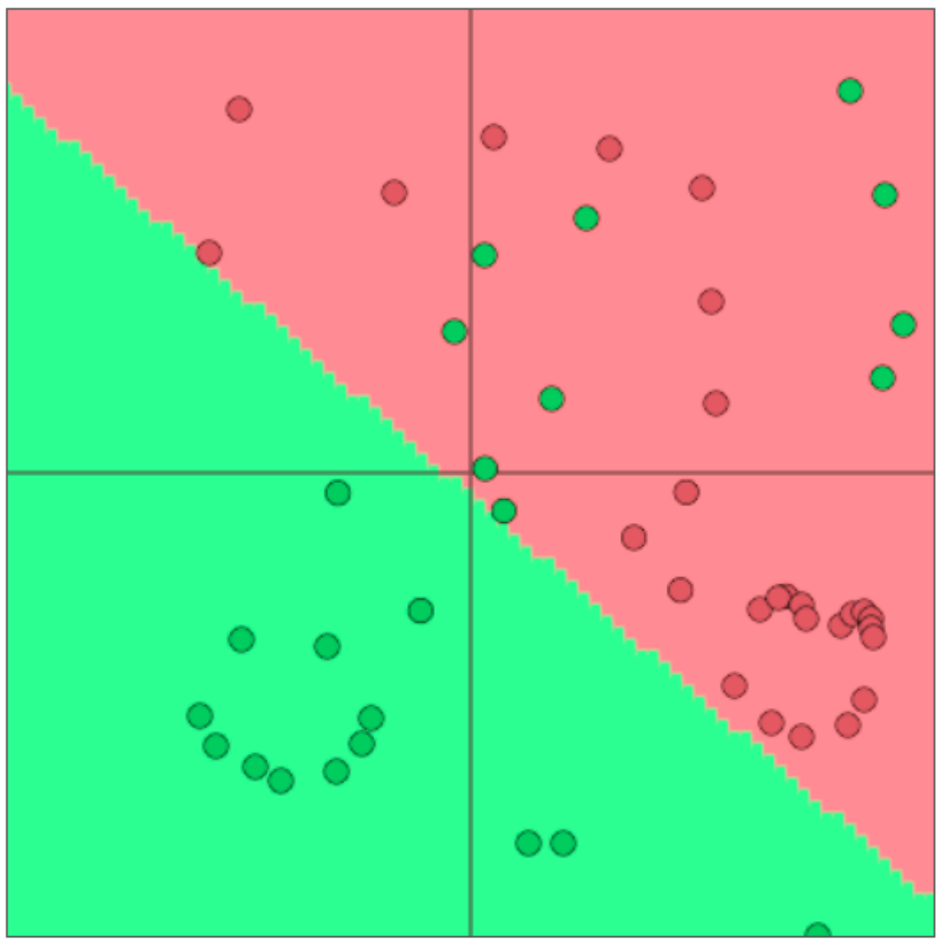
$$y = f(W_3 f(W_2 f(W_1 x))) = Wx$$

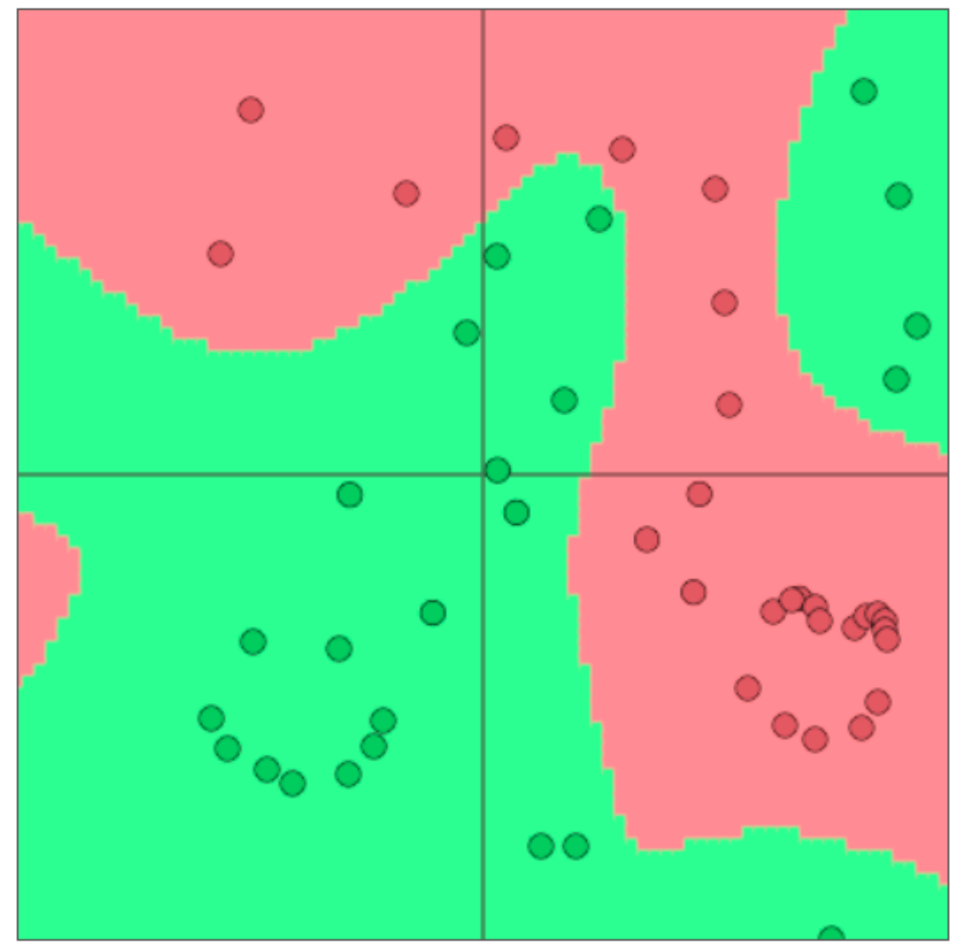
why nonlinearities?

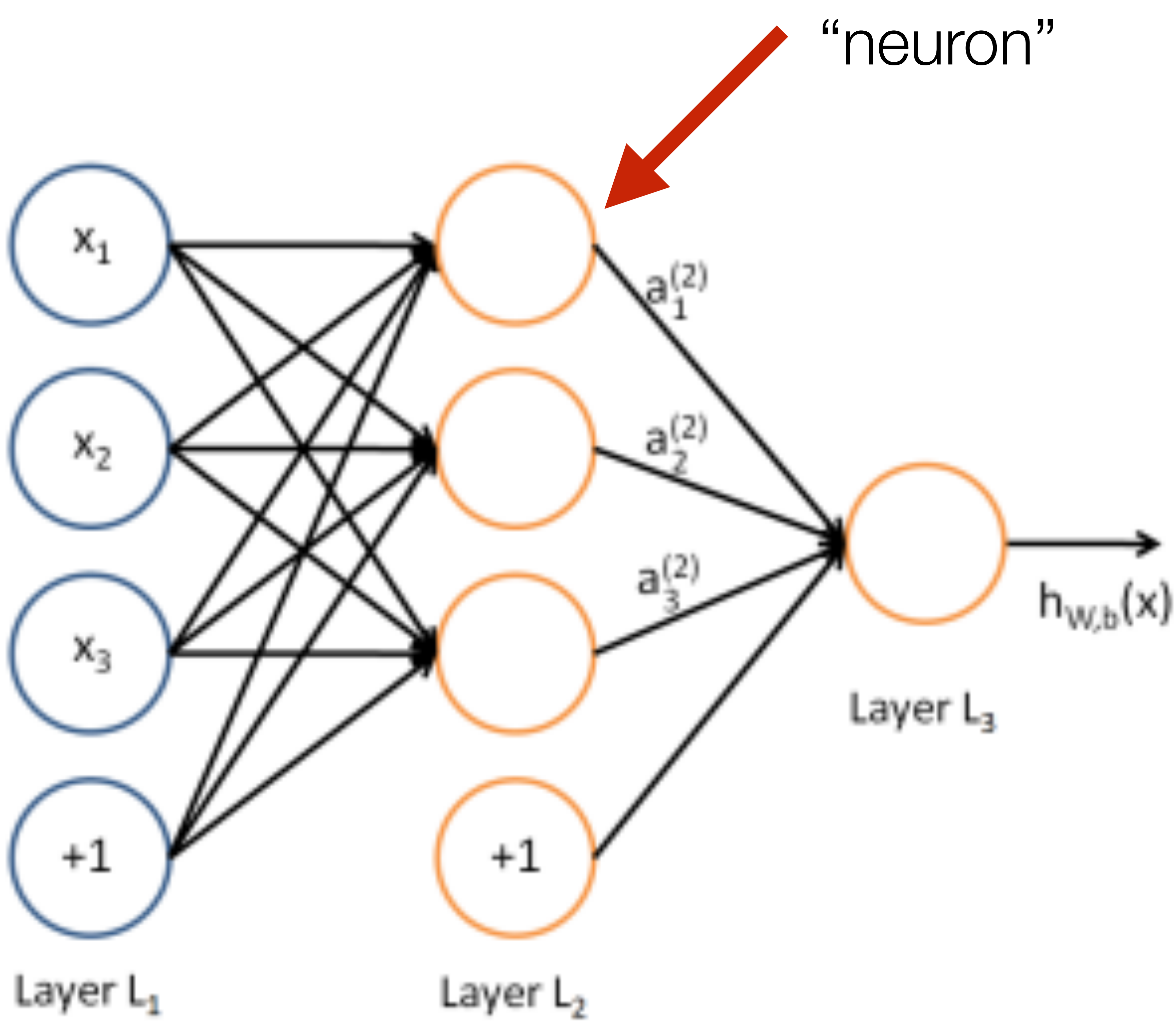


credit for figure:
Christopher Olah

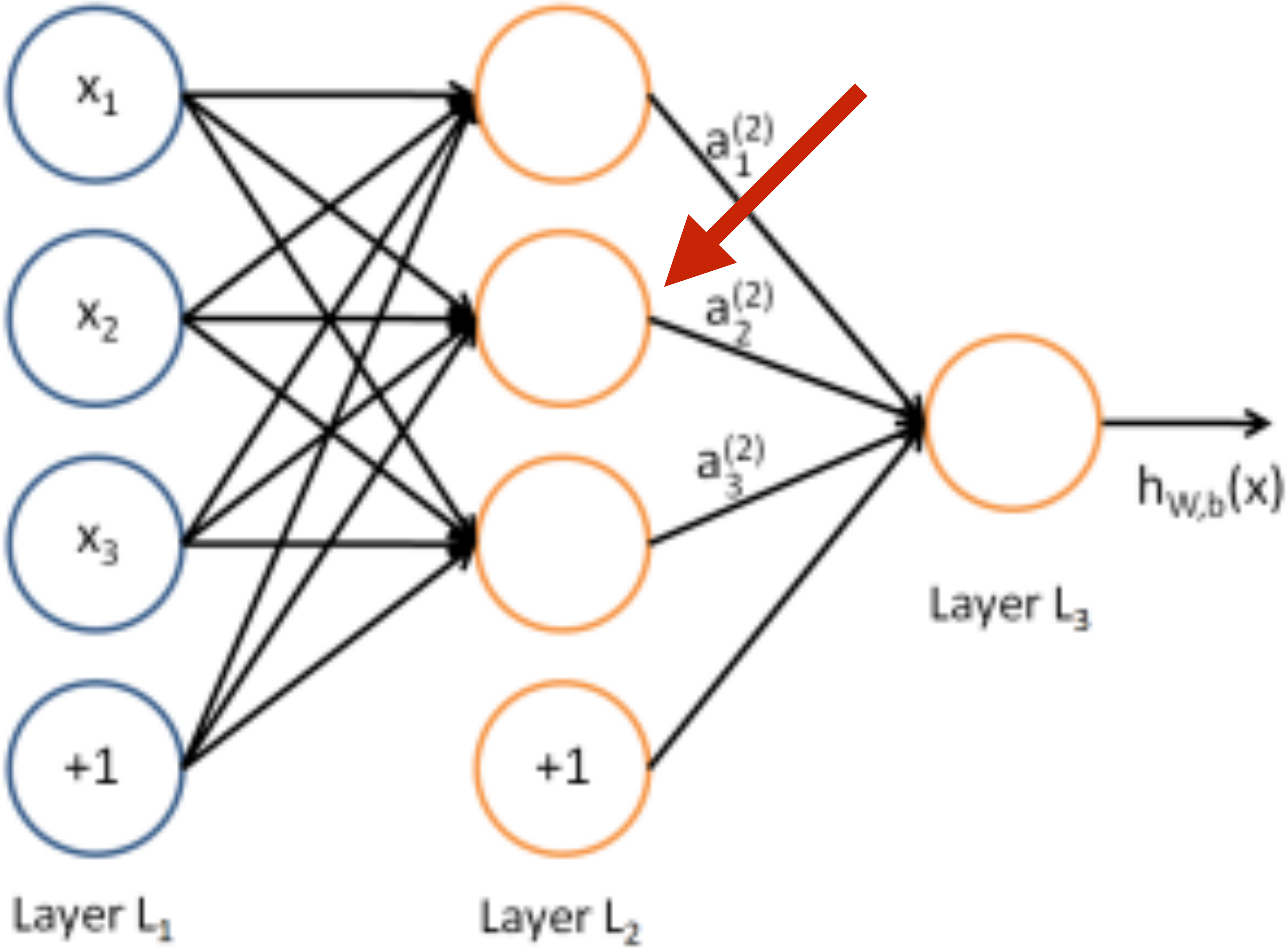
why nonlinearities?



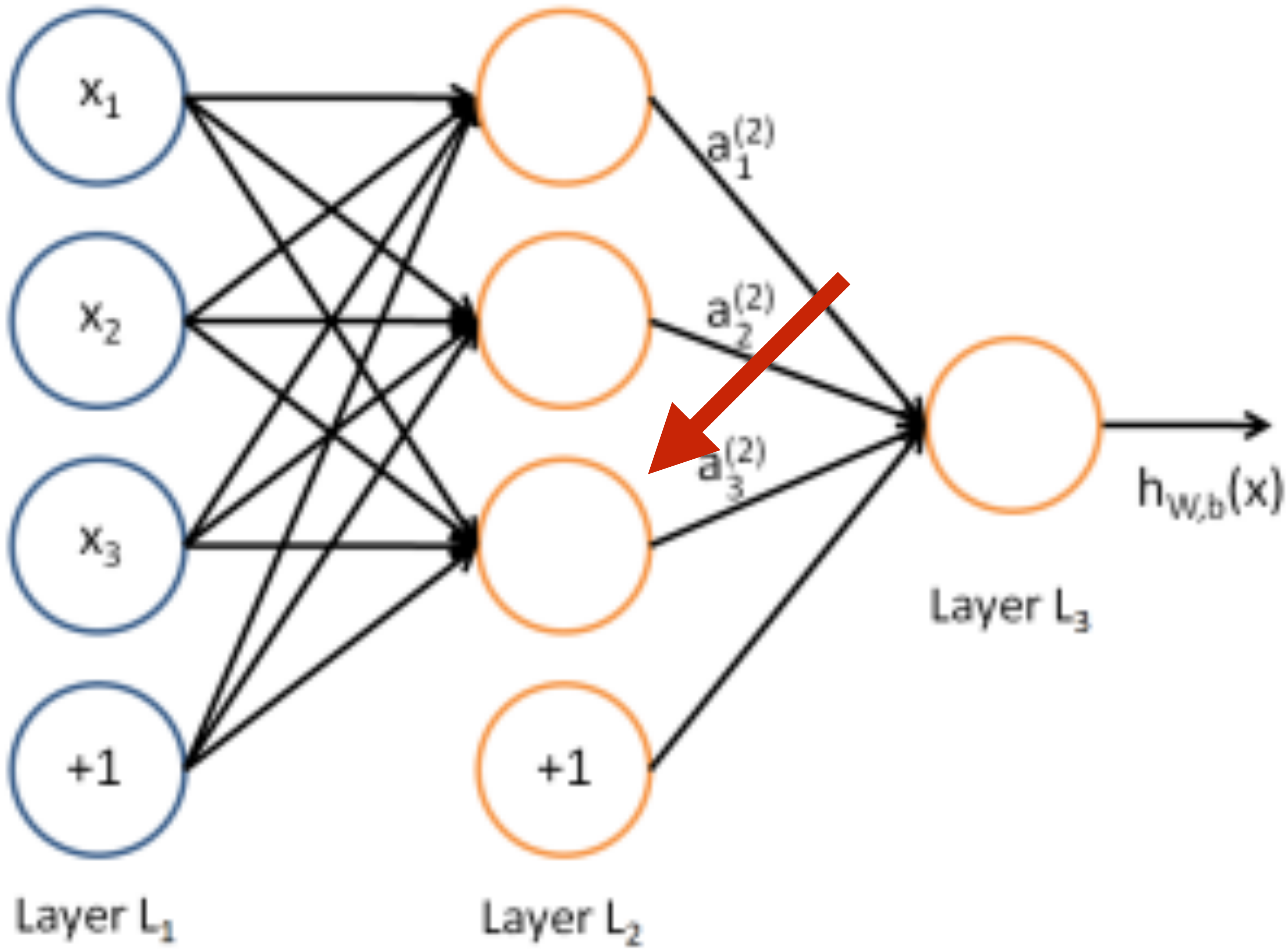




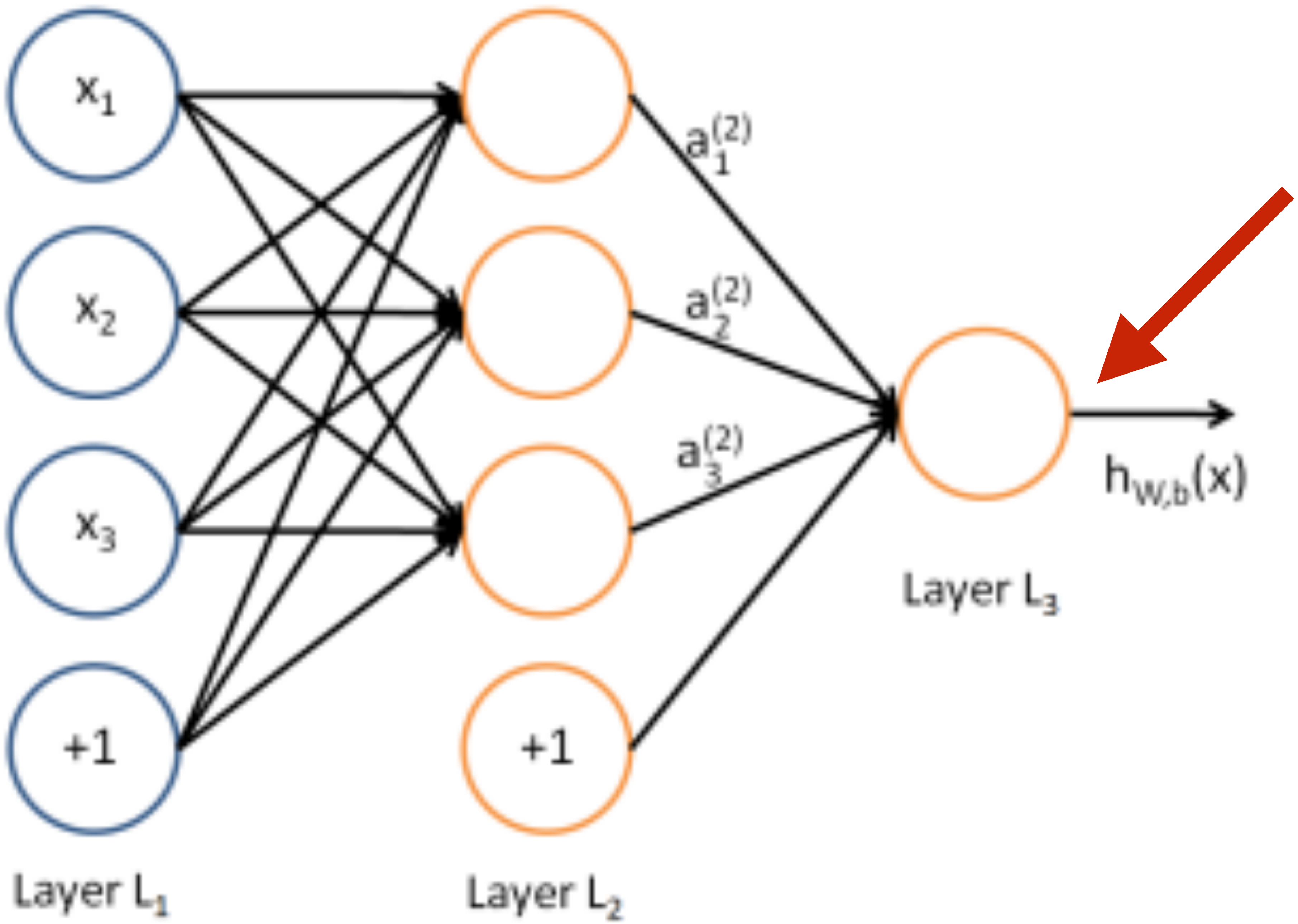
$$a_1^{(2)} = f\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}\right)$$



$$a_2^{(2)} = f\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}\right)$$

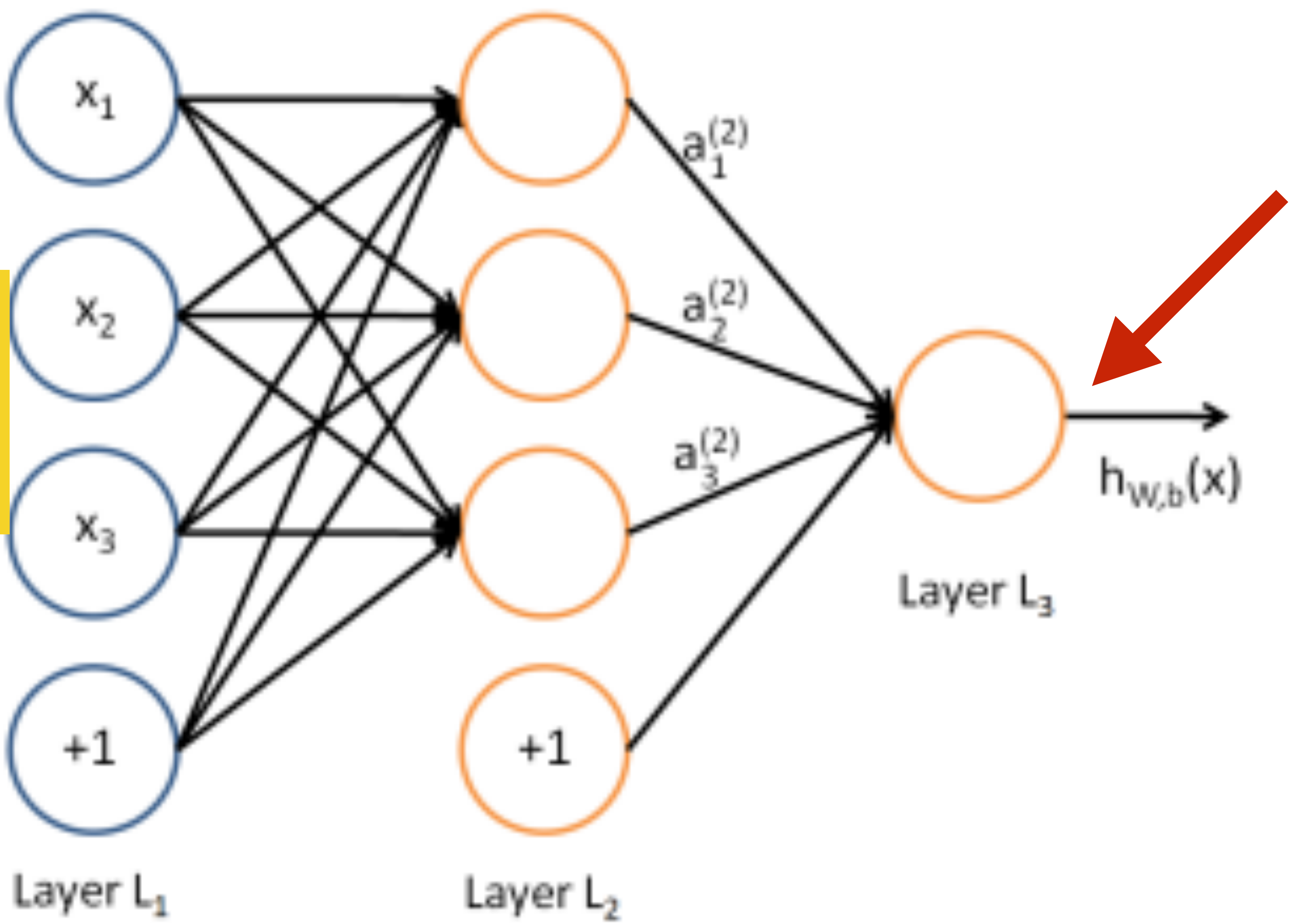


$$a_3^{(2)} = f\left(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)}\right)$$



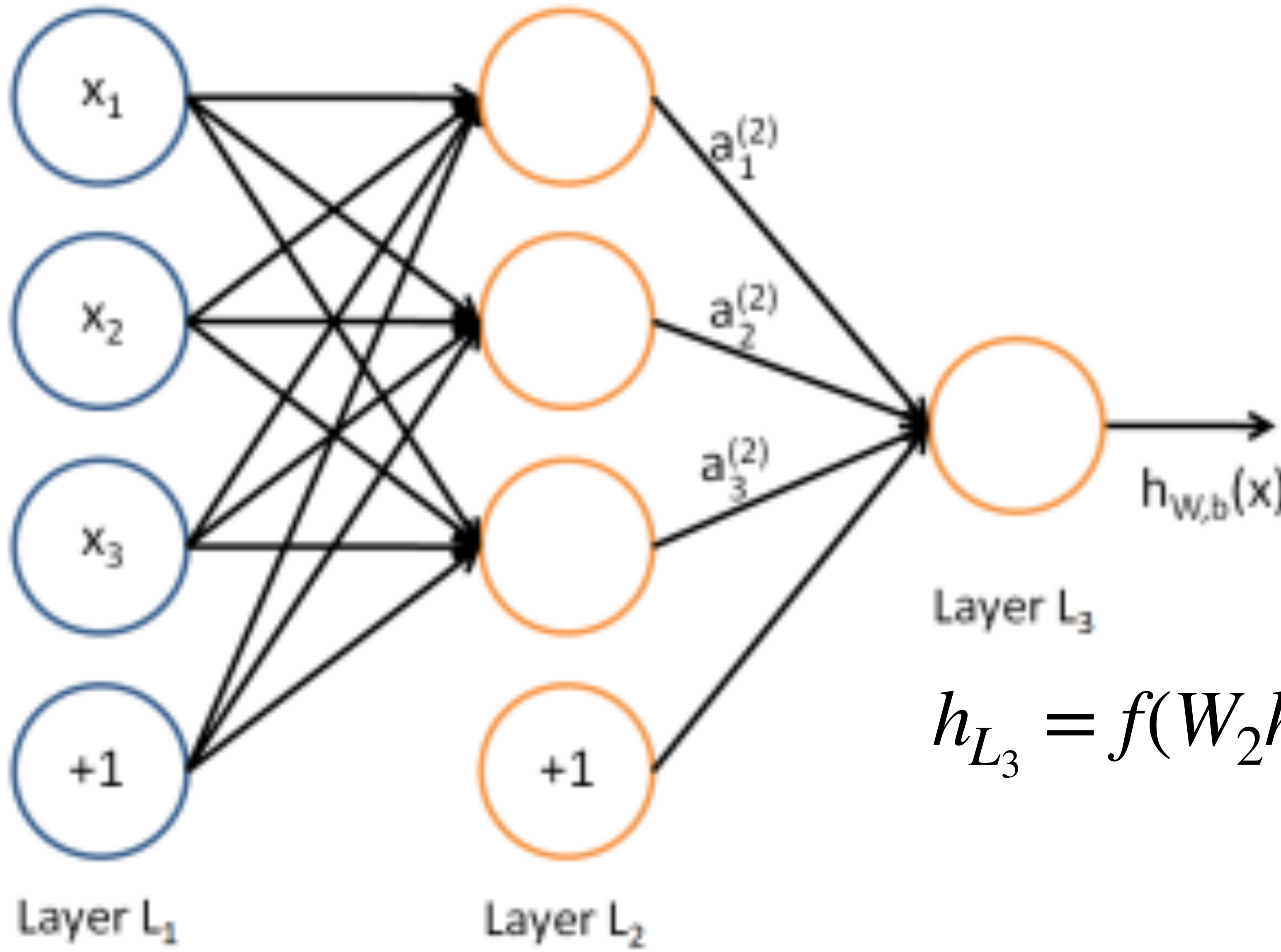
$$h_{W,b}(x) = a_1^{(3)} = f\left(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)}\right)$$

we will be learning the x's and the W's!



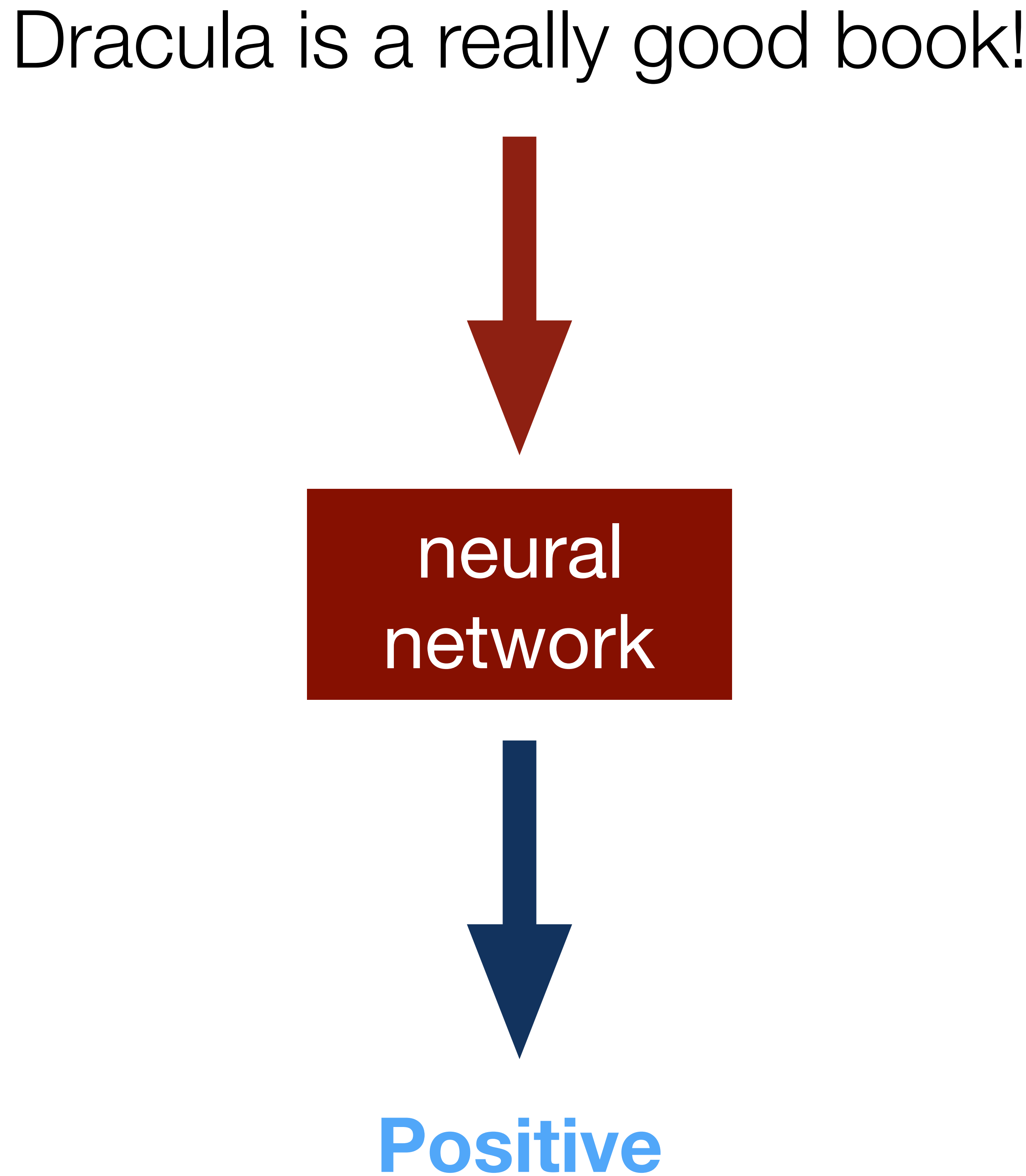
$$h_{W,b}(x) = a_1^{(3)} = f\left(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)}\right)$$

in matrix-vector notation…



$$h_{L_3} = f(W_2 h_{L_2} + b)$$

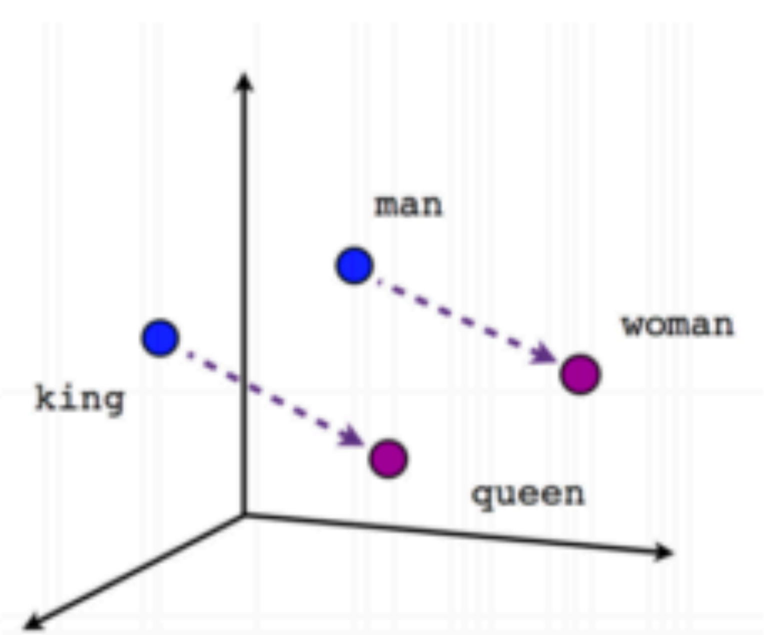
$$h_{L_2} = f(W_1 x + b)$$


Dracula is a really good book!
neural network
Positive

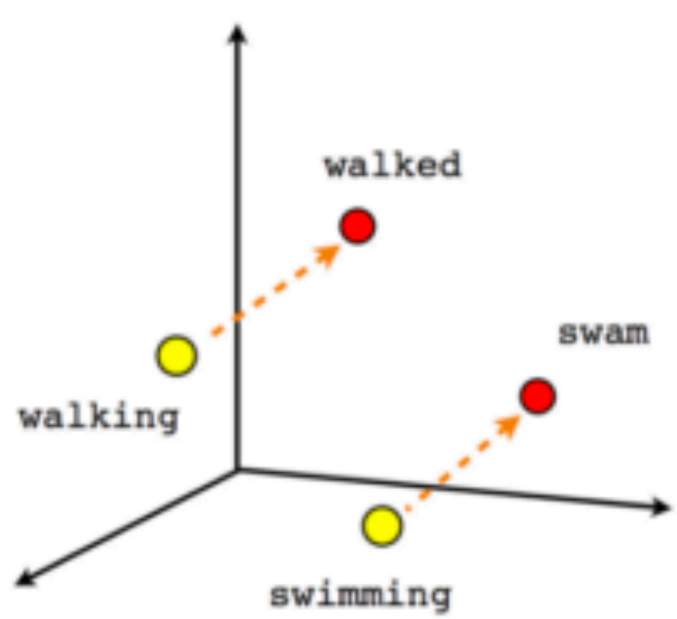
words as basic building blocks

- from last time: represent words with low-dimensional vectors called **embeddings** (Mikolov et al., NIPS 2013)

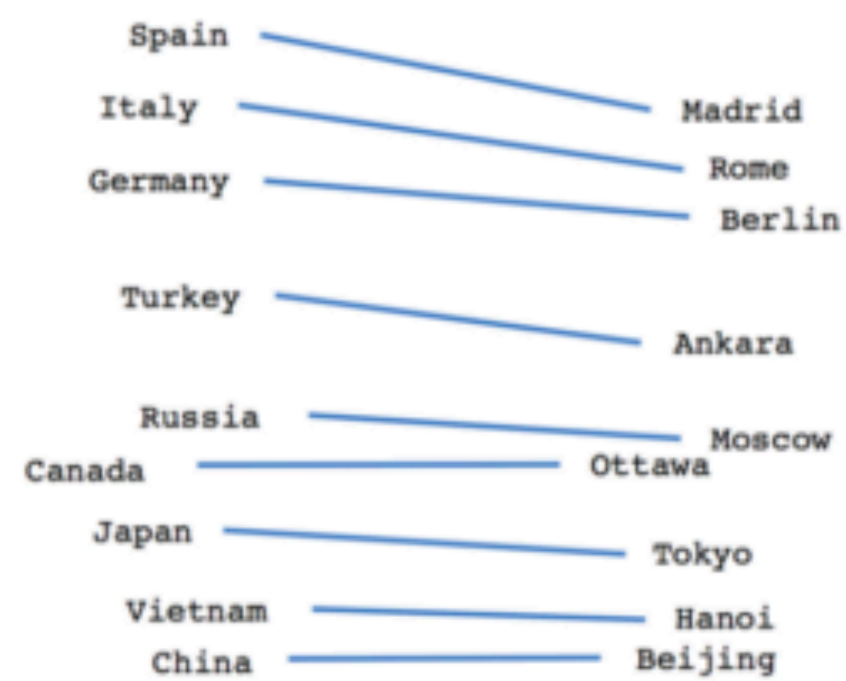
king =
[0.23, 1.3, -0.3, 0.43]



Male-Female



Verb tense

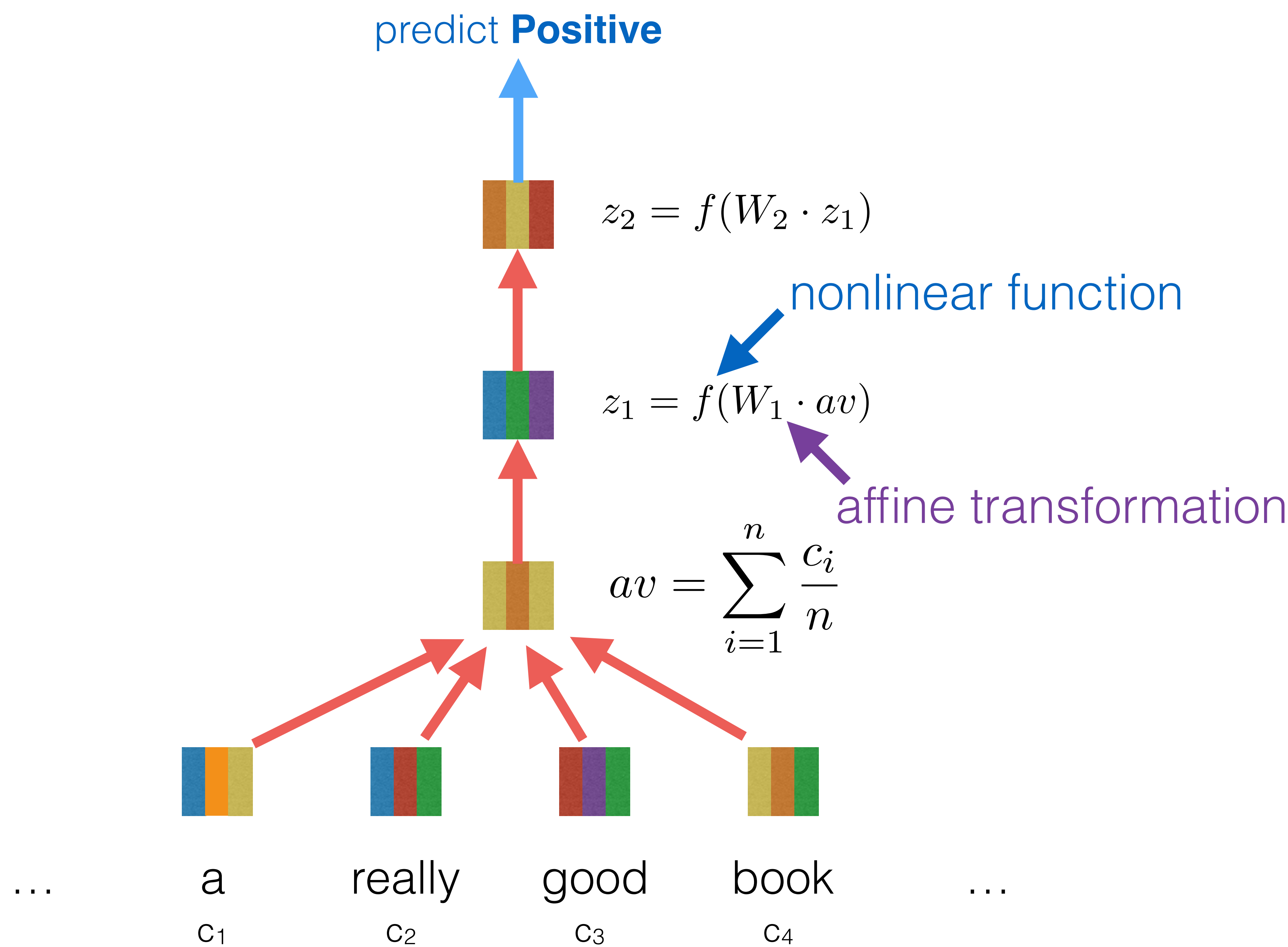


Country-Capital

composing embeddings

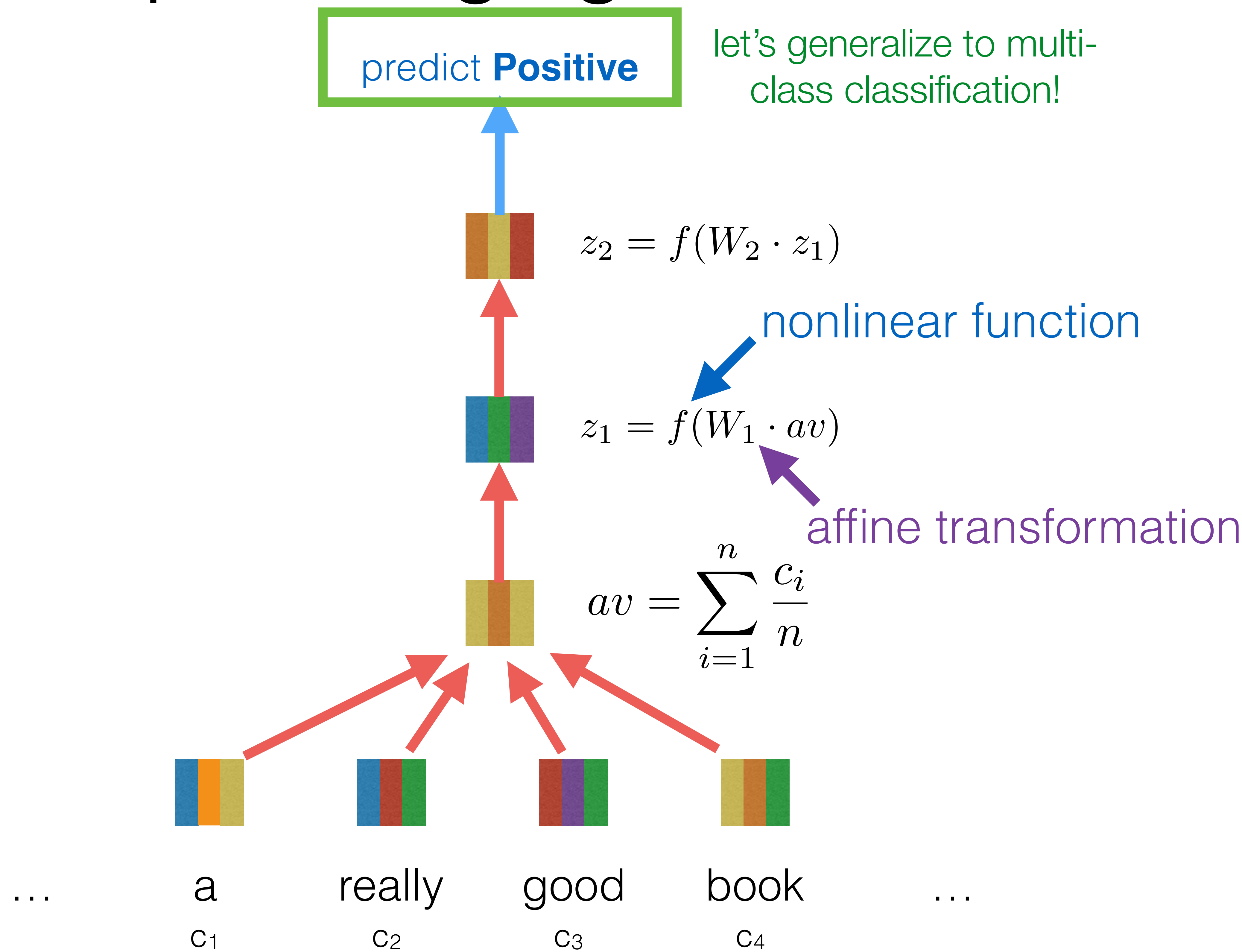
- neural networks **compose** word embeddings into vectors for phrases, sentences, and documents

deep averaging networks



Iyyer et al., ACL 2015 :)

deep averaging networks



softmax function

- let's say I have 3 classes instead of 2 (e.g., positive, neutral, negative)
- i want to compute probabilities for each class. for every class *c*, i have an associated weight vector β_c, and then i compute

$$P(y = c \mid \mathbf{x}) = \frac{e^{\beta_c \mathbf{x}}}{\sum_{k=1}^{3} e^{\beta_k \mathbf{x}}}$$

- sigmoid is a special case of softmax where number of classes = 2

in practice, this computation is done more efficiently…

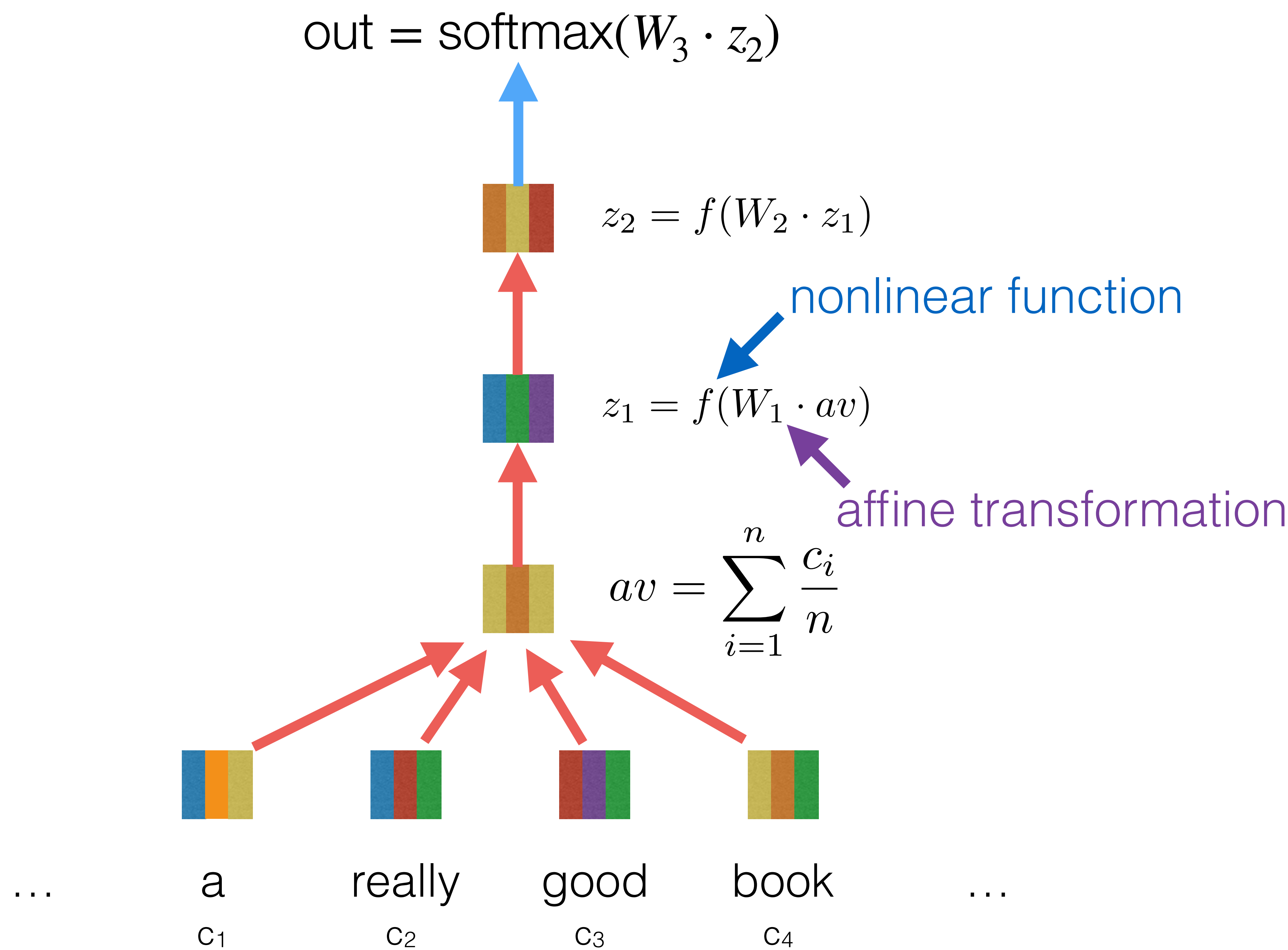
$$\text{softmax}(x) = \frac{e^{x}}{\sum_j e^{x_j}}$$

x is a vector

x_j is dimension *j* of x

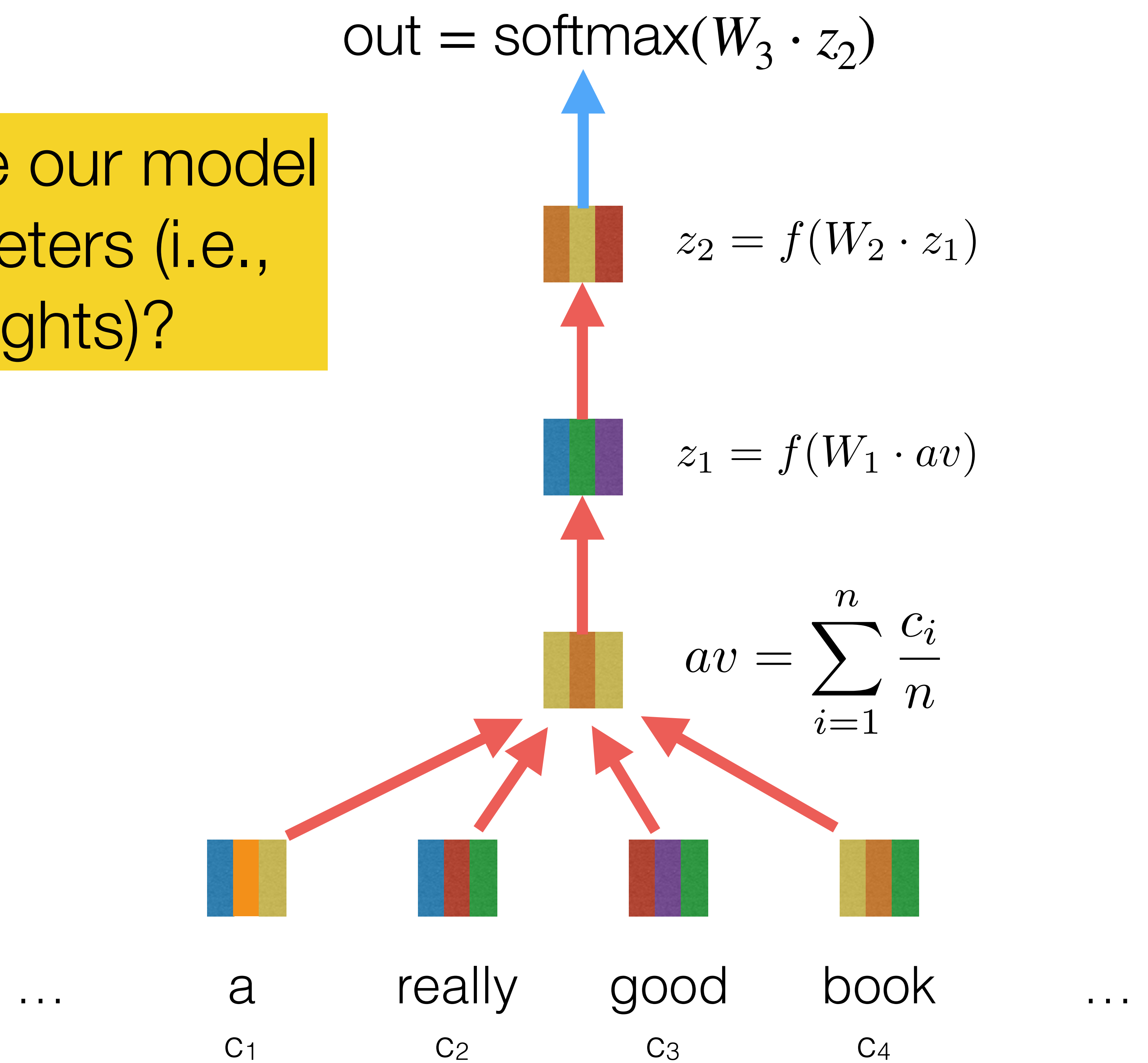
each dimension *j* of the softmaxed output represents the probability of class *j*

deep averaging networks



deep averaging networks

what are our model parameters (i.e., weights)?



Training with softmax and cross-entropy error

- For each training example {x,y}, our objective is to maximize the probability of the correct class y

- Hence, we minimize the negative log probability of that class:

$$\boldsymbol{L} = -\log p(y|x) = -\log\left(\frac{\exp(f_y)}{\sum_{c=1}^{C}\exp(f_c)}\right)$$

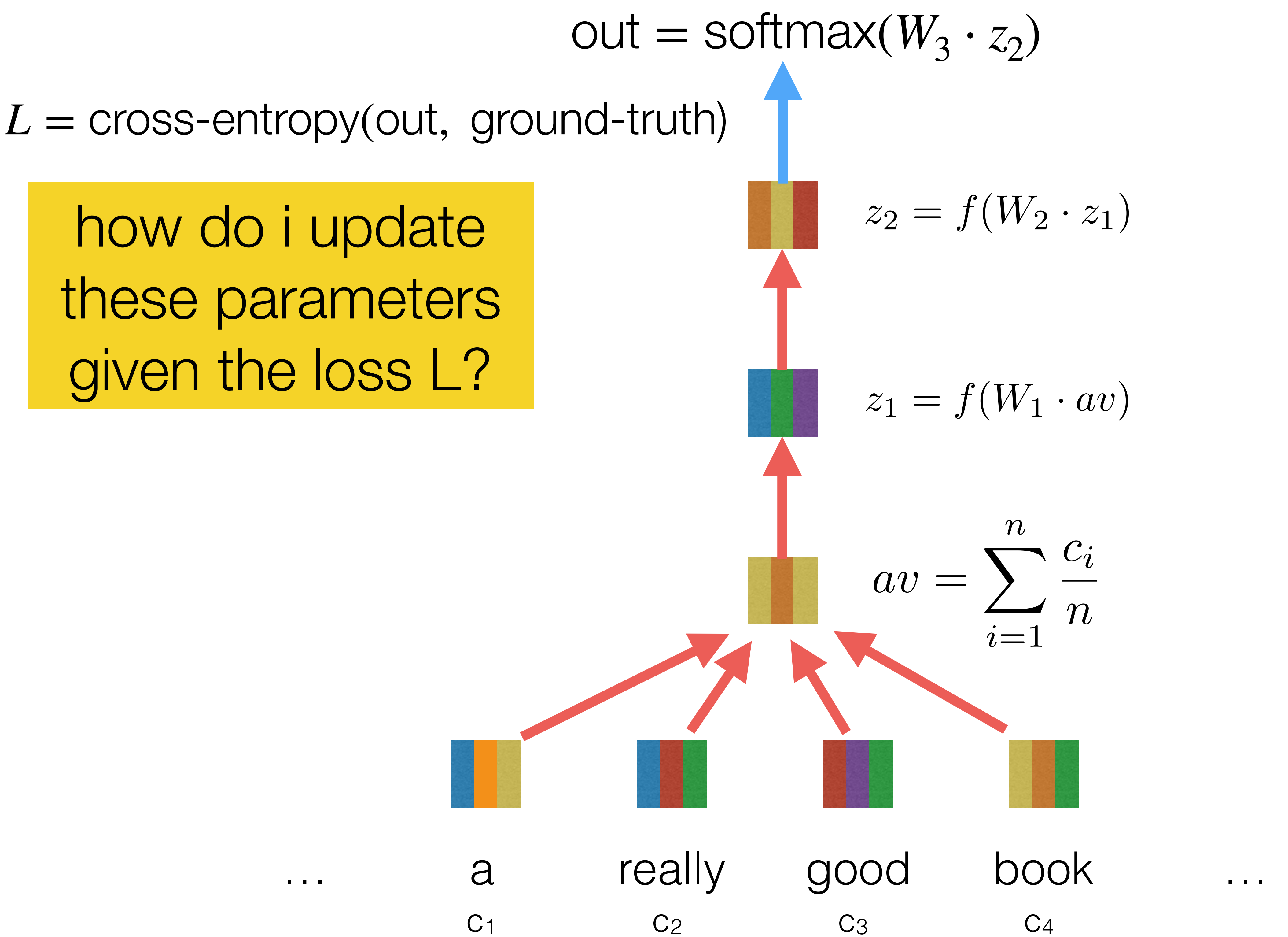
Background: Why “Cross entropy” error

- Assuming a ground truth (or gold or target) probability distribution that is 1 at the right class and 0 everywhere else: p = [0,…,0,1,0,…0] and our computed probability is q, then the cross entropy is:

$$H(p,q) = -\sum_{c=1}^{C} p(c) \log q(c)$$

- **Because of one-hot p, the only term left is the negative log probability of the true class**

deep averaging networks



deep averaging networks

$$\text{out} = \text{softmax}(W_3 \cdot z_2)$$

$$L = \text{cross-entropy}(\text{out, ground-truth})$$

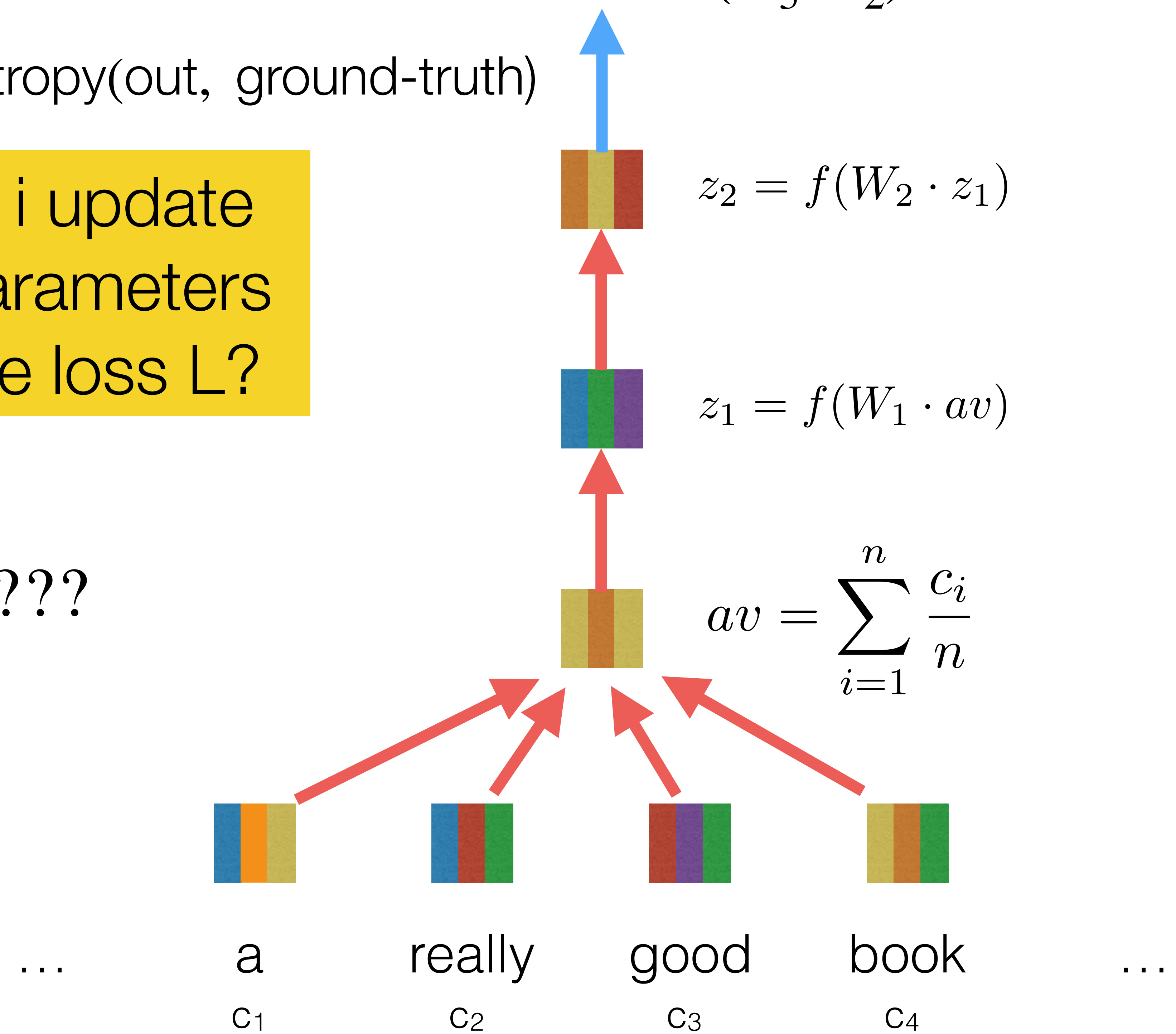
how do i update these parameters given the loss L?

$$z_2 = f(W_2 \cdot z_1)$$

$$z_1 = f(W_1 \cdot av)$$

$$av = \sum_{i=1}^{n} \frac{c_i}{n}$$

$$\frac{\partial L}{\partial c_i} = ???$$

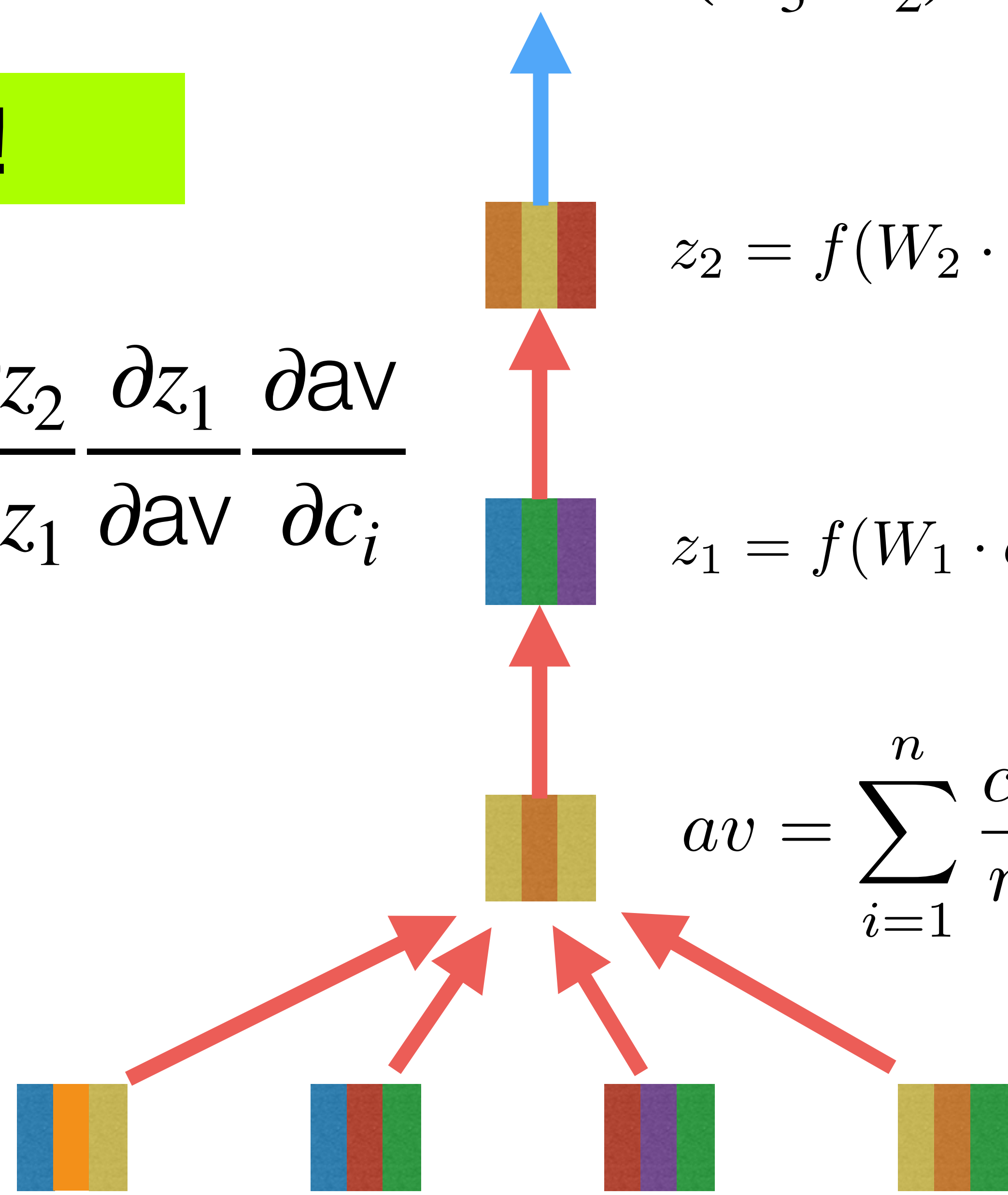


deep averaging networks

$$\text{out} = \text{softmax}(W_3 \cdot z_2)$$

chain rule!!!

$$\frac{\partial L}{\partial c_i} = \frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2} \frac{\partial z_2}{\partial z_1} \frac{\partial z_1}{\partial \text{av}} \frac{\partial \text{av}}{\partial c_i}$$



$$z_2 = f(W_2 \cdot z_1)$$

$$z_1 = f(W_1 \cdot av)$$

$$av = \sum_{i=1}^{n} \frac{c_i}{n}$$

… a really good book …

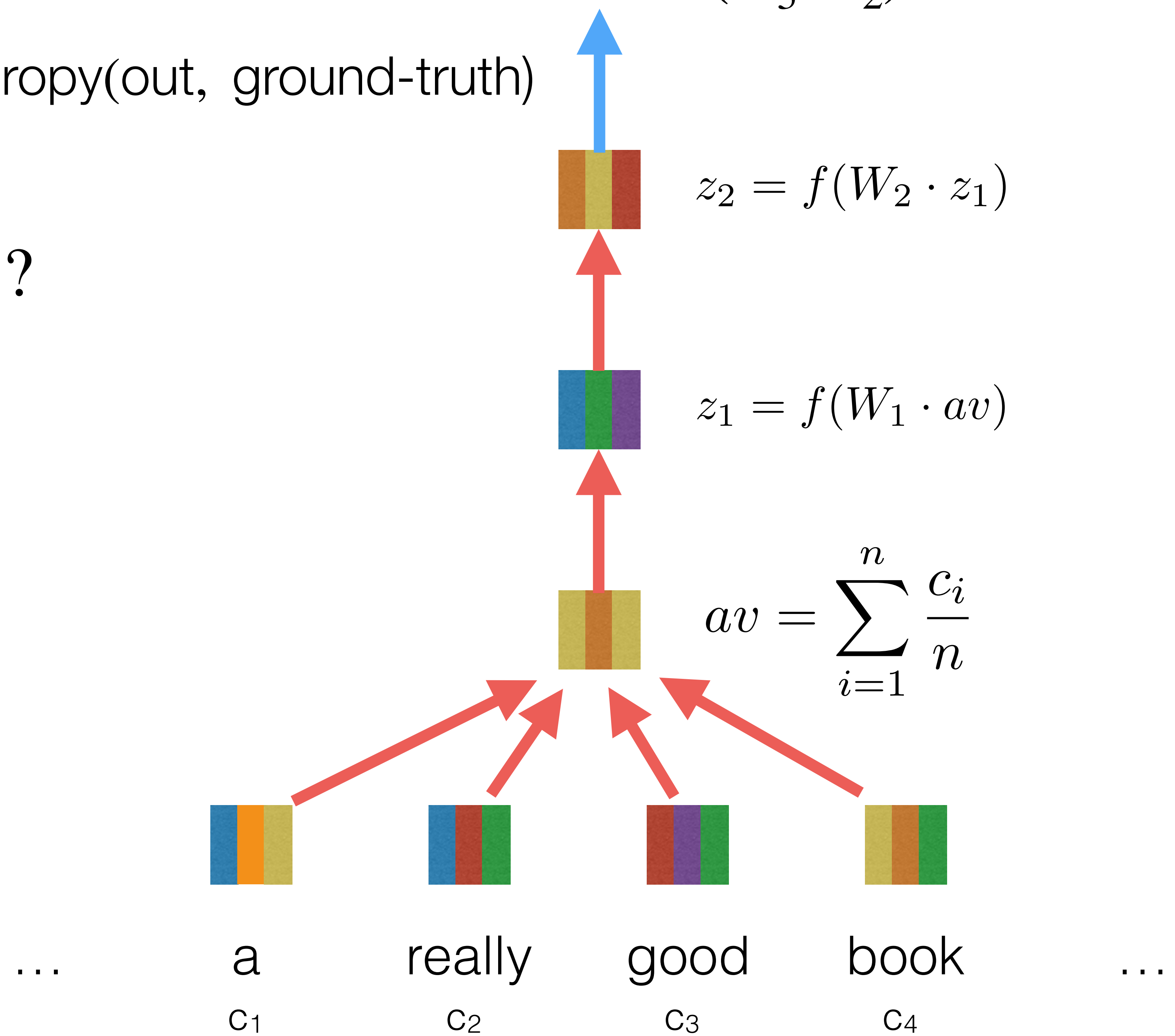
c_1 c_2 c_3 c_4

deep averaging networks

out = softmax($W_3 \cdot z_2$)

L = cross-entropy(out, ground-truth)

$$\frac{\partial L}{\partial W_2} = ???$$

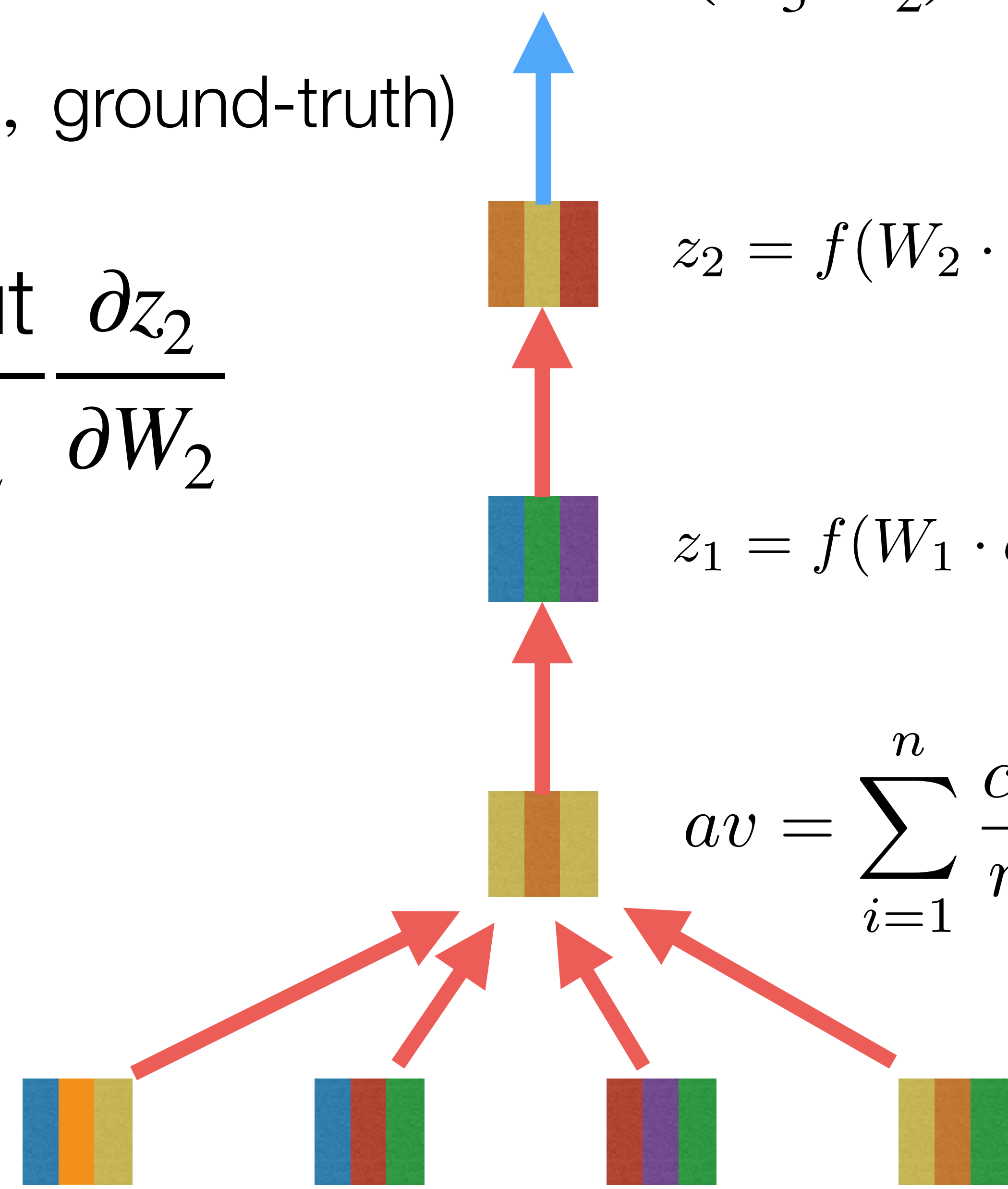


deep averaging networks

$$\text{out} = \text{softmax}(W_3 \cdot z_2)$$

$$L = \text{cross-entropy}(\text{out},\ \text{ground-truth})$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2} \frac{\partial z_2}{\partial W_2}$$



$$z_2 = f(W_2 \cdot z_1)$$

$$z_1 = f(W_1 \cdot av)$$

$$av = \sum_{i=1}^{n} \frac{c_i}{n}$$

… a really good book …

c_1 c_2 c_3 c_4

backpropagation

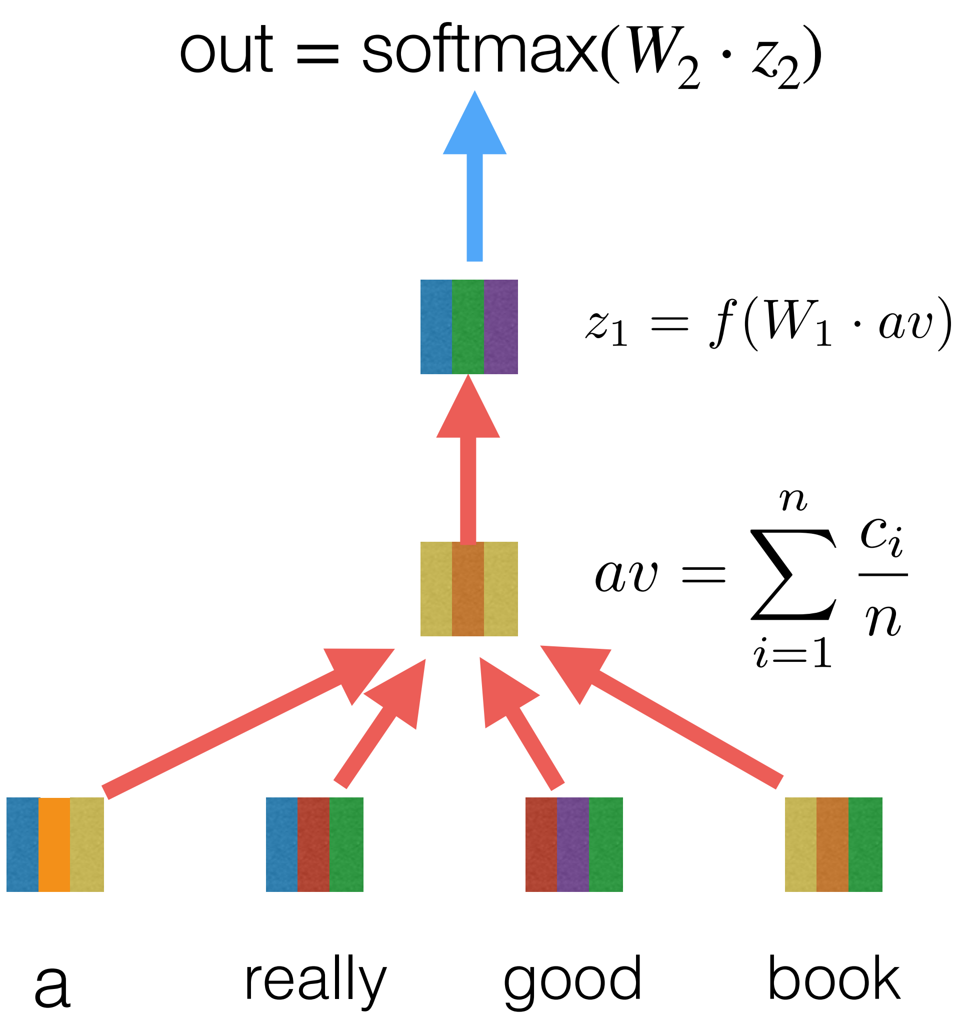
- use the chain rule to compute partial derivatives w/ respect to each parameter
- trick: re-use derivatives computed for higher layers to compute derivatives for lower layers!

$$\frac{\partial L}{\partial c_i} = \frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2} \frac{\partial z_2}{\partial z_1} \frac{\partial z_1}{\partial \text{av}} \frac{\partial \text{av}}{\partial c_i}$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2} \frac{\partial z_2}{\partial W_2}$$

Rumelhart et al., 1986

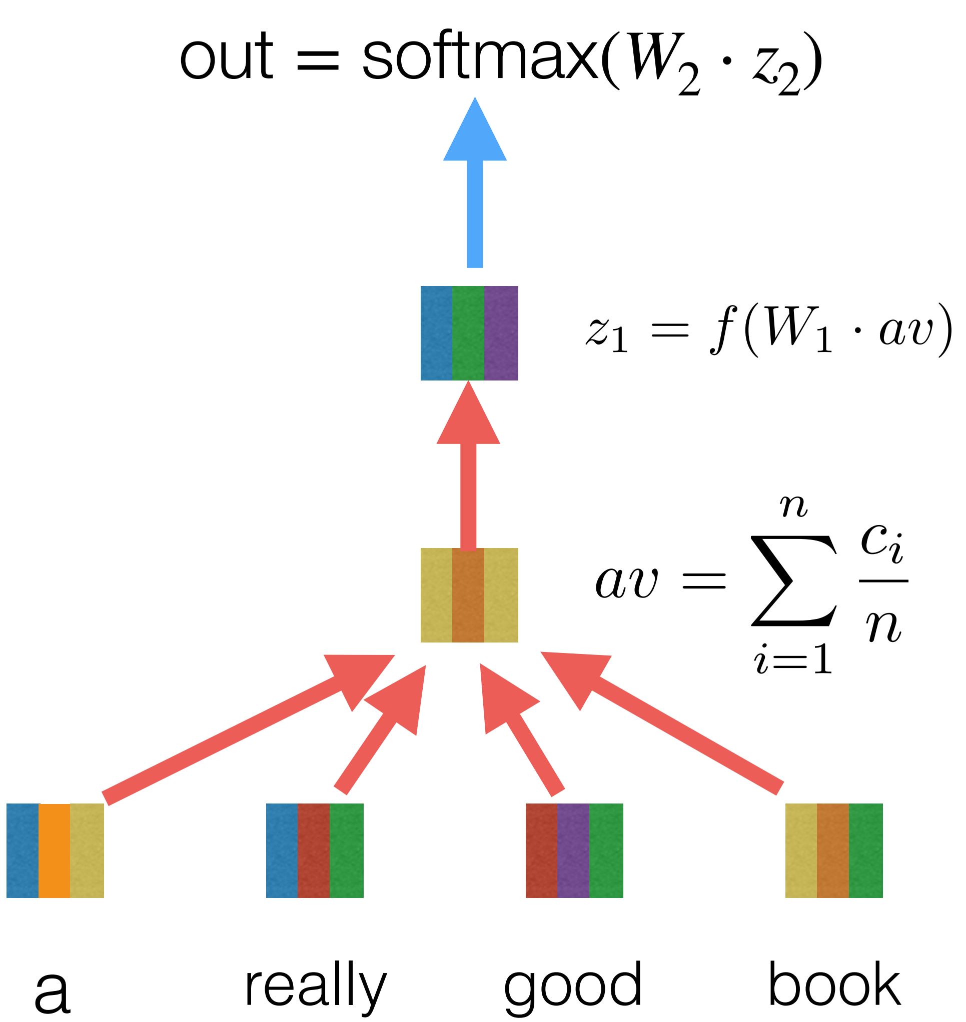
deep learning frameworks make building NNs super easy!



set up the network

```
def __init__(self, n_classes, vocab_size, emb_dim=300,
             n_hidden_units=300):
    super(DanModel, self).__init__()
    self.n_classes = n_classes
    self.vocab_size = vocab_size
    self.emb_dim = emb_dim
    self.n_hidden_units = n_hidden_units
    self.embeddings = nn.Embedding(self.vocab_size,
                                   self.emb_dim)
    self.classifier = nn.Sequential(
           nn.Linear(self.n_hidden_units,
                     self.n_hidden_units),
           nn.ReLU(),
           nn.Linear(self.n_hidden_units,
                     self.n_classes))
    self._softmax = nn.Softmax()
```

deep learning frameworks make building NNs super easy!

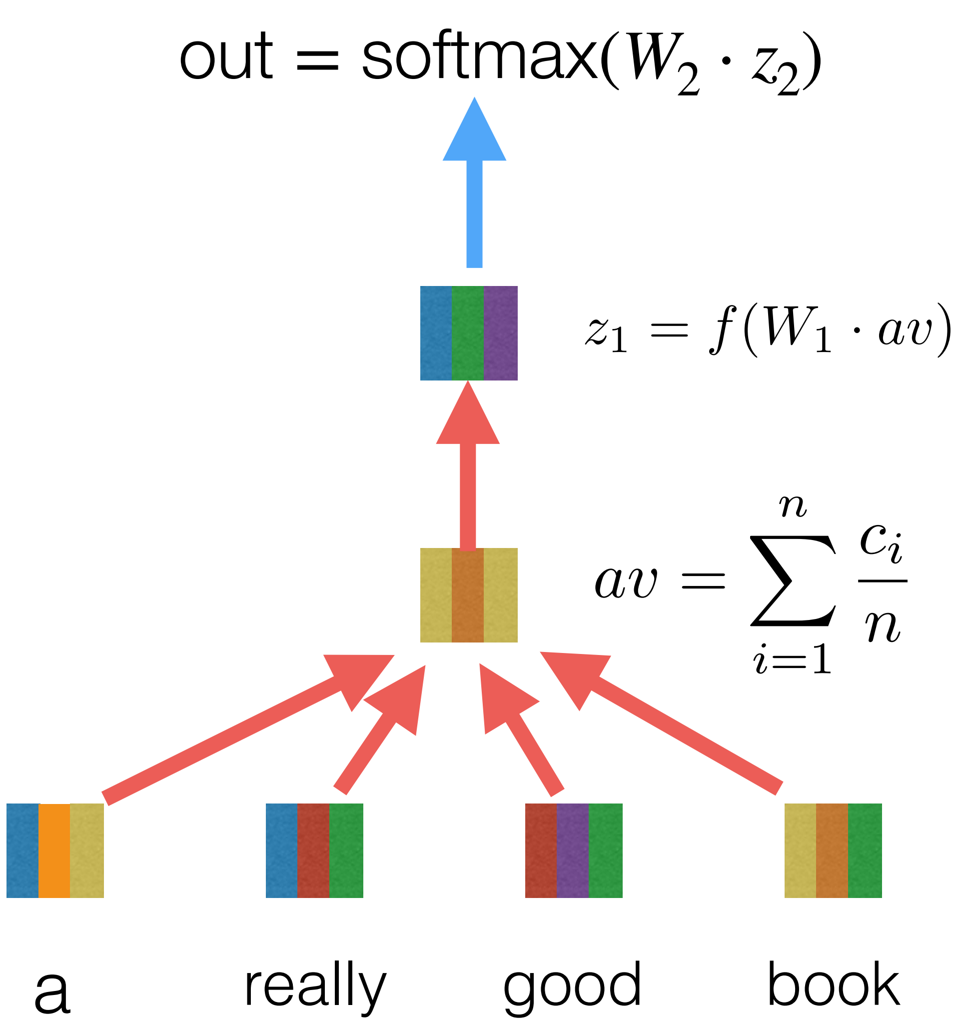


<u>do a forward pass to compute prediction</u>

```
def forward(self, batch, probs=False):
    text = batch['text']['tokens']
    length = batch['length']
    text_embed = self._word_embeddings(text)
    # Take the mean embedding. Since padding results
    # in zeros its safe to sum and divide by length
    encoded = text_embed.sum(1)
    encoded /= lengths.view(text_embed.size(0), -1)

    # Compute the network score predictions
    logits = self.classifier(encoded)
    if probs:
        return self._softmax(logits)
    else:
        return logits
```

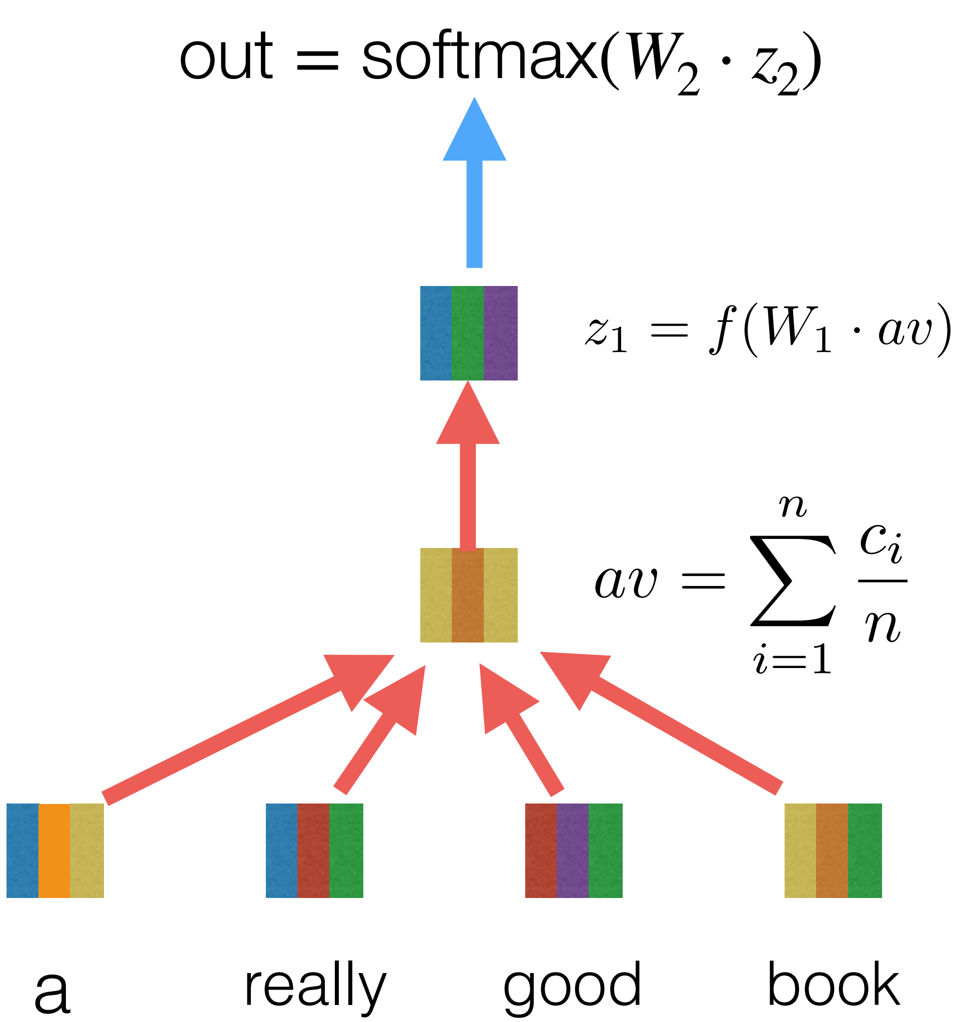
deep learning frameworks make building NNs super easy!



<u>do a backward pass to update weights</u>

```
def _run_epoch(self, batch_iter, train=True):
    self._model.train()
    for batch in batch_iter:
        model.zero_grad()
        out = model(batches)
        batch_loss = criterion(out,
                               batch['label'])
        batch_loss.backward()
        self.optimizer.step()
```

deep learning frameworks make building NNs super easy!



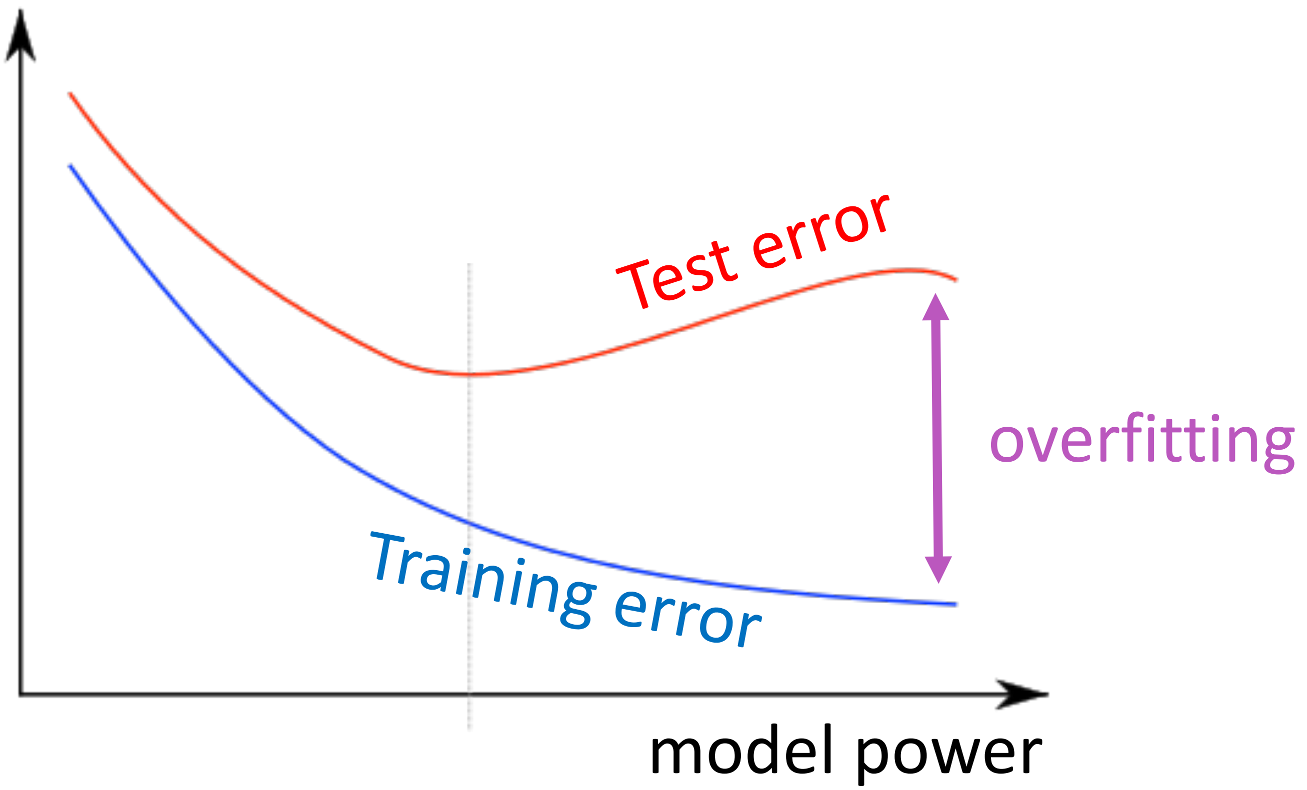
<u>do a backward pass to update weights</u>

```
def _run_epoch(self, batch_iter, train=True):
    self._model.train()
    for batch in batch_iter:
        model.zero_grad()
        out = model(batches)
        batch_loss = criterion(out,
                               batch[’label’])
        batch_loss.backward()
        self.optimizer.step()
```

that’s it! no need to compute gradients by hand!
however, you will have to do this in HW2 :(

Regularization

- Regularization prevents overfitting when we have a lot of features (or later a very powerful/deep model,++)



L2 regularization

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) \boxed{+ \lambda \sum_k \theta_k^2}$$

θ represents all of the model's parameters!

L2 regularization

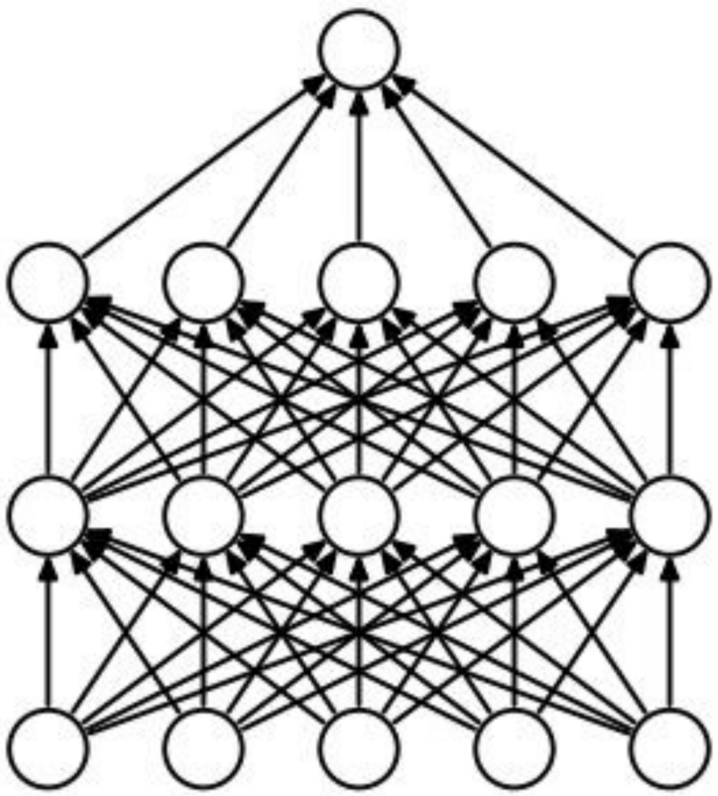
$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) \boxed{+ \lambda \sum_k \theta_k^2}$$

θ represents all of the model's parameters!

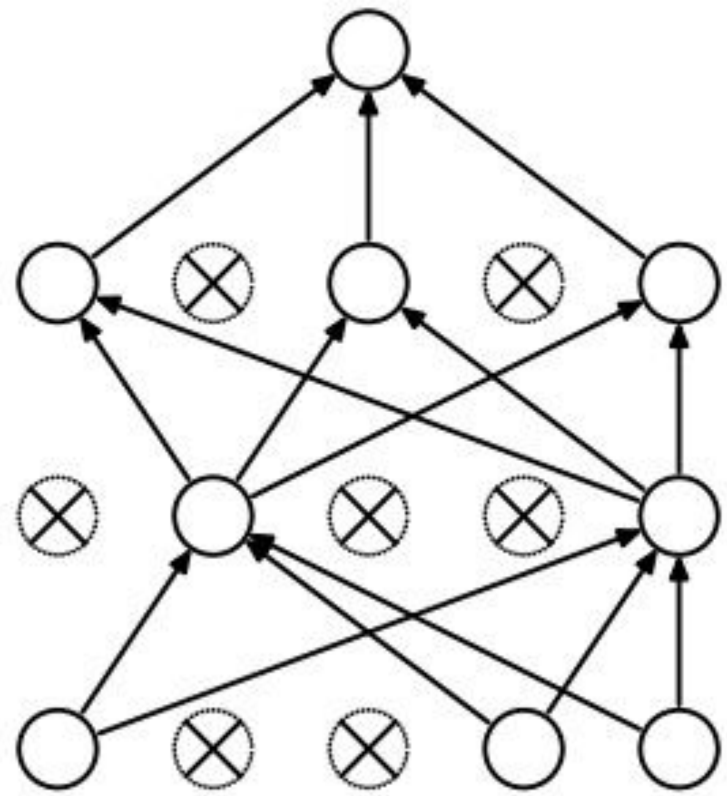
penalizing their norm leads to smaller weights >
we are constraining the parameter space >
we are putting a prior on our model

dropout (for neural networks)

randomly set p% of neurons to 0 in the forward pass



(a) Standard Neural Net

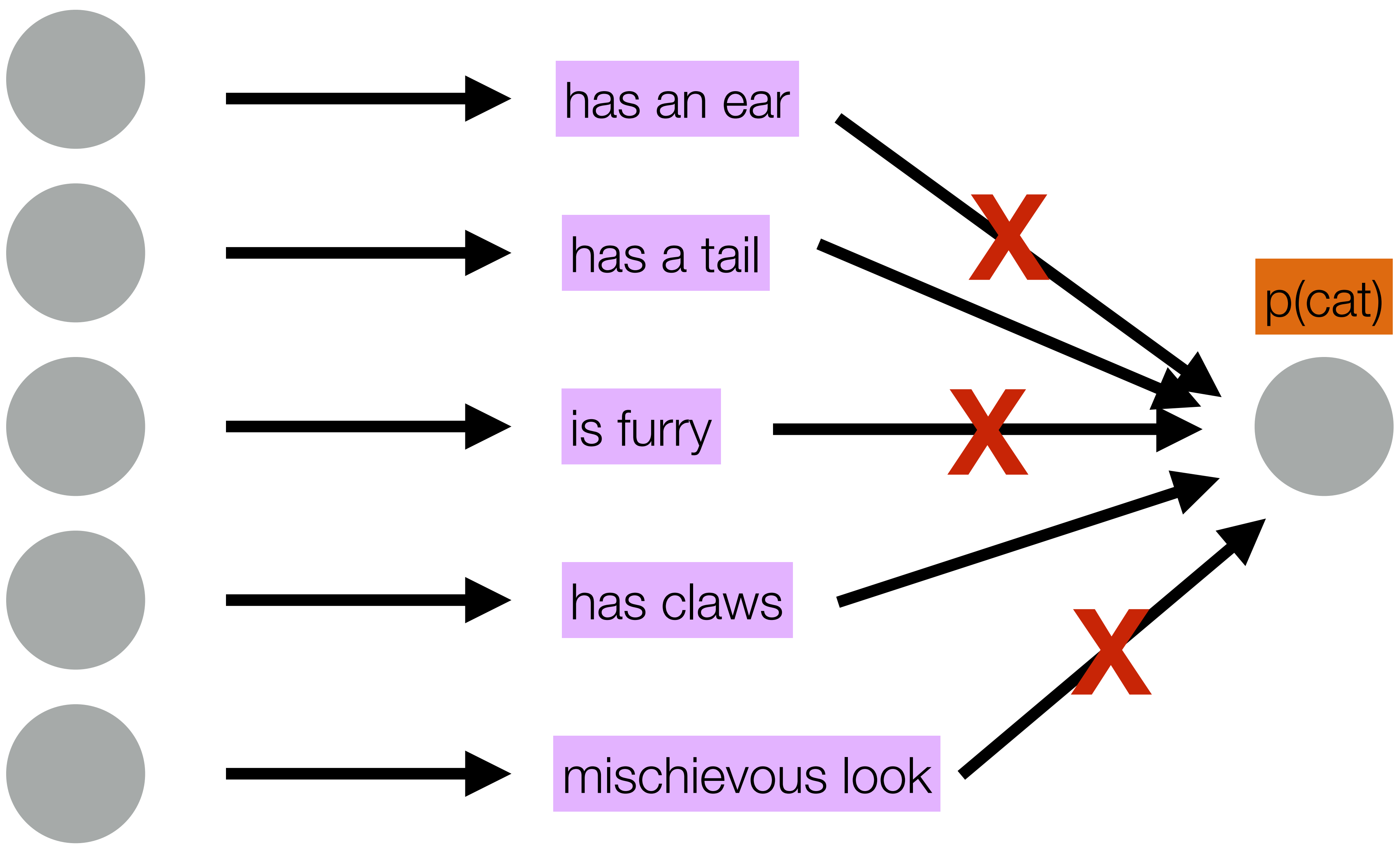


(b) After applying dropout.

[Srivastava et al., 2014]

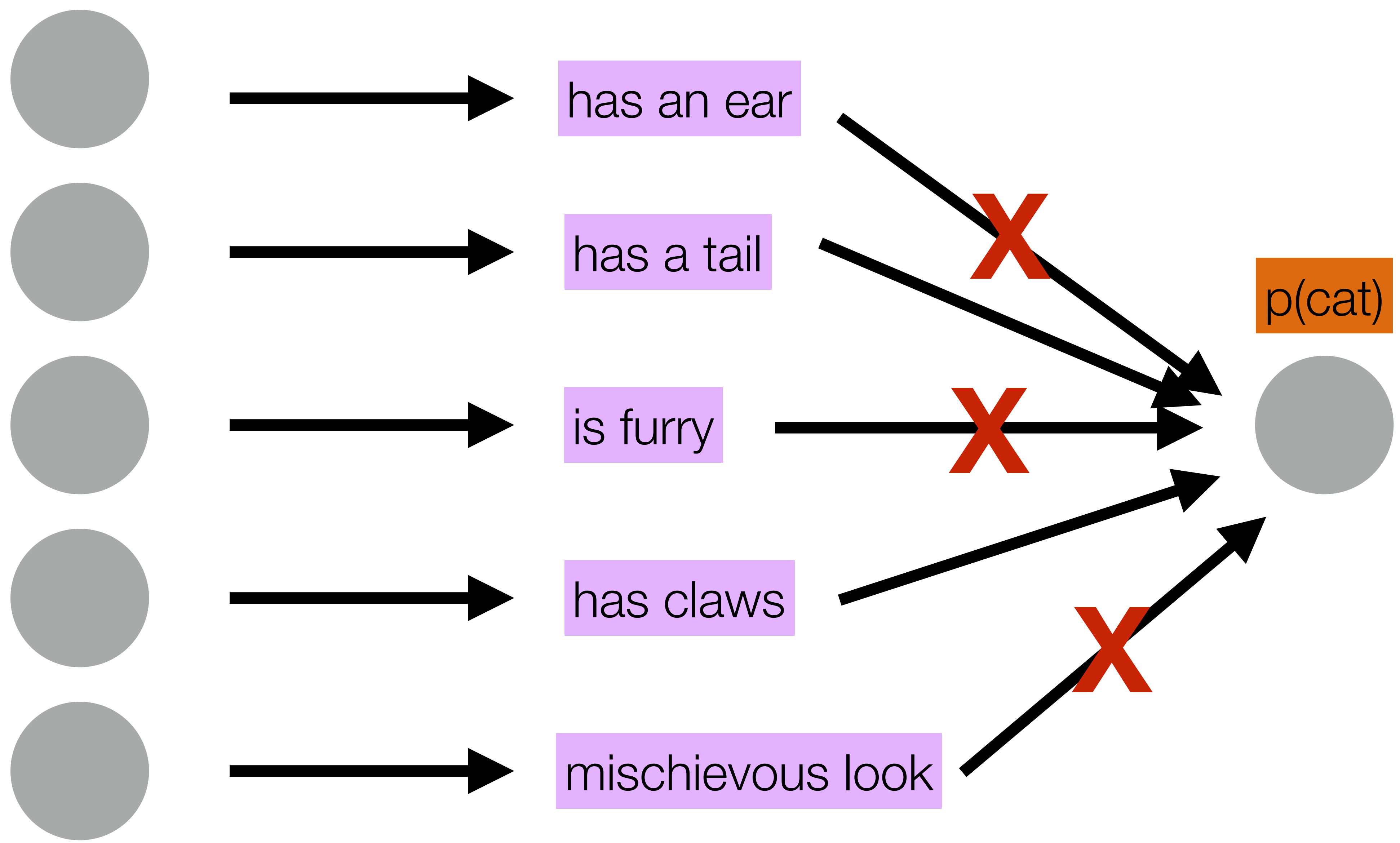
why does this make sense?

randomly set *p*% of neurons to 0 in the forward pass



why does this make sense?

randomly set *p*% of neurons to 0 in the forward pass



network can't just rely on one neuron!

exercise!