Neural networks in NLP

CS 490A, Fall 2021

https://people.cs.umass.edu/~brenocon/cs490a_f21/

Laure Thompson and Brendan O'Connor
College of Information and Computer Sciences
University of Massachusetts Amherst

some slides adapted from Mohit Iyyer, Jordan Boyd-Graber, Richard Socher, Jacob Eisenstein (INLP textbook)

Announcements

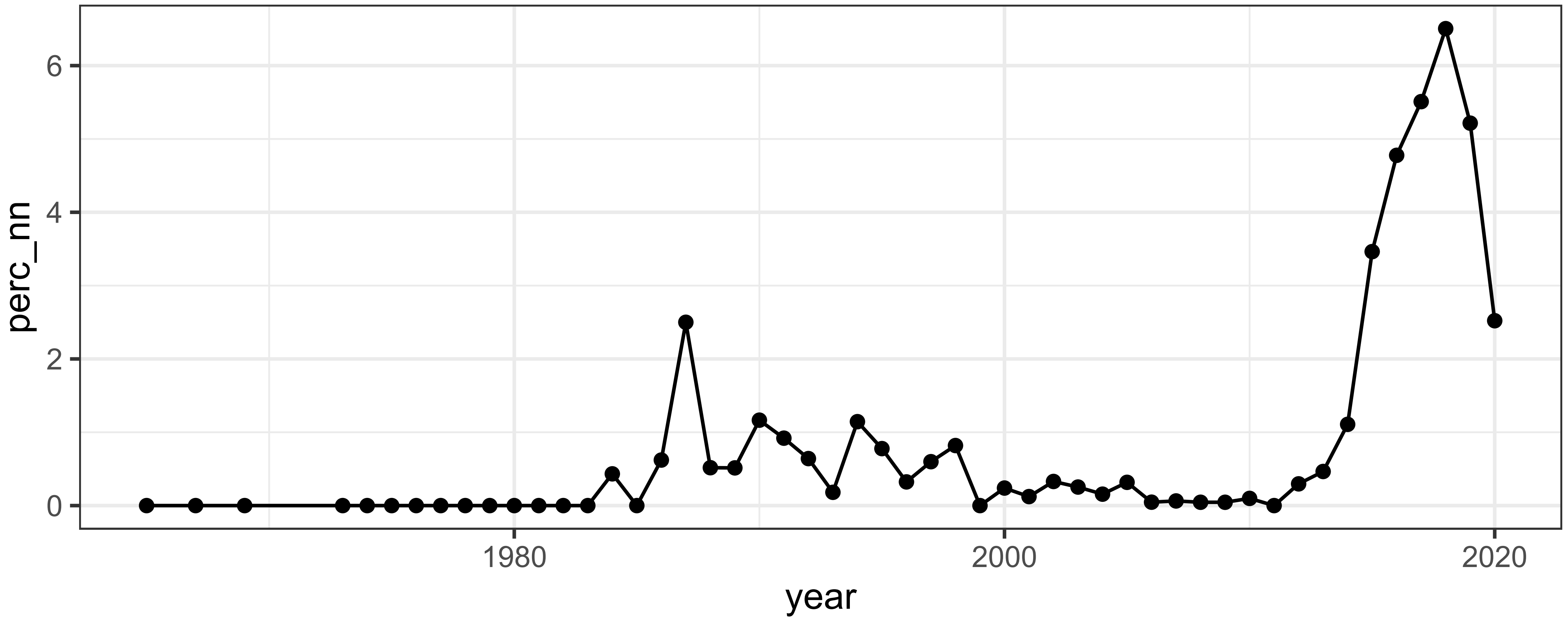
- 10/28 embedding demo: download links updated
  - To come: Exercise 10 to just play around with embeddings in the same way
- Thank you for scheduling your MANDATORY :) project meeting! Don't lose points on this!
  - Project proposals makeups due tomorrow
- Midterm:
  - In class, next Tuesday
  - For accommodations, alternative proctoring, or scheduling issues, let us know ASAP
  - Midterm review questions to be posted (show)

Neural Networks in NLP

- Motivations:
    - Word sparsity => denser word representations
    - Nonlinearity
- Models
    - BoE / Deep Averaging
- Learning
    - Backprop
    - Dropout

The Second Wave: NNs in NLP

- % of ~ACL paper titles with “connectionist/connectionism”, “parallel distributed”, “neural network”, “deep learning”
    - https://www.aclweb.org/anthology/



NN Text Classification

- Goals:
  - Avoid feature engineering
  - Generalize beyond individual words
  - Compose meaning from context
- Now: we have several general model architectures (+pretraining) that work well for many different datasets (and tasks!)
- Less clear: why they work and what they're doing

Word sparsity

- Alternate view of Bag-of-Words classifiers: every word has a “one-hot” representation.
    - Represent each word as a vector of zeros with a single 1 identifying the index of the word
- Doc BOW **x** = average of all words’ vectors

vocabulary
i
hate
love
the
movie
film

movie = <0, 0, 0, 0, 1, 0>

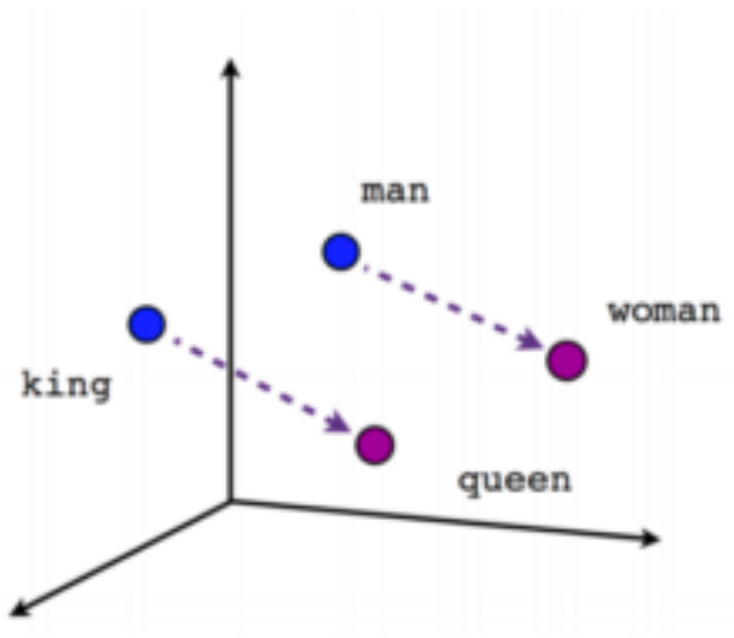
film = <0, 0, 0, 0, 0, 1>

what are the issues of representing a word this way?

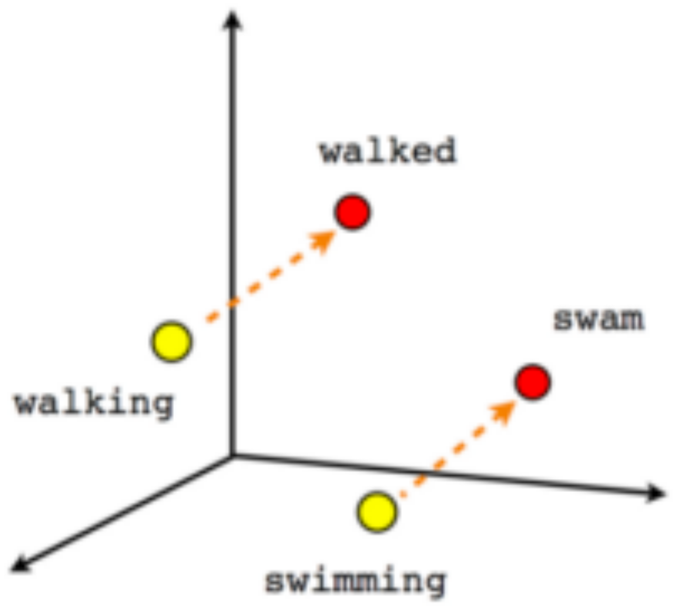
Word embeddings

- Today: word embeddings are the first “lookup” layer in an NN. Every word in vocabulary has a vector — these are model parameters.

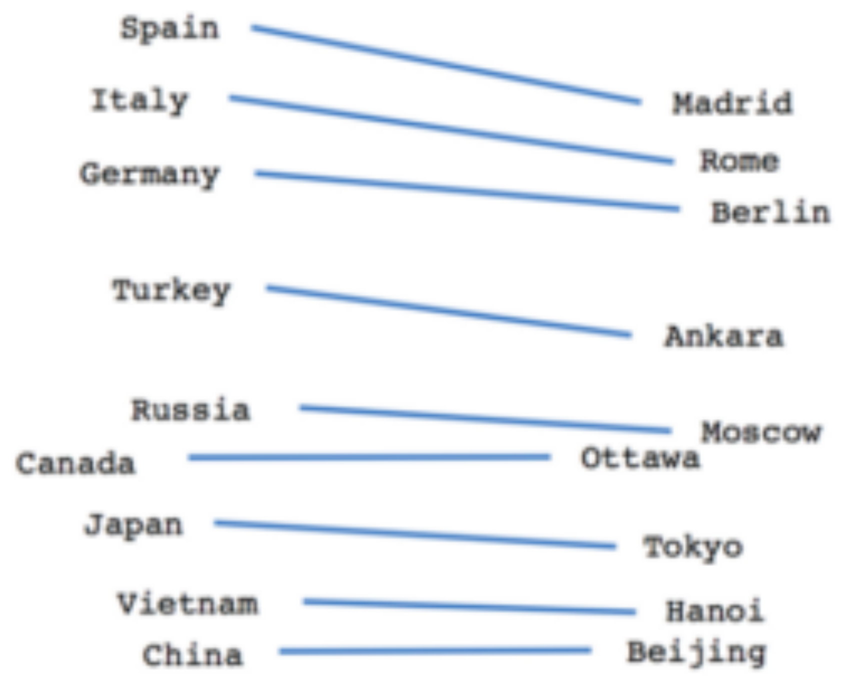
king = [0.23, 1.3, -0.3, 0.43]



Male-Female



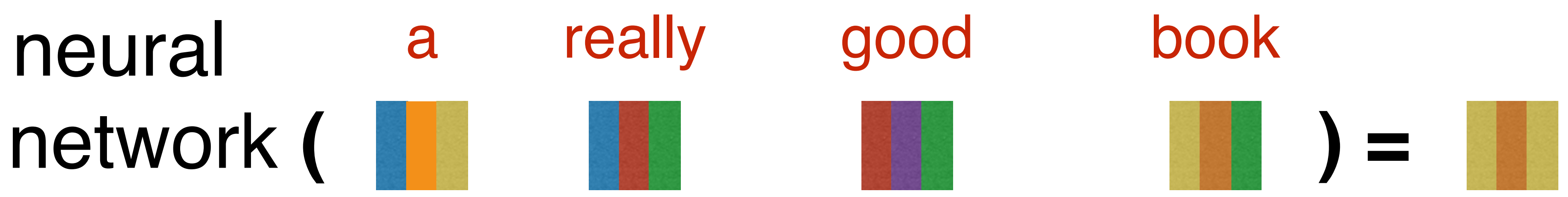
Verb tense



Country-Capital

composing embeddings

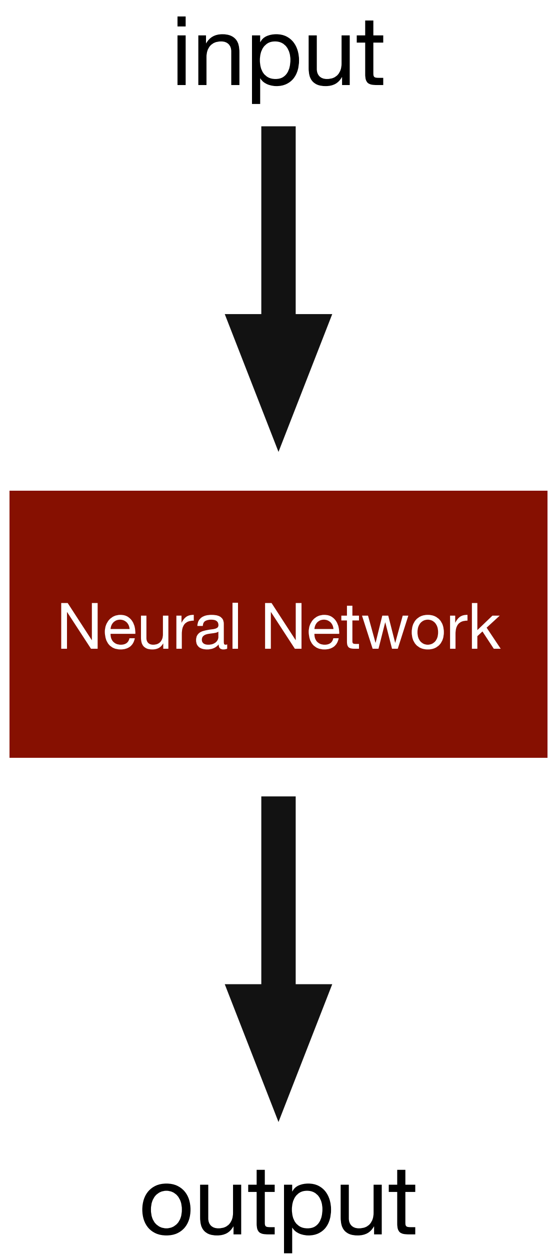
- neural networks **compose** word embeddings into vectors for phrases, sentences, and documents



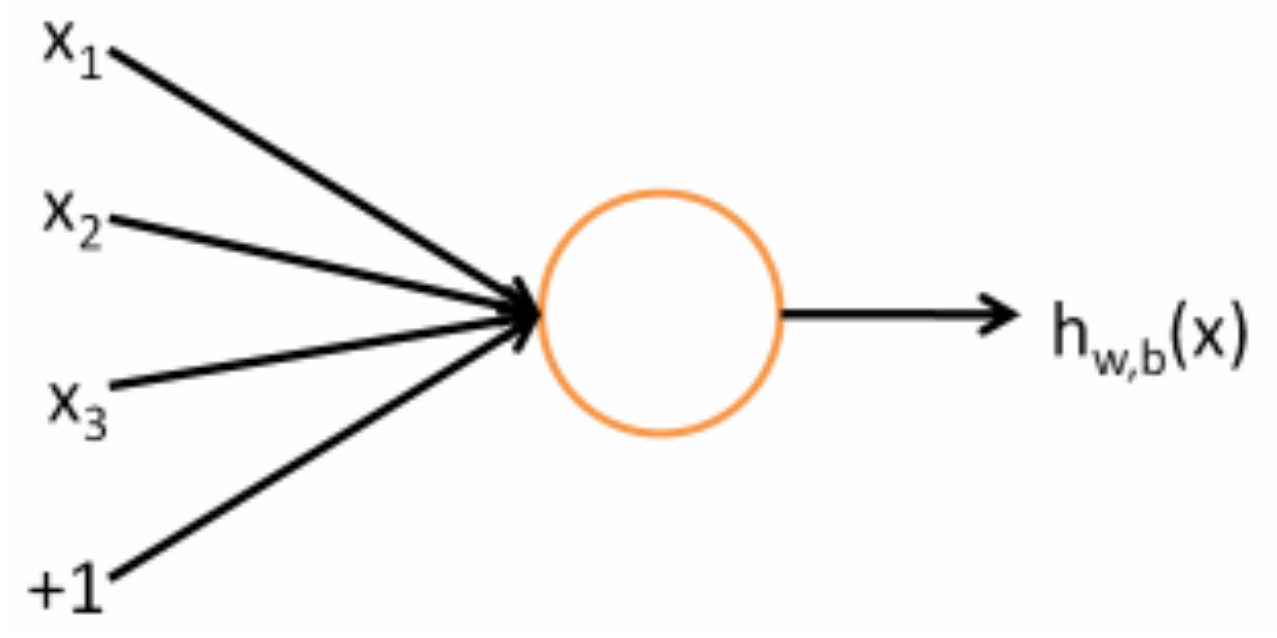
what is **deep learning**?

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

what is **deep learning?**



Logistic Regression by Another Name: Map inputs to output



Input

Vector $x_1 \dots x_d$

Output

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

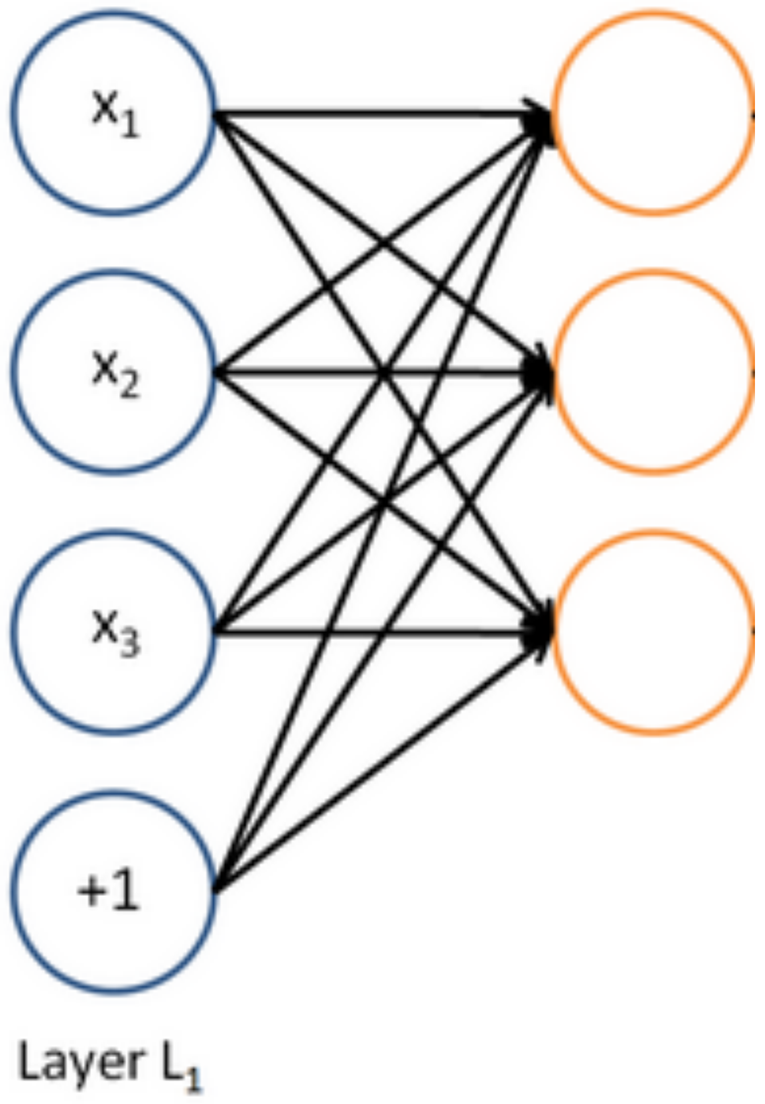
Activation

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

pass through nonlinear sigmoid

NN: kind of like several intermediate logregs

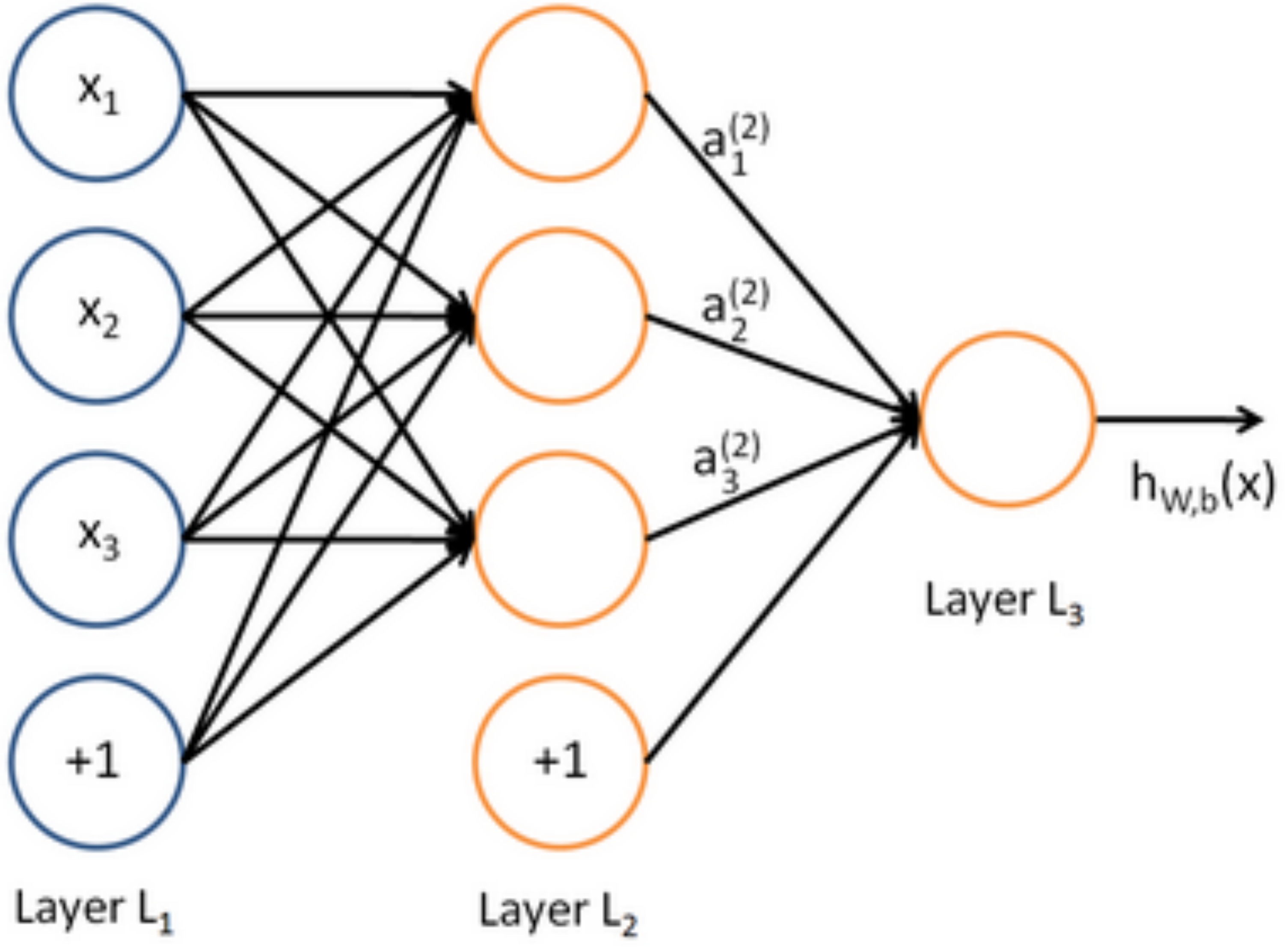
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



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

NN: kind of like several intermediate logregs

… which we can feed into another logistic regression function

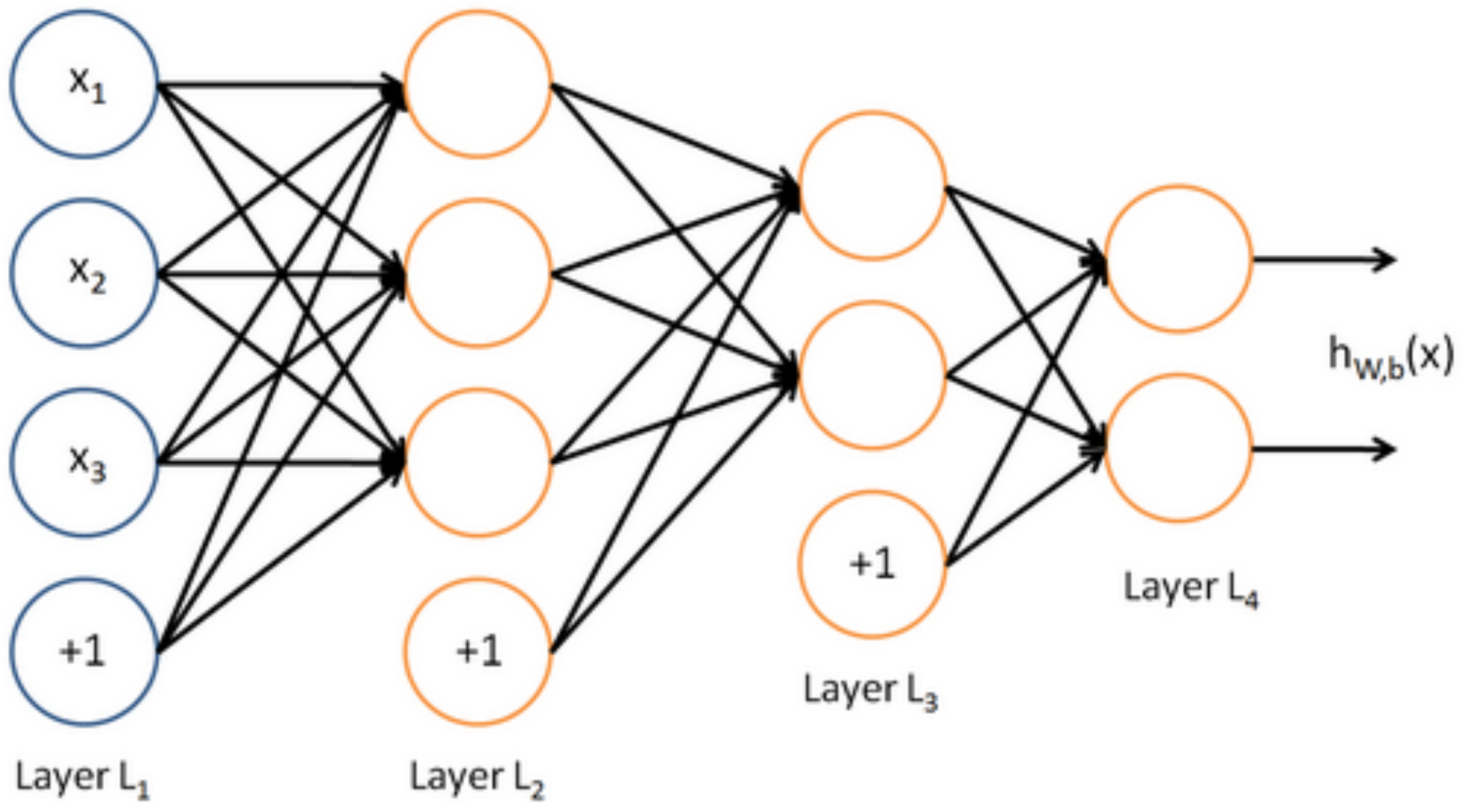


It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

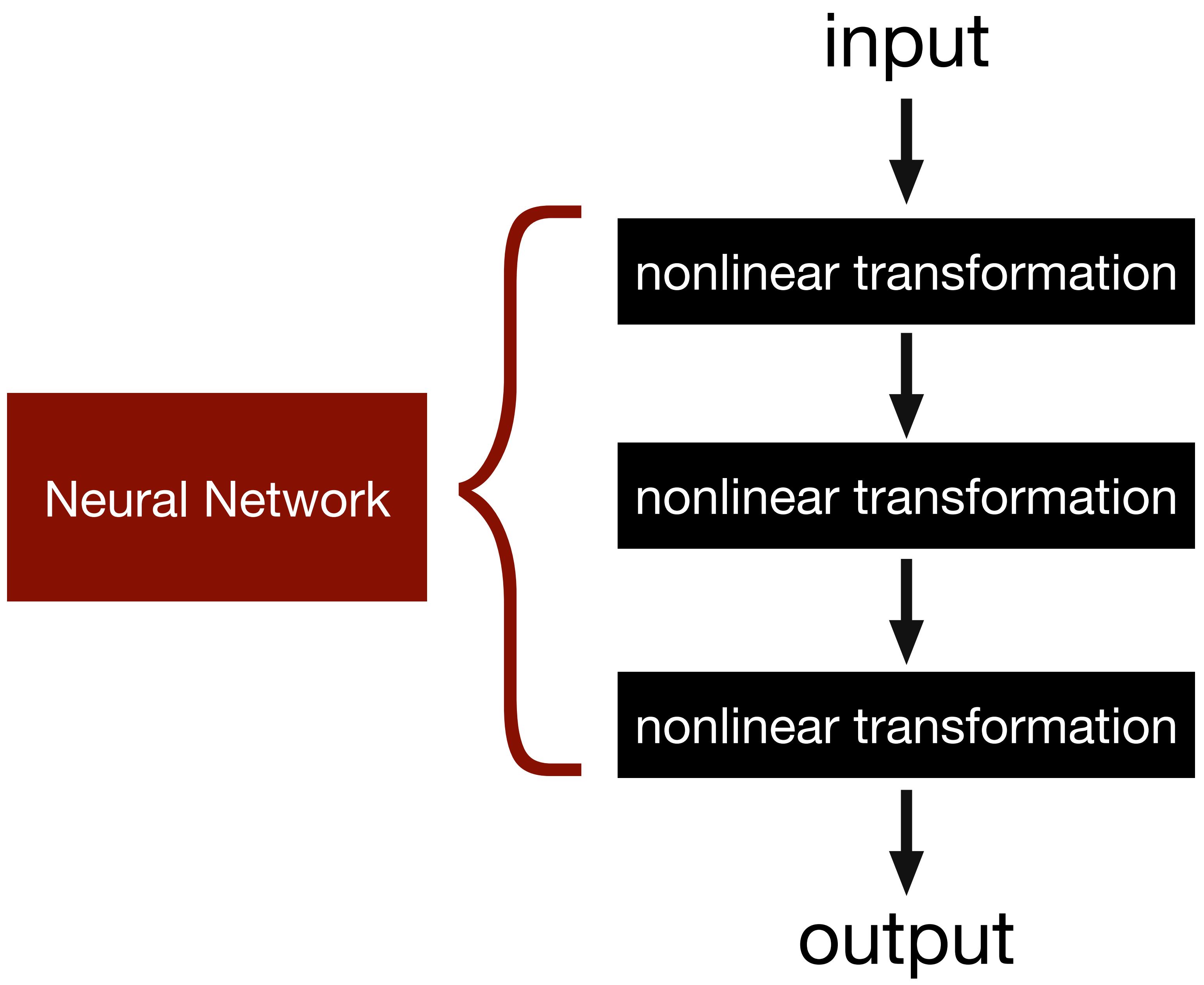
NN: kind of like several intermediate logregs

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

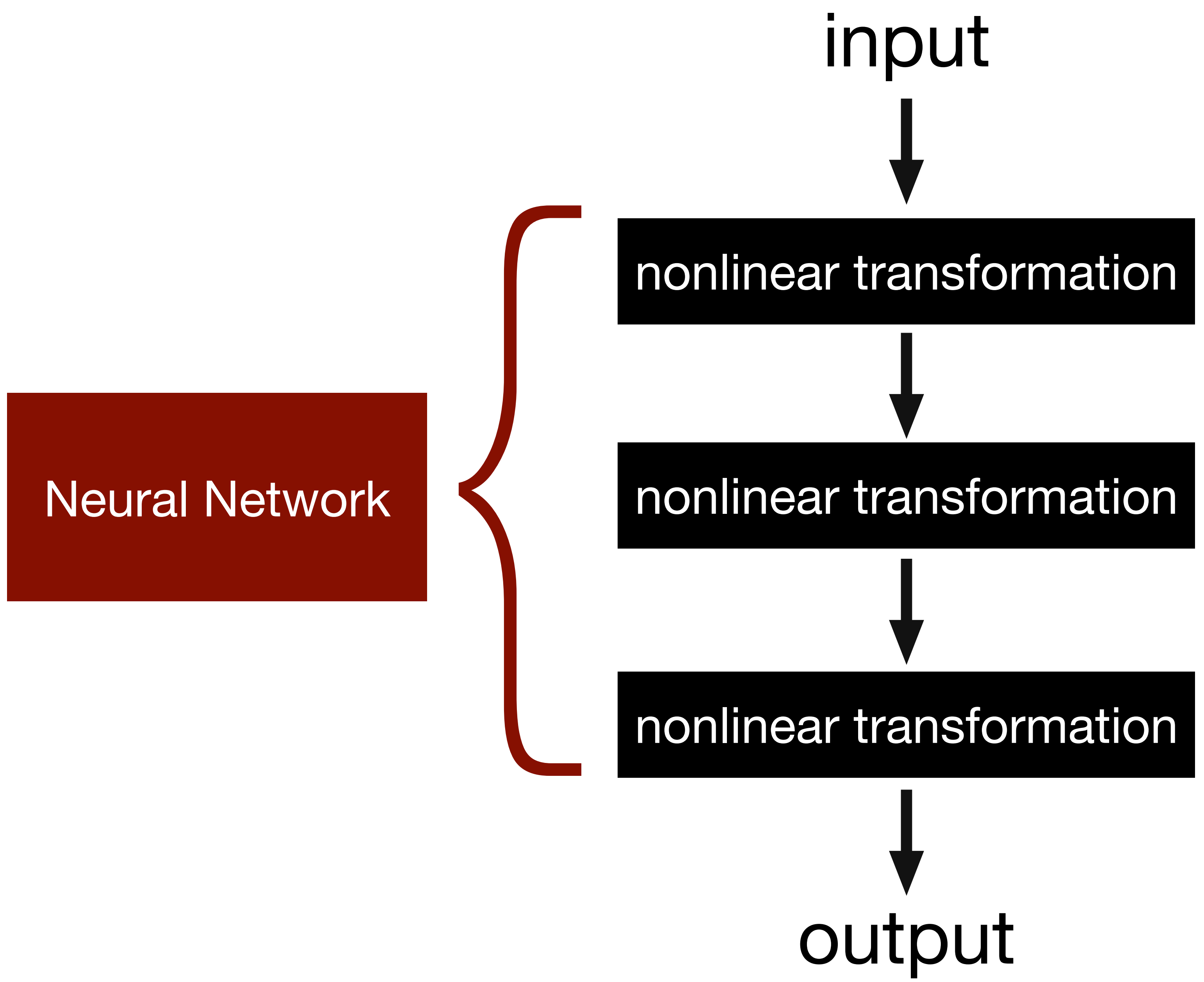
a.k.a. **feedforward network** (see INLP on terminology)



what is deep learning?



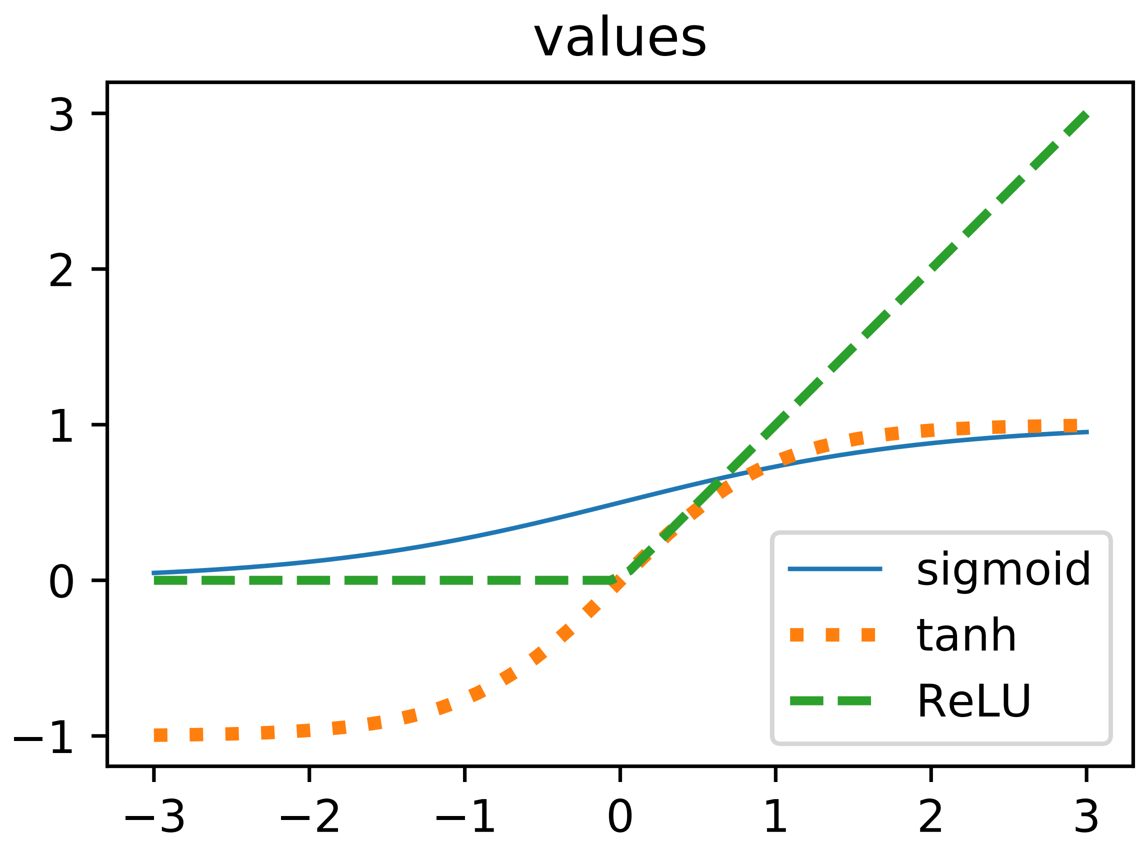
what is deep learning?



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

Nonlinear activations

- "Squash functions"!



- Logistic / Sigmoid

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

- tanh

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

- ReLU

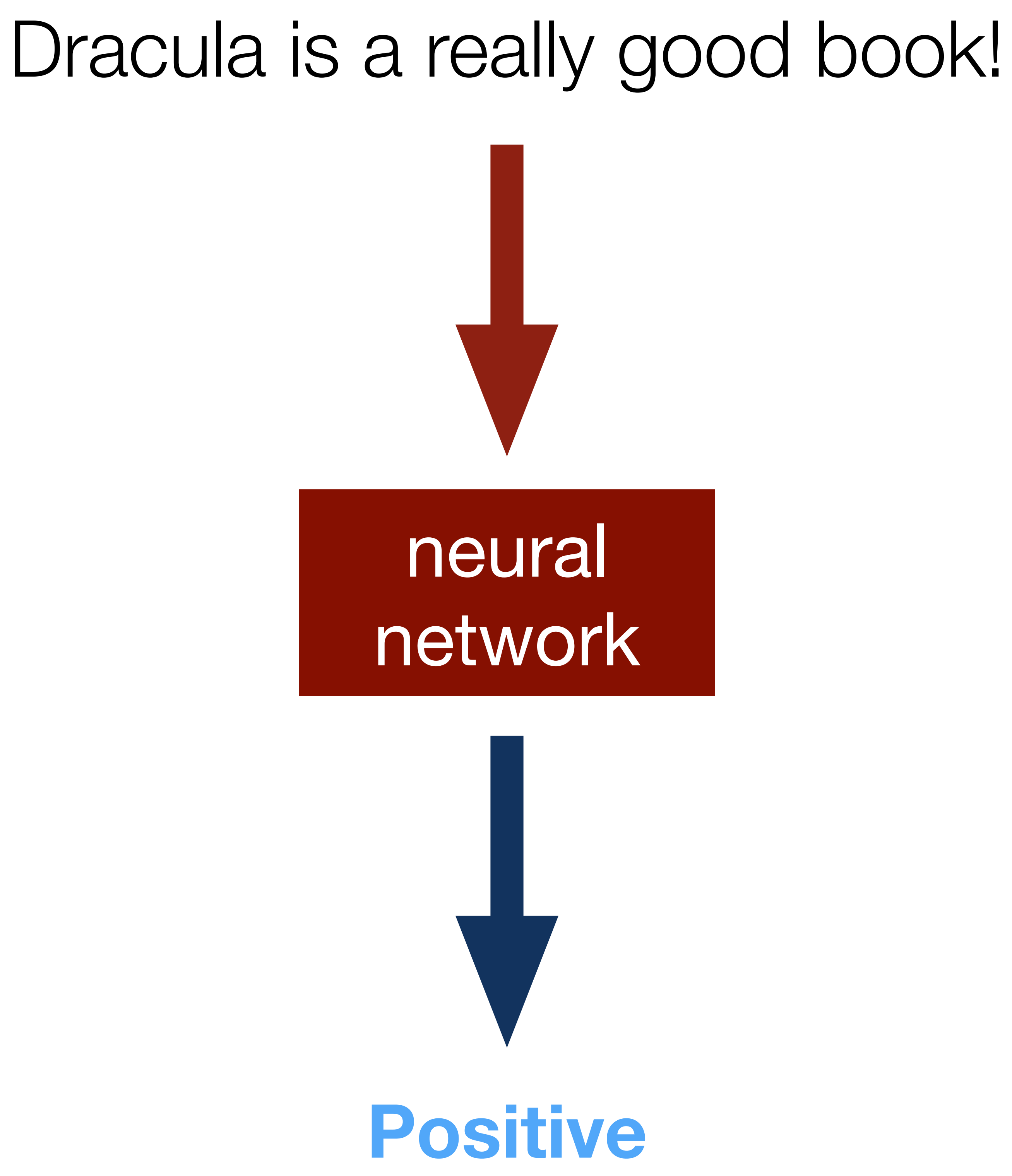
$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases} \tag{3}$$

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*(**x**) = **x**)
more powerful than a one-layer network?

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

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


Dracula is a really good book!
neural network
Positive

softmax function

- let’s say I have 3 classes (e.g., positive, **neutral**, negative)
- use multiclass logreg with “cross product” features between input vector **x** and 3 output classes. for every class *c*, i have an associated weight vector β_c , then

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

softmax function

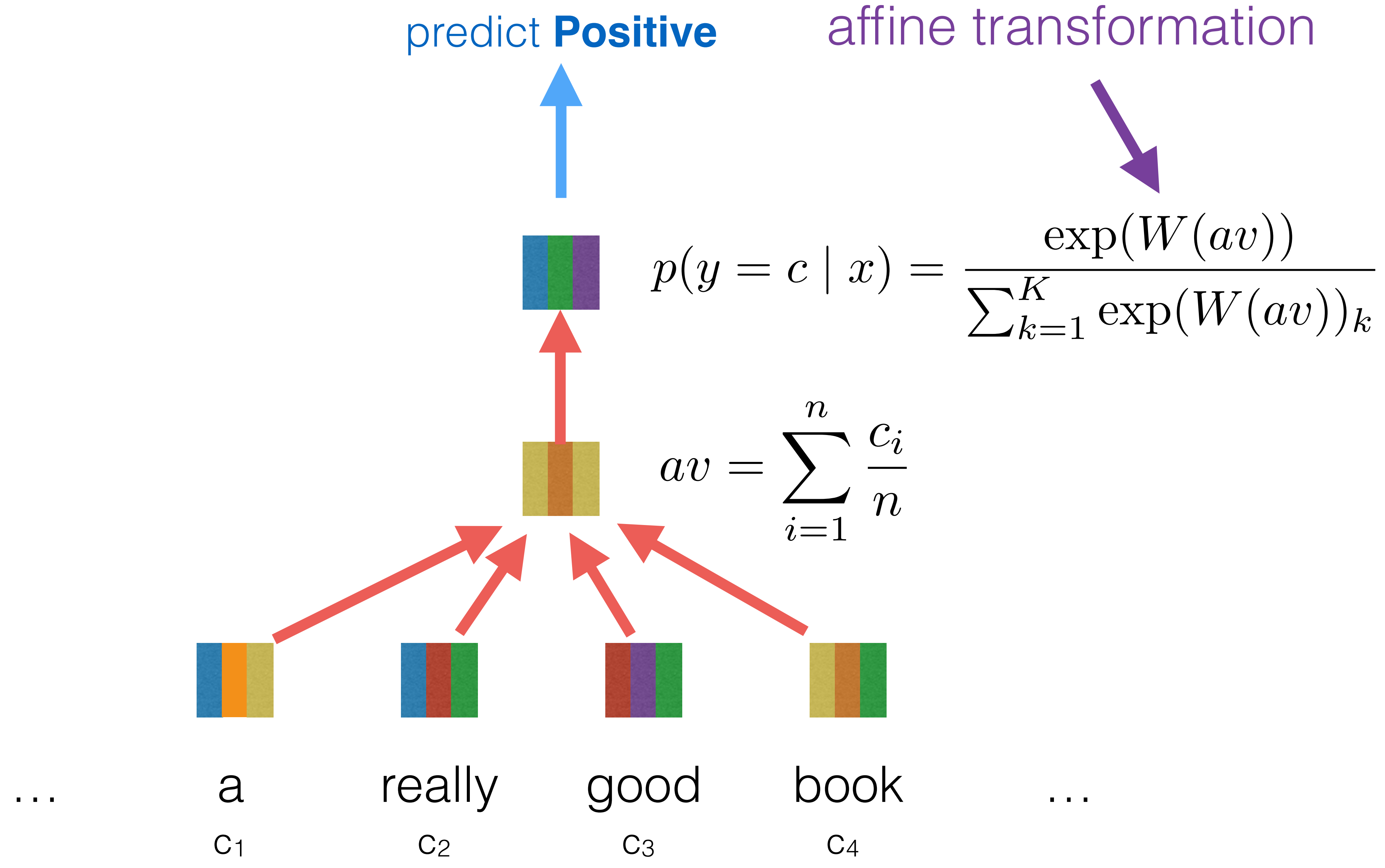
$$\text{softmax}(x) = \frac{e^x}{\sum_j e^{x_j}}$$

x is a vector

x_j is dimension *j* of x

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

“bag of embeddings”

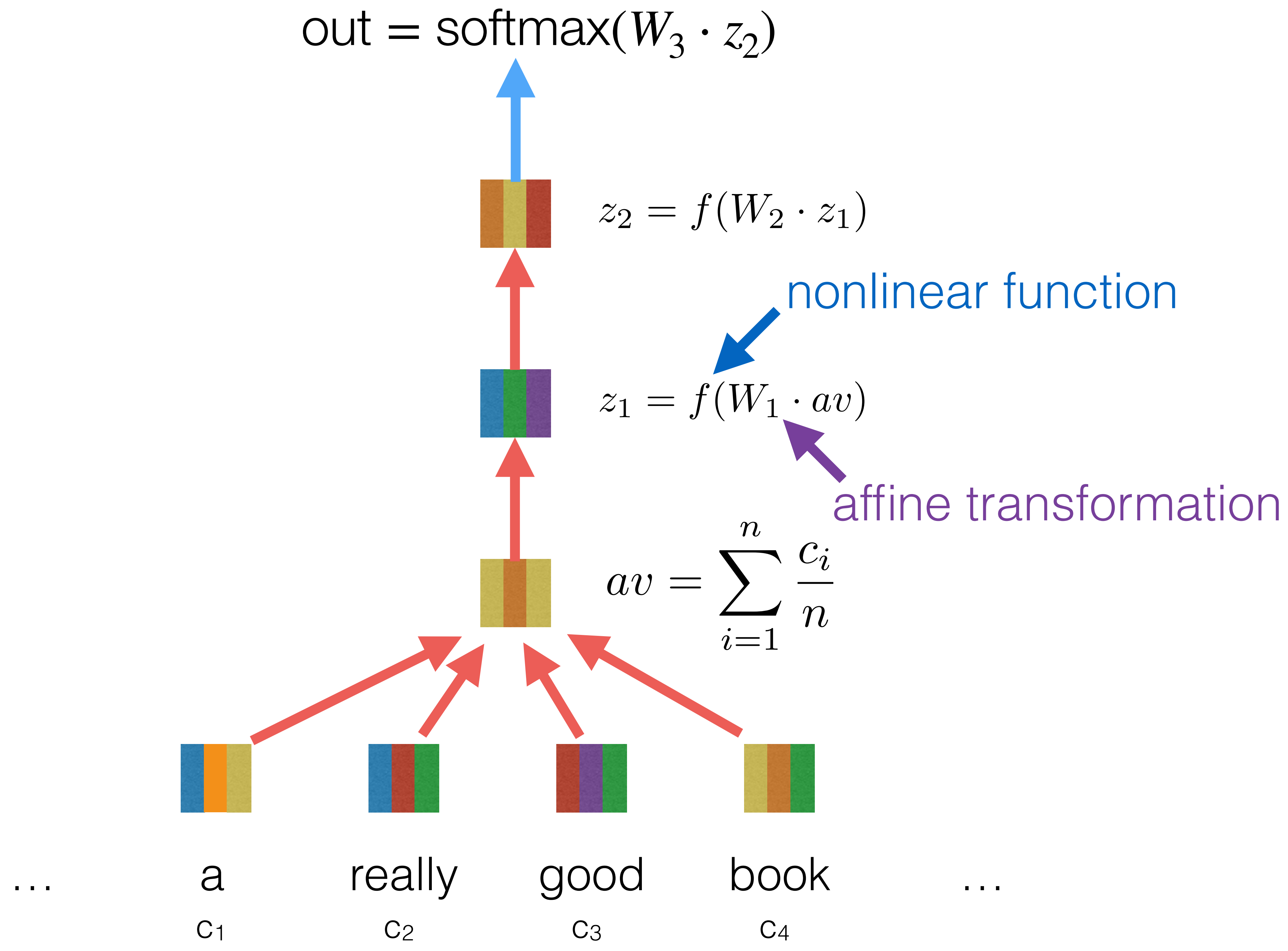


$$p(y = c \mid x) = \frac{\exp(W(av))}{\sum_{k=1}^{K} \exp(W(av))_k}$$

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

Iyyer et al., ACL 2015

deep averaging networks

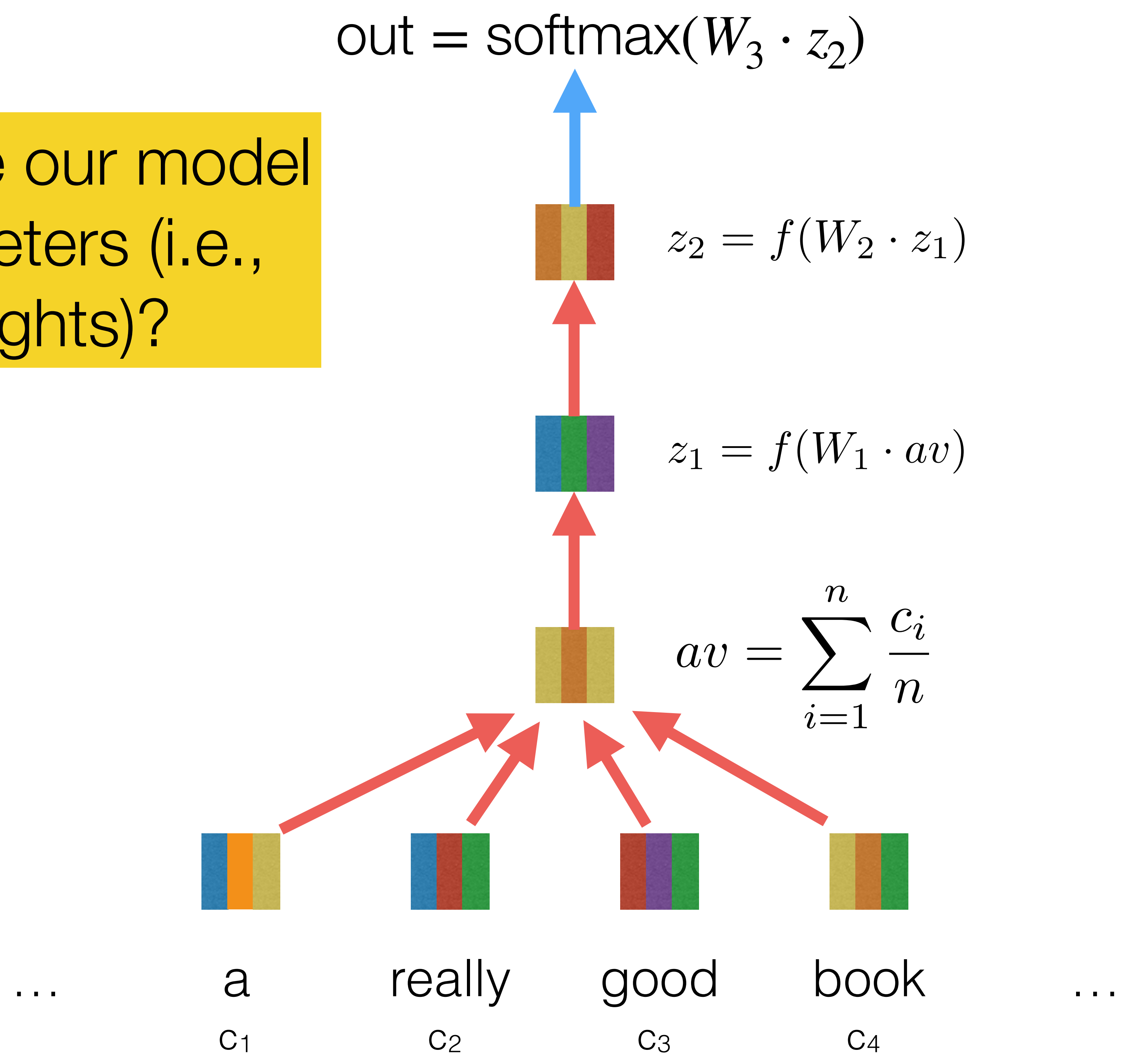


Word embeddings

- Do we need pretrained word embeddings at all?
  - With little labeled data: use pretrained embeddings
  - With lots of labeled data: just learn embeddings directly for your task!
- Think of last week's word embedding models as training an NN-like model (matrix factorization) for a language model-like task (predicting nearby words)
- (Future: in BERT/ELMO, use a pretrained full NN, not just the word embeddings matrix)

deep averaging networks

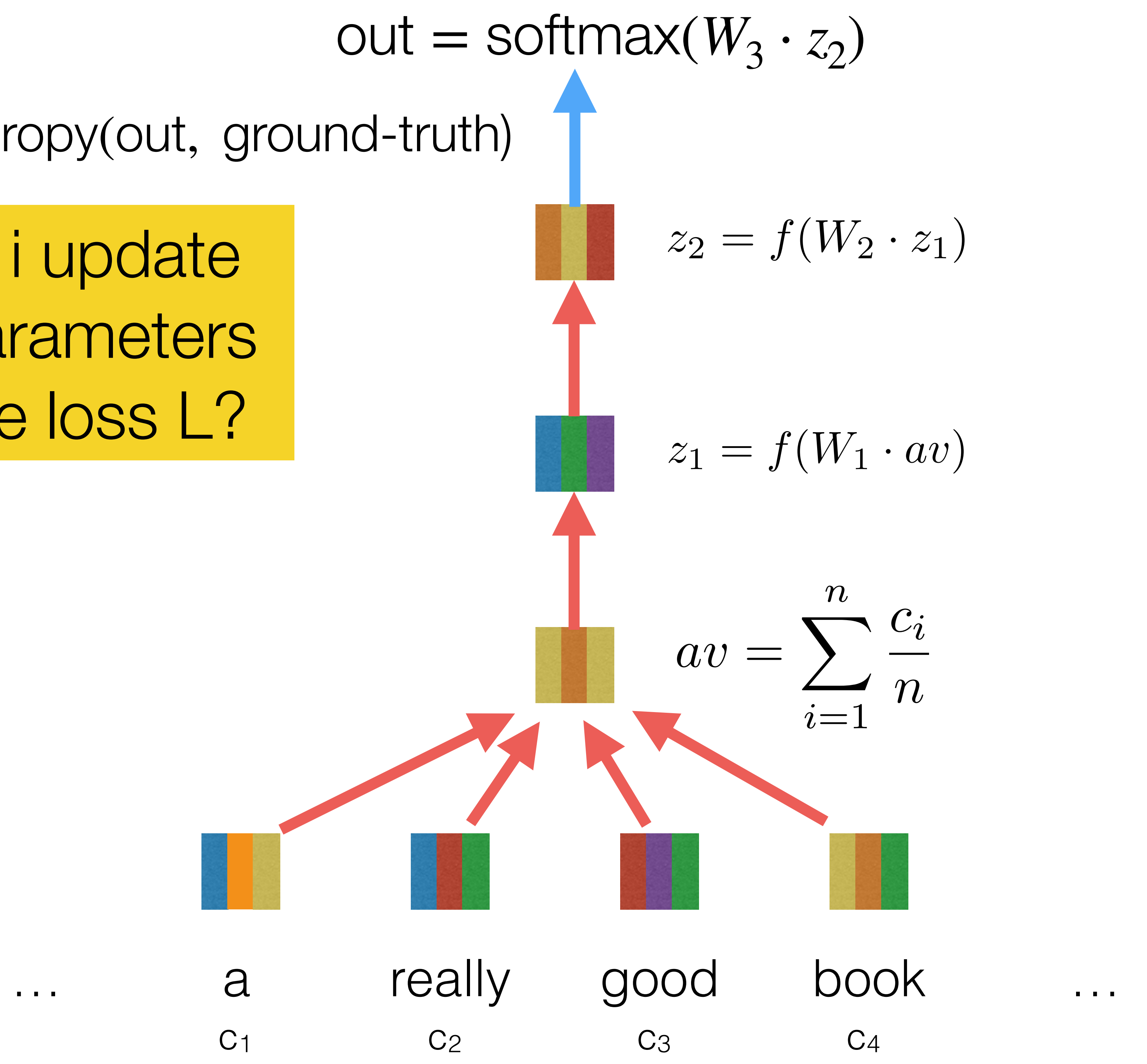
what are our model parameters (i.e., weights)?



deep averaging networks

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

how do i update these parameters given the loss L?

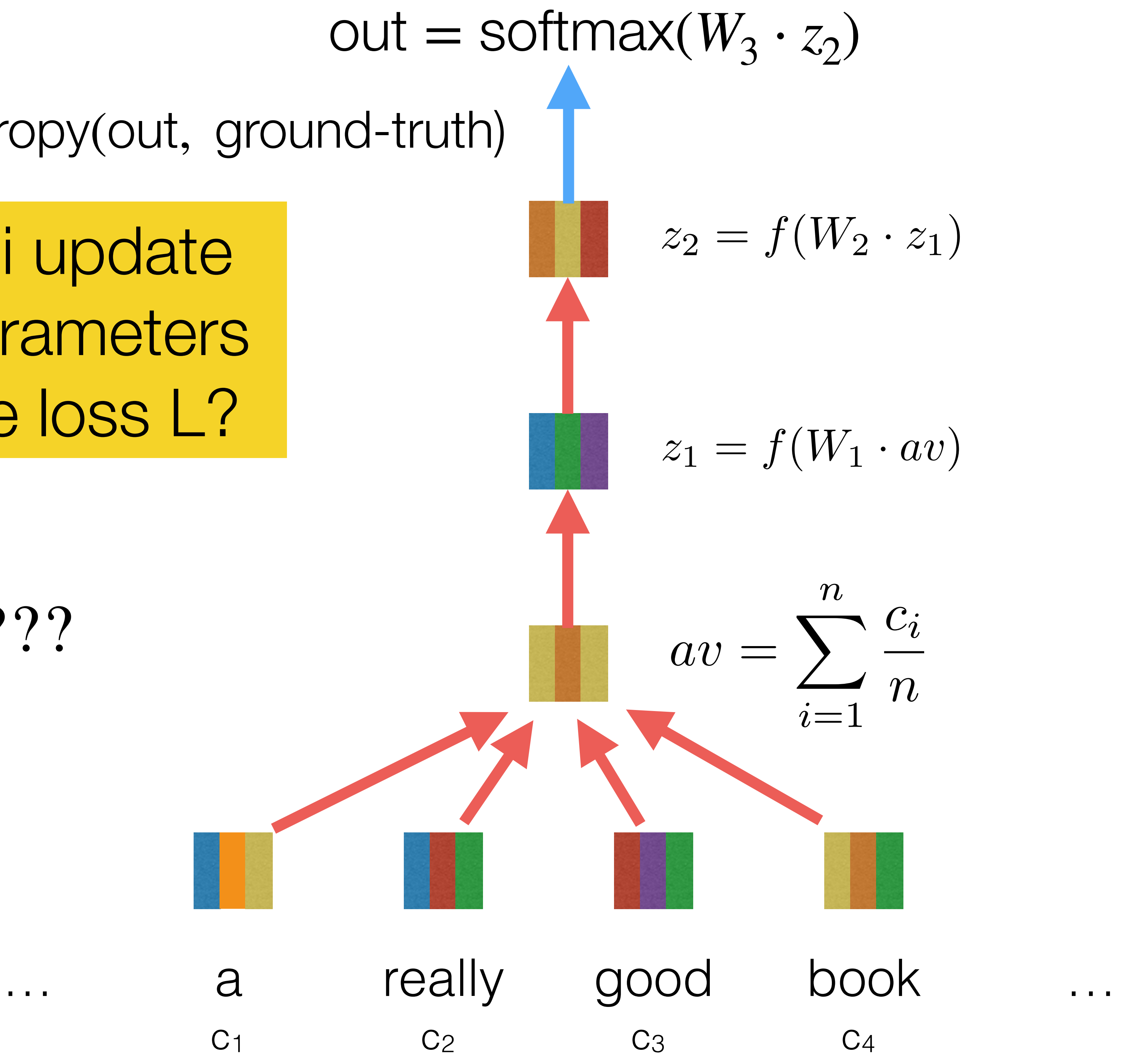


deep averaging networks

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

how do i update these parameters given the loss L?

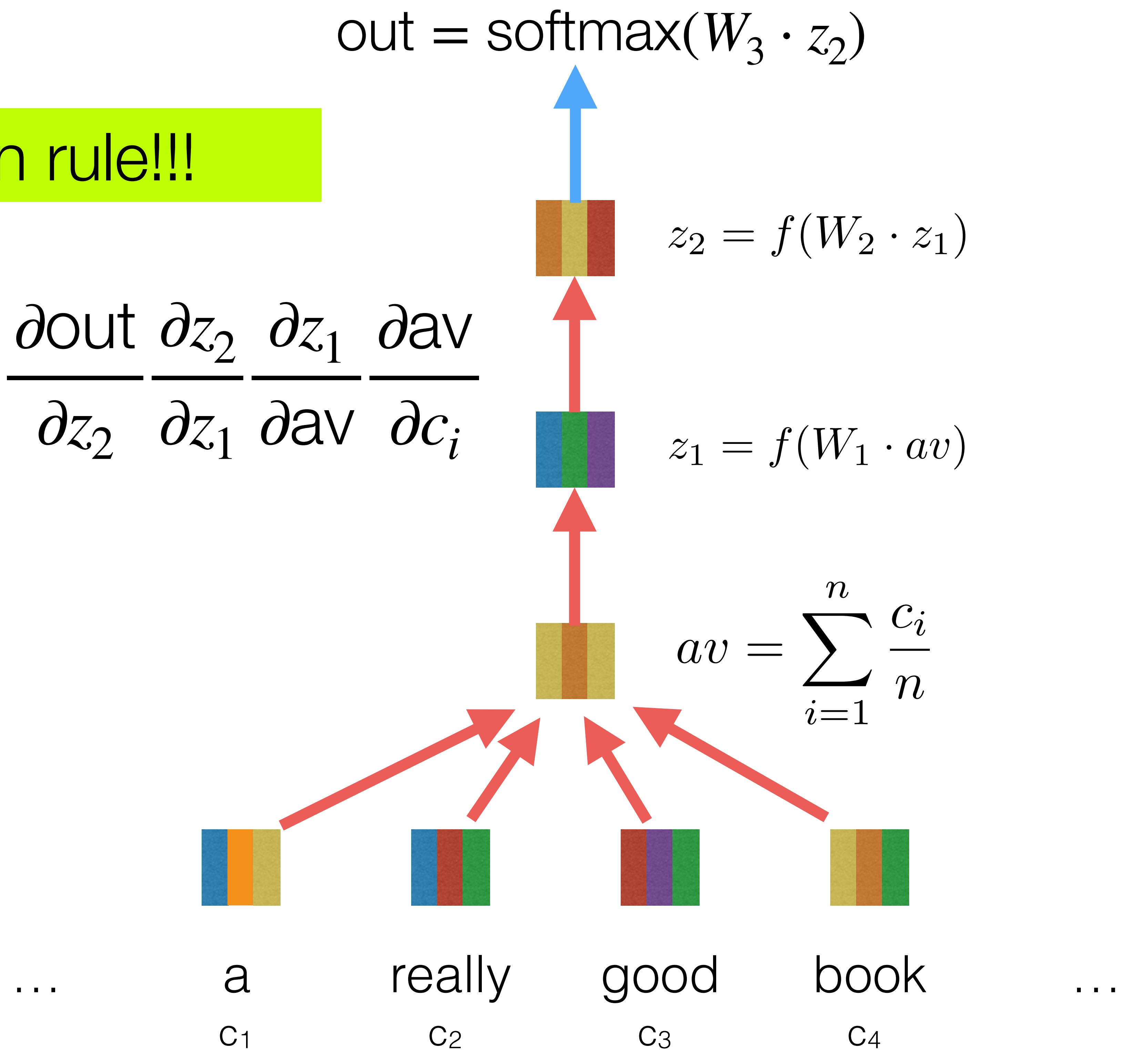
$$\frac{\partial L}{\partial c_i} = ???$$



deep averaging networks

chain rule!!!

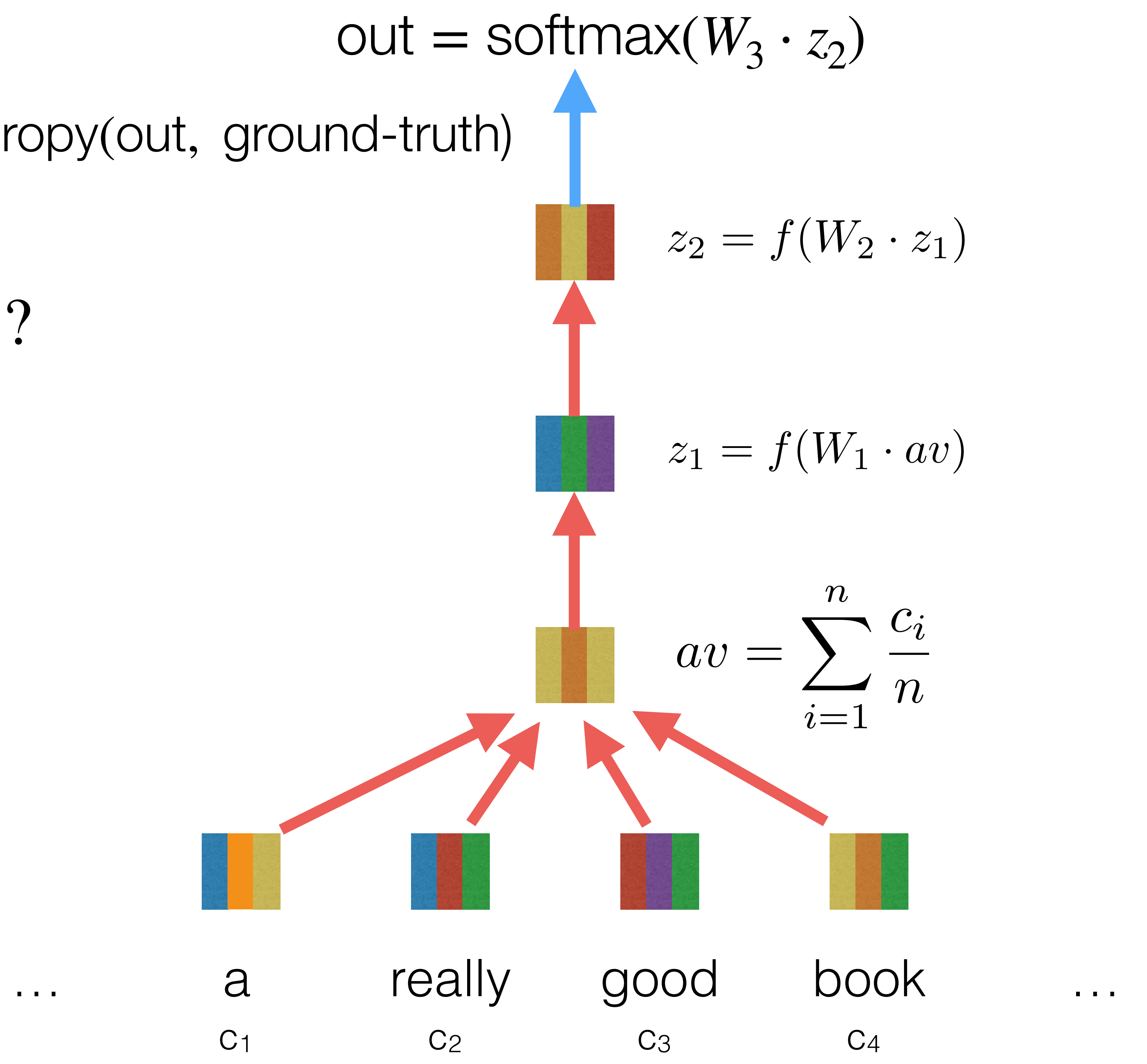
$$\frac{\partial L}{\partial c_i} = \frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2} \frac{\partial z_2}{\partial z_1} \frac{\partial z_1}{\partial \text{av}} \frac{\partial \text{av}}{\partial c_i}$$



deep averaging networks

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

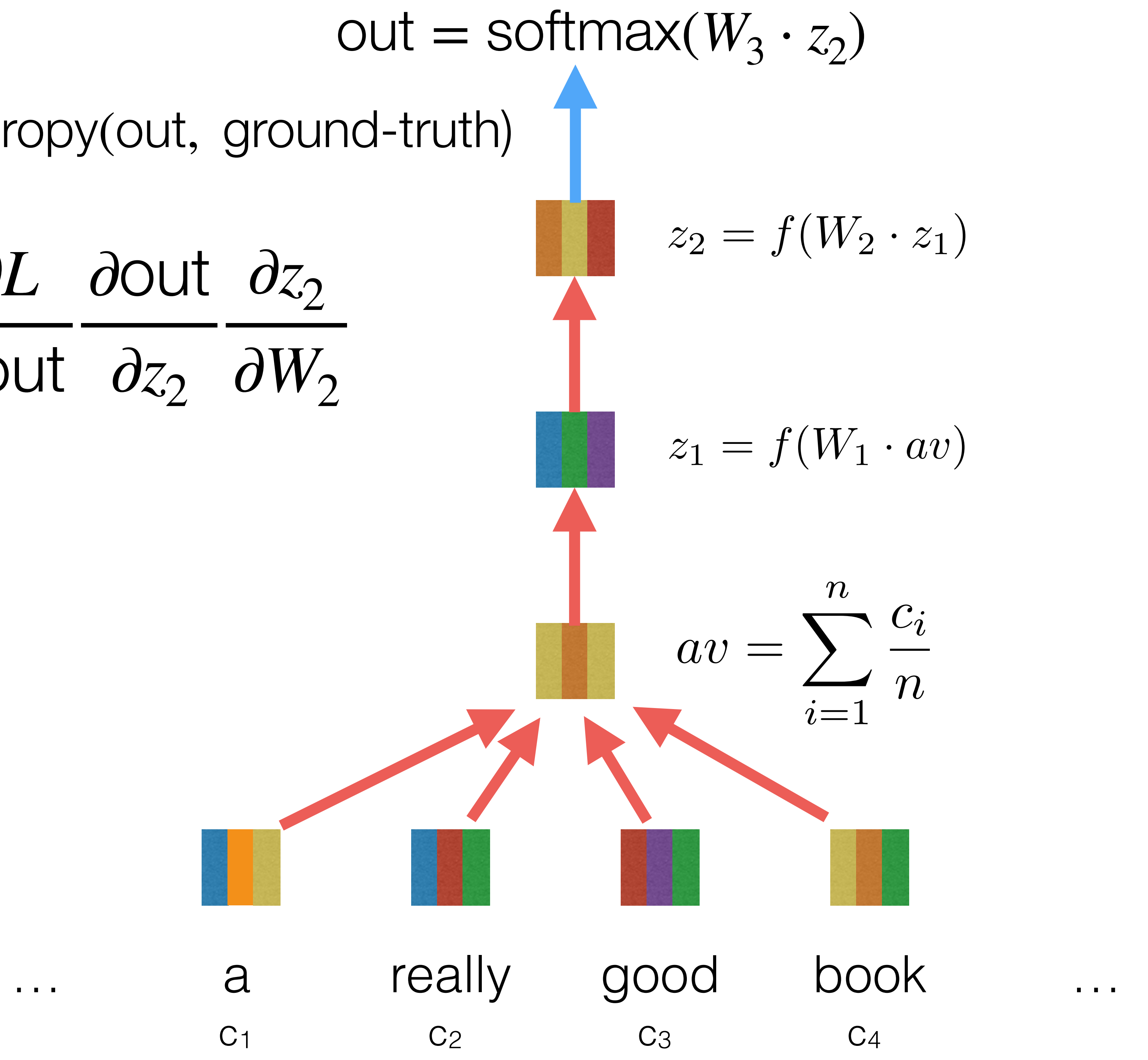
$$\frac{\partial L}{\partial W_2} = ???$$



deep averaging networks

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

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



backpropagation

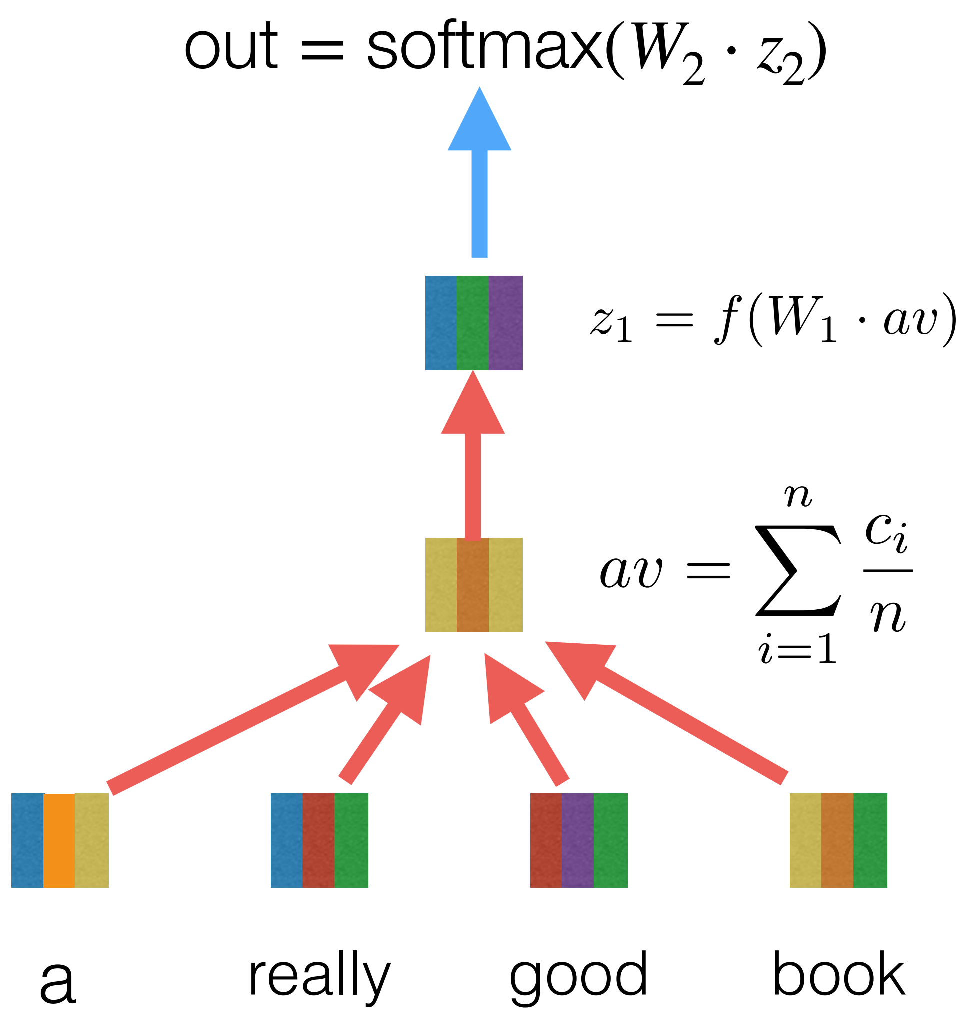
- use the chain rule to compute partial derivatives w/ respect to each parameter
- trick: re-use derivatives computed for higher layers to compute derivatives for lower layers!

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

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

Rumelhart et al., 1986

deep learning frameworks make building NNs super easy!

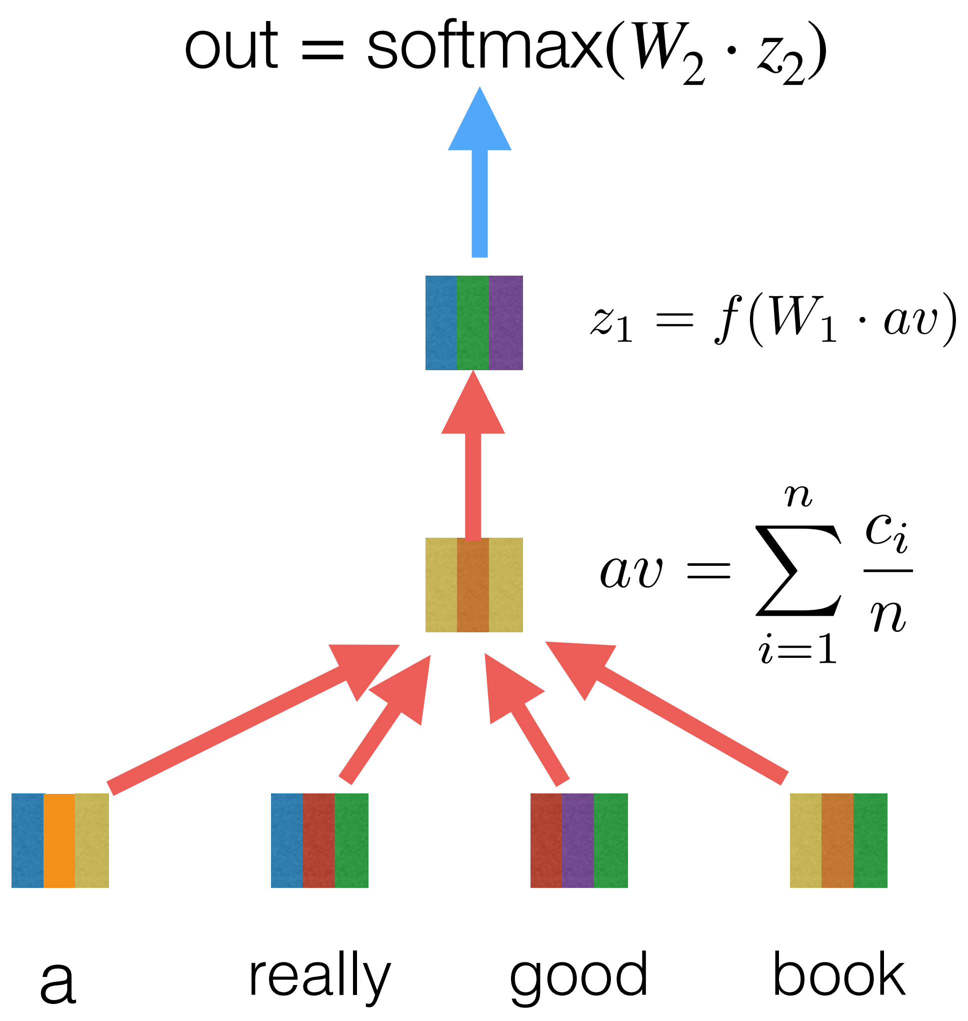


out = softmax($W_2 \cdot z_2$)
$z_1 = f(W_1 \cdot av)$
$av = \sum_{i=1}^{n} \frac{c_i}{n}$
a
really
good
book

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!



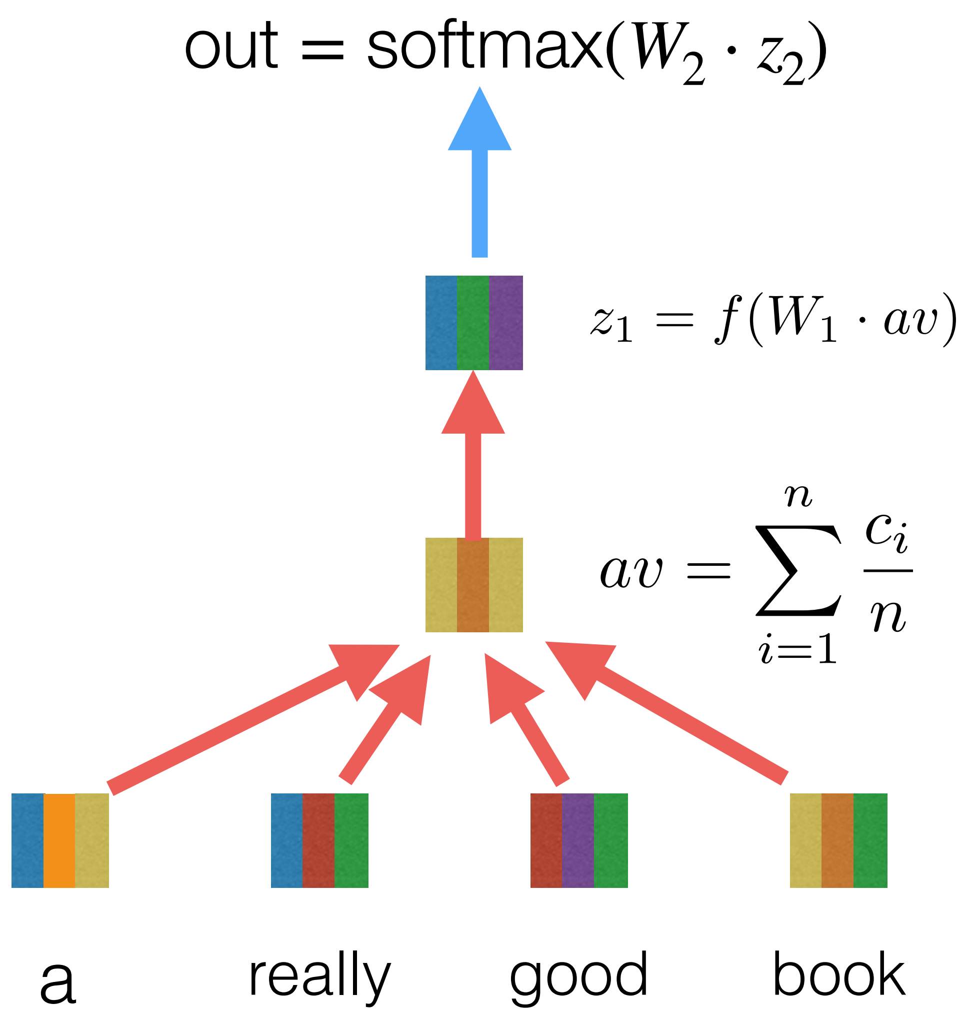
out = softmax$(W_2 \cdot z_2)$
$z_1 = f(W_1 \cdot av)$
$av = \sum_{i=1}^{n} \frac{c_i}{n}$
a
really
good
book

do a forward pass to compute prediction

```
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
```

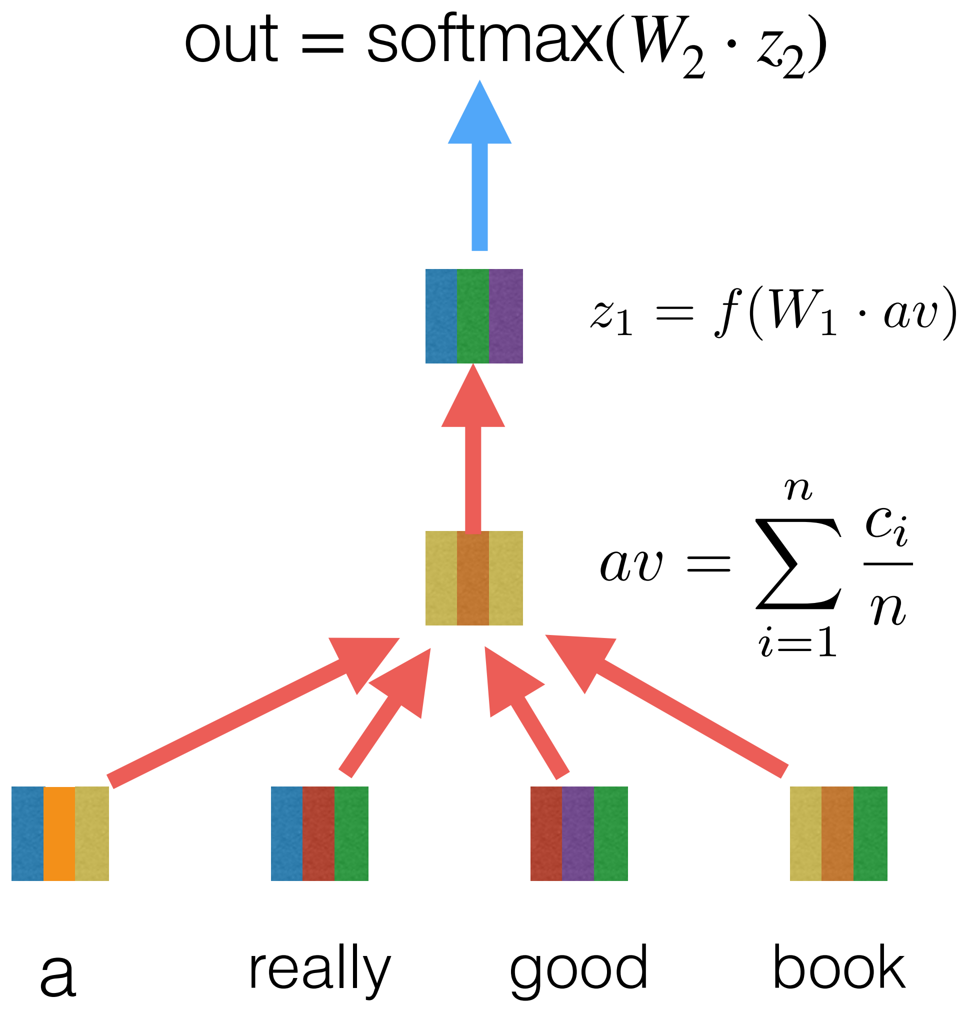
deep learning frameworks make building NNs super easy!


out = softmax($(W_2 \cdot z_2)$)
$z_1 = f(W_1 \cdot av)$
$av = \sum_{i=1}^{n} \frac{c_i}{n}$
a
really
good
book

<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!



out = softmax($W_2 \cdot z_2$)
$z_1 = f(W_1 \cdot av)$
$av = \sum_{i=1}^{n} \frac{c_i}{n}$
a
really
good
book

do a backward pass to update weights

```
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()
```

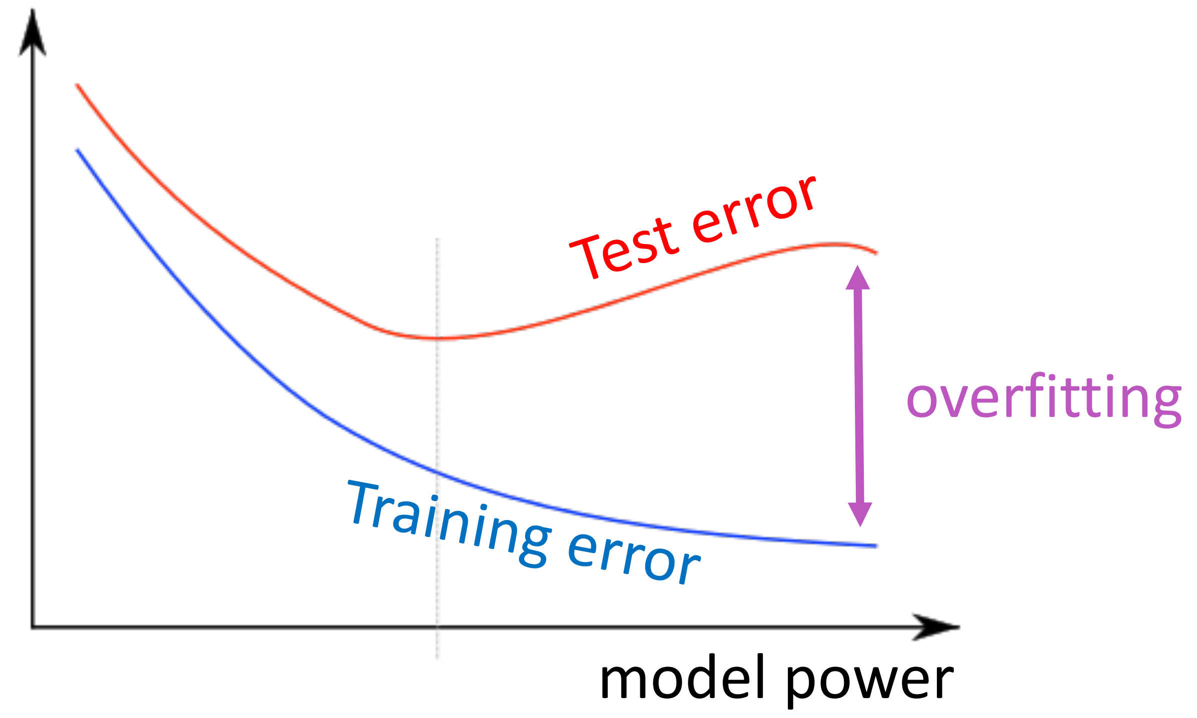
that's it! no need to compute gradients by hand!

Stochastic gradient descent for parameter learning

- Neural net objective is non-convex. How to learn the parameters?
- SGD: iterate many times,
    - Take sample of the labeled data
    - Calculate gradient. Update params: step in its direction
        - (Adam/Adagrad SGD: with some adaptation based on recent gradients)
- No guarantees on what it learns, and in practical settings doesn't exactly converge to a mode. But often gets to good solutions (!)
    - Best way to check: At each epoch (pass through the training dataset), evaluate current model on development set. If model is getting a lot worse, stop.

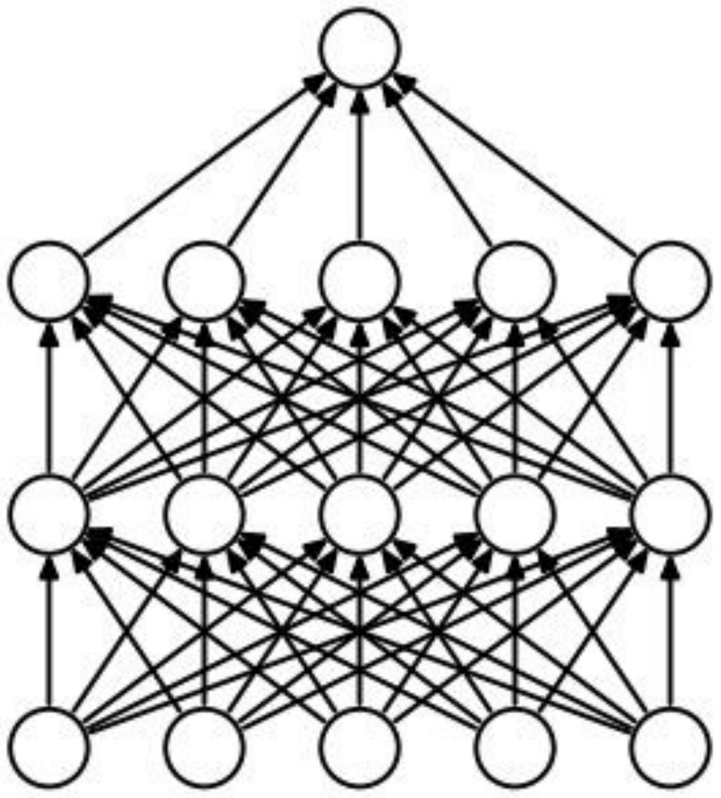
How to control overfitting?

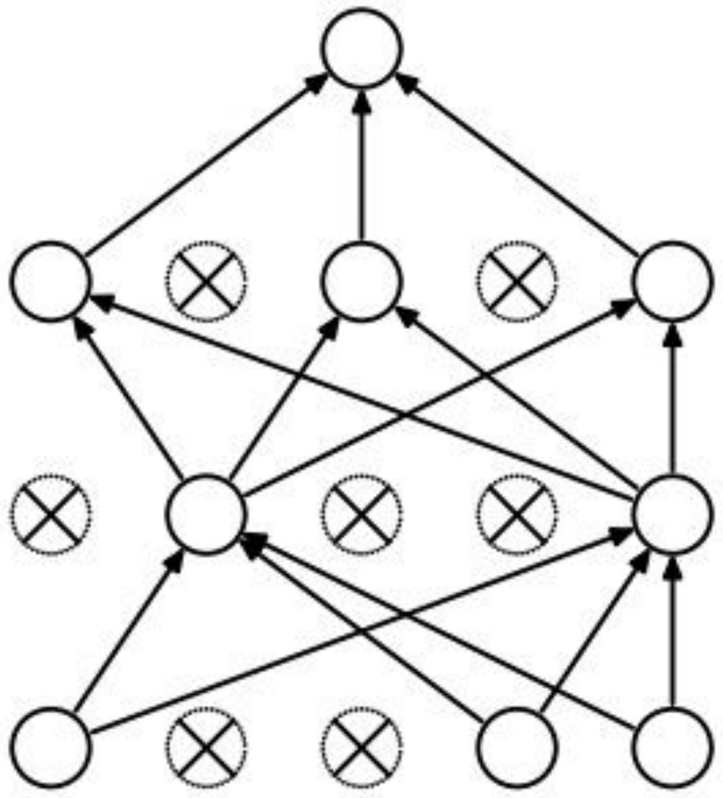
- Most popular for non-NN logreg: L2 regularization (or similar)
- Most popular for neural networks
  - Early stopping
  - Dropout (next slide)



Dropout for NNs

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

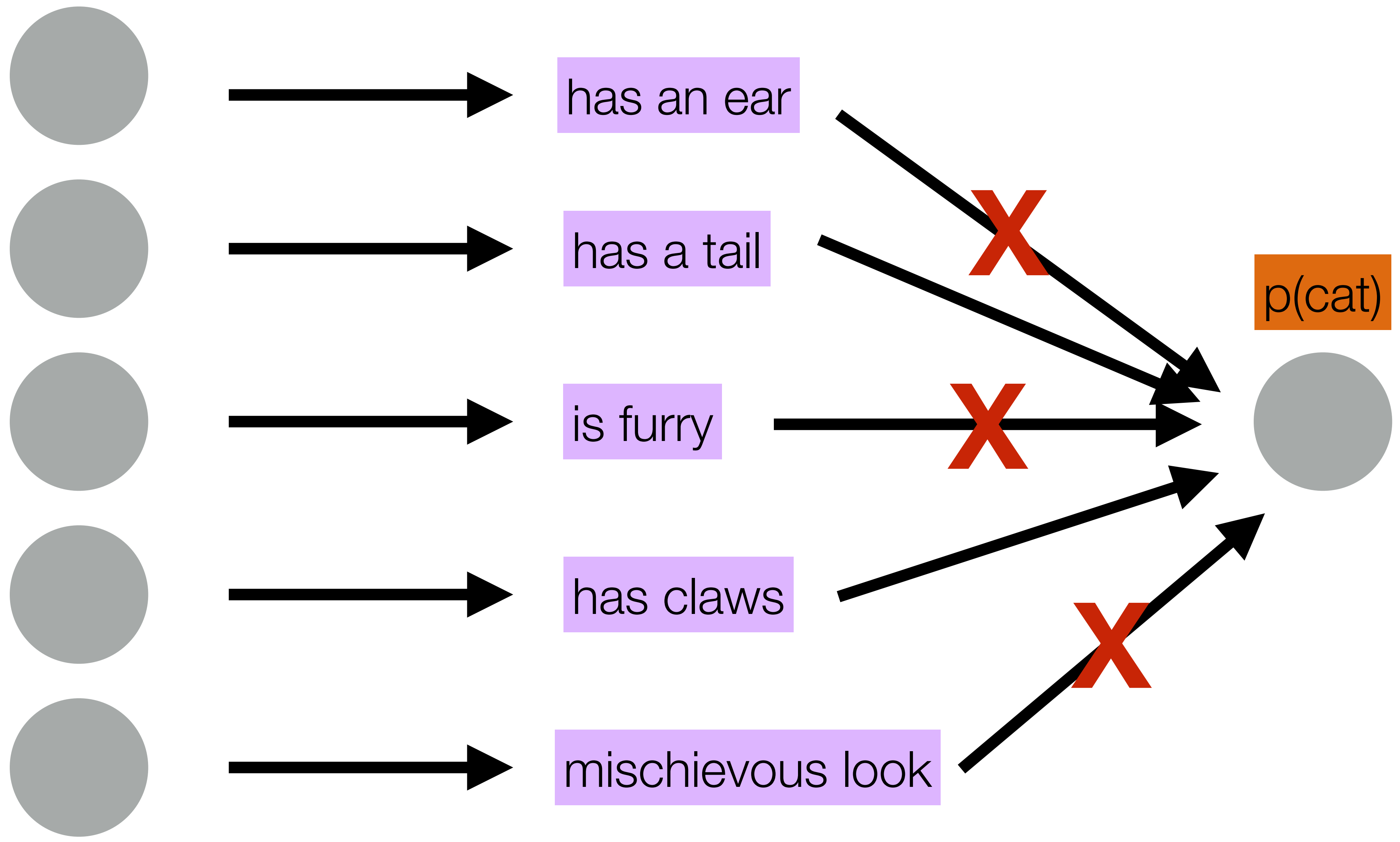

(a) Standard Neural Net


(b) After applying dropout.

[Srivastava et al., 2014]

Why?

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



A few other tricks

- Training can be unstable! Therefore some tricks.
  - Initialization — random small but reasonable values can help.
  - Layer normalization (very important for some recent architectures)
- Big, robust open-source libraries to let you computation graphs, then run backprop for you
  - PyTorch, Tensorflow (+ many higher-level libraries on top; e.g. HuggingFace, AllenNLP…)

NNs in NLP

- Easy to experiment with different architectures - lots of research
  - Today: averaging & deep averaging
  - Others: convolutional NNs, recurrent NNs (incl. LSTMs), ...
  - After midterm: self-attention "Transformers"
- Very useful: Pretrained NN language models (similar to pretraining word embeddings), applied to any task
  - "BERT" = Pretrained Transformers