

# Neural Networks in NLP

CS 485, Fall 2024

Applications of Natural Language Processing

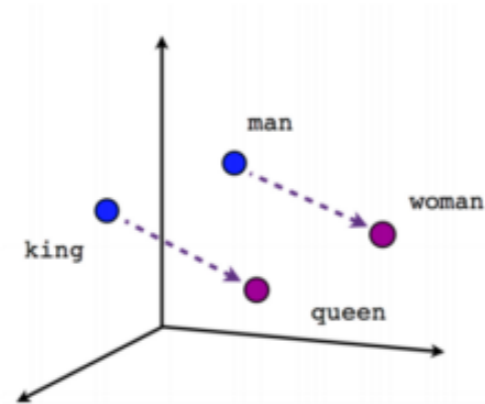
Brendan O'Connor

College of Information and Computer Sciences  
University of Massachusetts Amherst

*[Slides from Mohit Iyer and Richard Socher]*

- Progress report: due Friday 11/15 (next week)
- HW3 and HW4 after that (short)
  - HW3: using BERT for classification (out next week)
  - HW4:TBD
- Final presentations: Dec 3 & 5

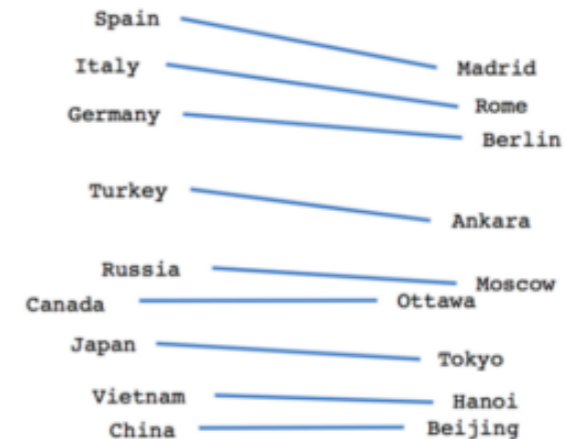
- We have word embeddings...



Male-Female
















Verb tense



Country-Capital

- Can we compose meanings (embeddings) for phrases, sentences, etc.?

neural network (           ) =   

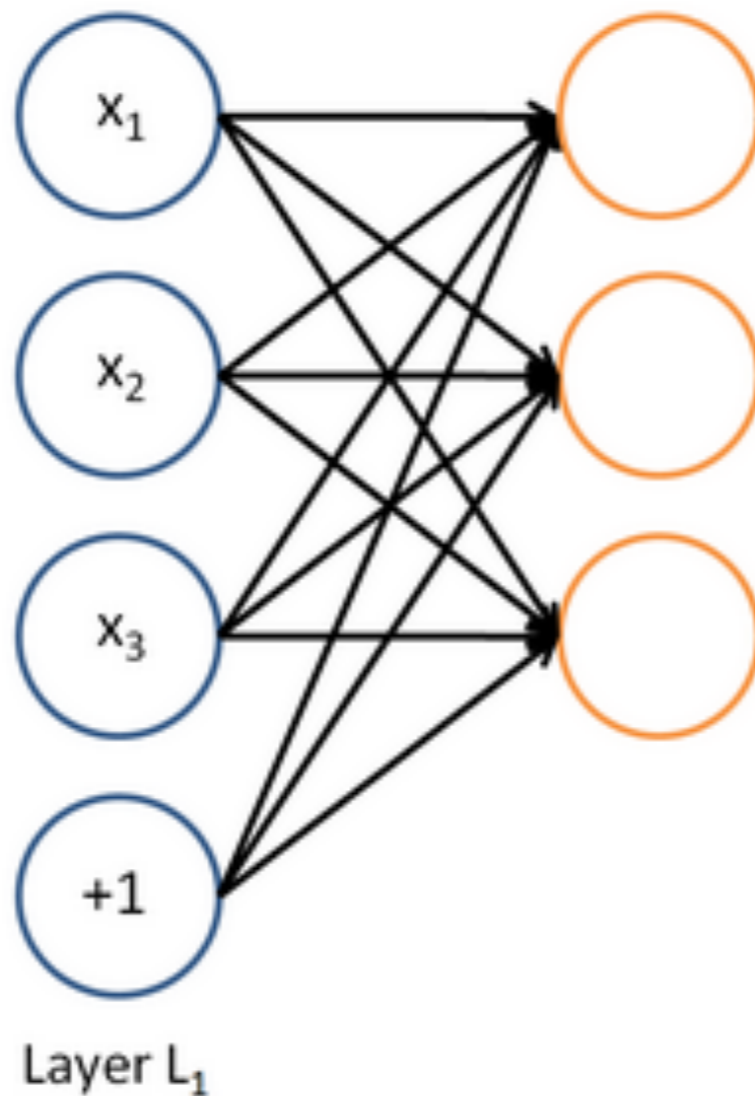
- Or, contextual meaning for *each* token?
- Key idea: *automatically* determine how to combine embedding from different tokens

# Neural Network models

- We've looked at linear models within logistic regression (for classification & language modeling)
- A proper neural network adds a **hidden layer** with a **non-linear activation function**, which allows (in theory) it to learn arbitrary functions (!) via yet more latent representations
- Pros:
  - flexible learning algorithm allows for customized architectures to reflect sequential language data
    - and thus latent representations for tokens, phrases, etc., beyond just word embeddings
  - and can make them really big, if you have lots of data
  - and can be parallelized internally (via GPUs or similar)
- Cons:
  - often unclear what it's learning internally ("black box")
  - they can get slow & compute-intensive

NN: kind of like several intermediate logregs

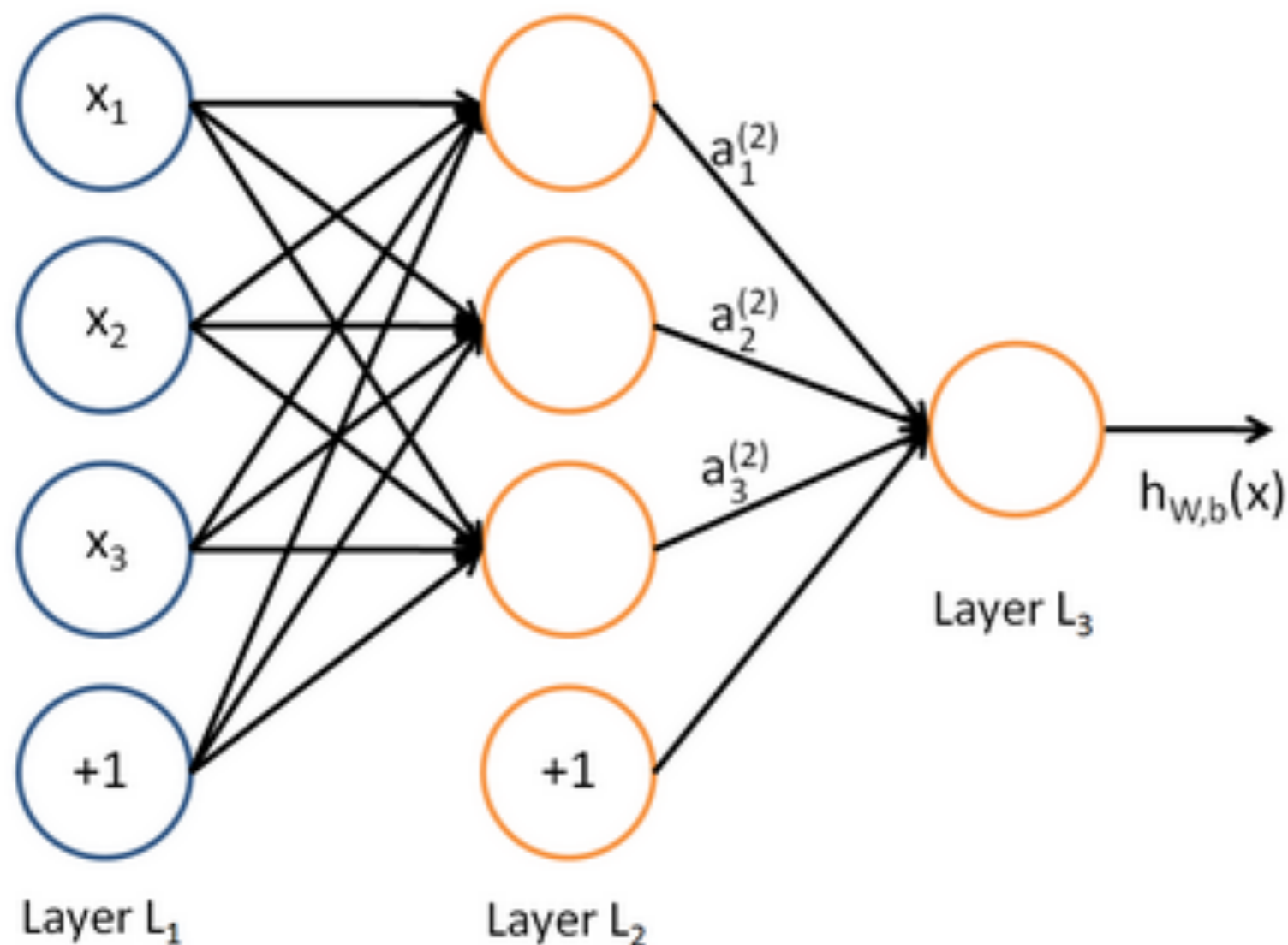
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...



*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

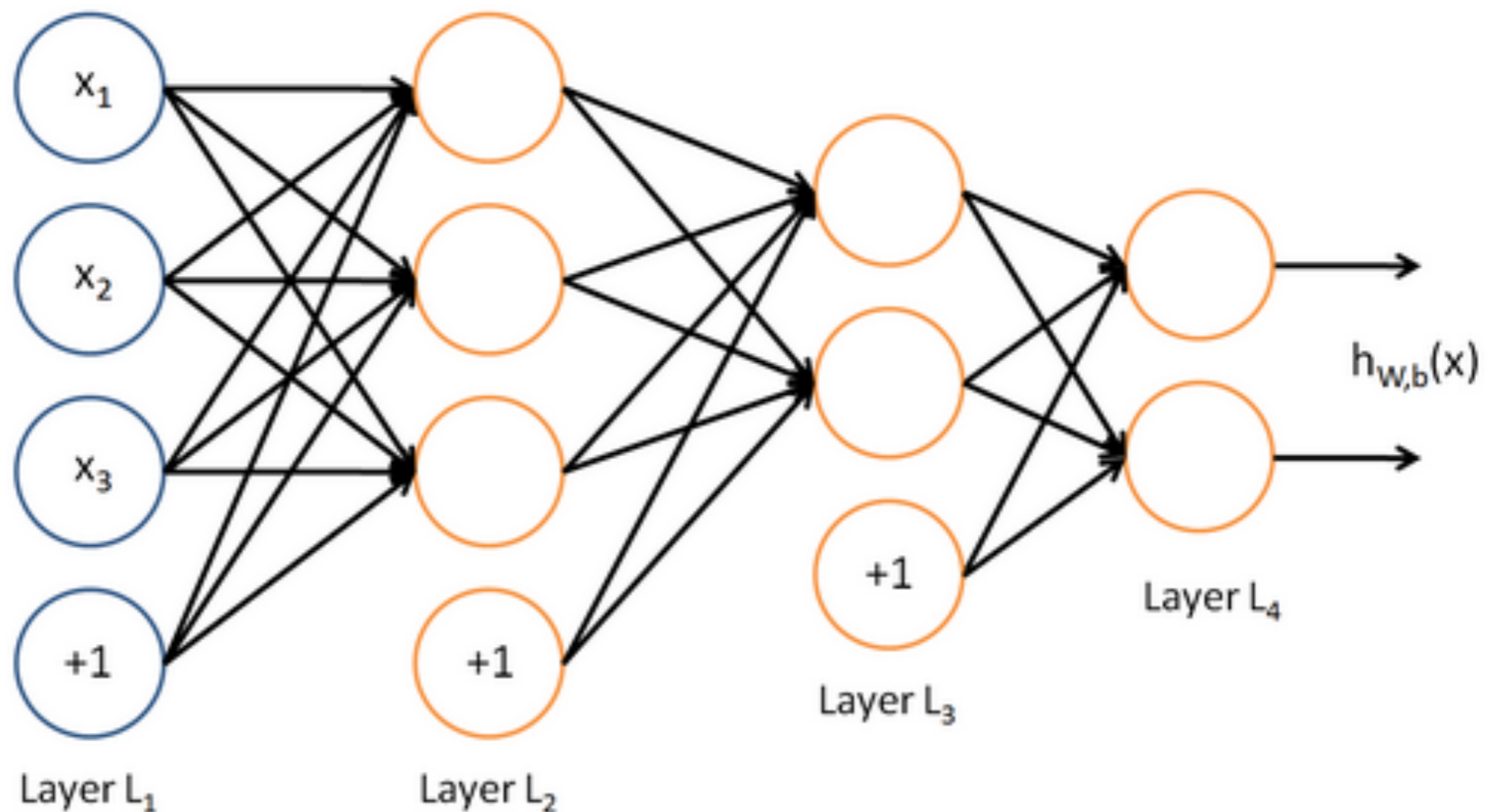


*It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.*

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)



# Nonlinear activations

- “Squash functions”!

- Logistic / Sigmoid

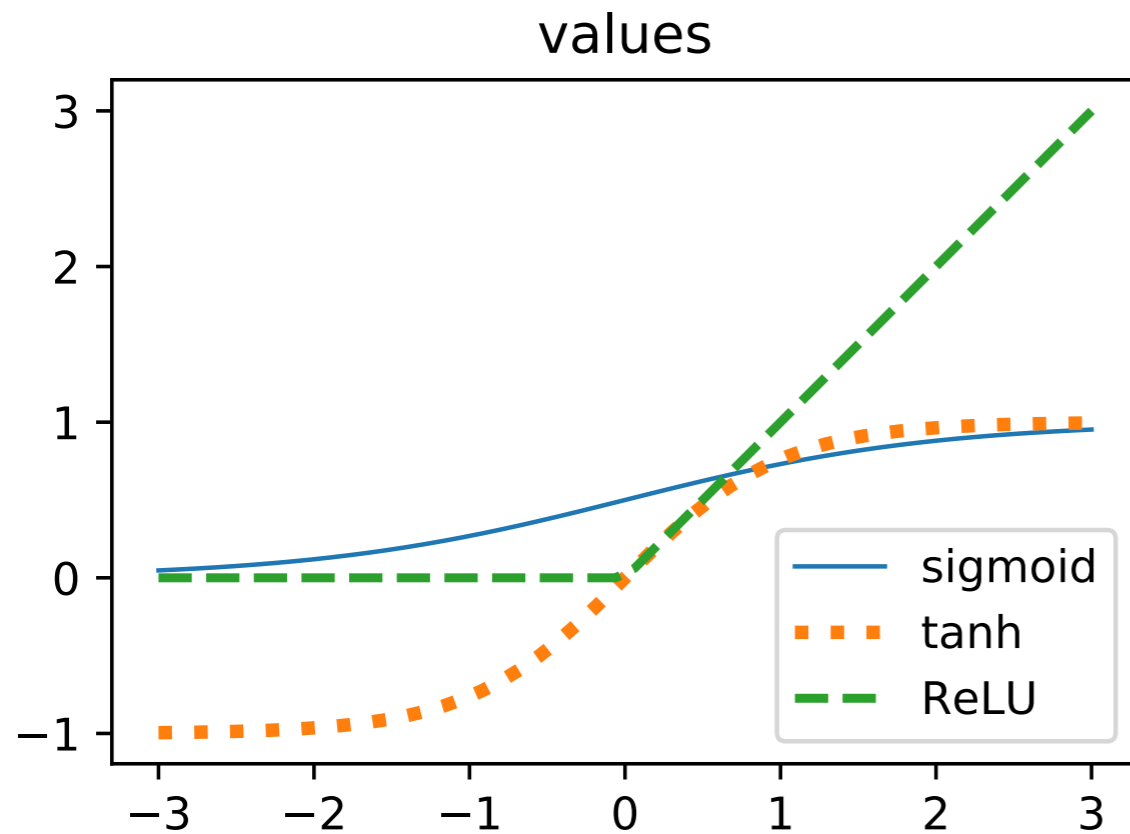
$$f(x) = \frac{1}{1 + e^{-x}} \quad (1)$$

- tanh

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

- ReLU

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





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?

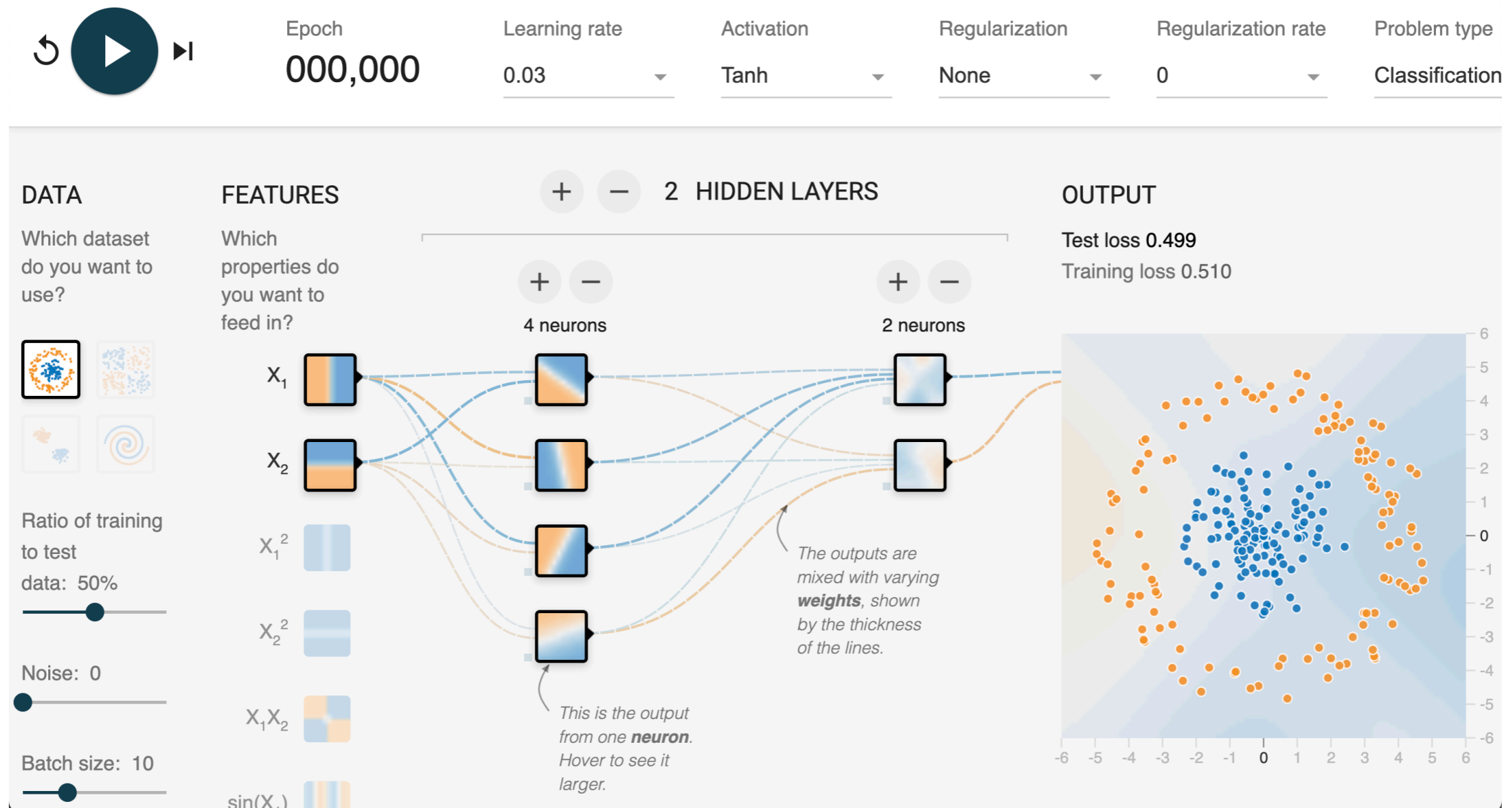
is a multi-layer neural network with no nonlinearities  
(i.e.,  $f$  is the identity  $f(\mathbf{x}) = \mathbf{x}$ )  
more powerful than a one-layer network?

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

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

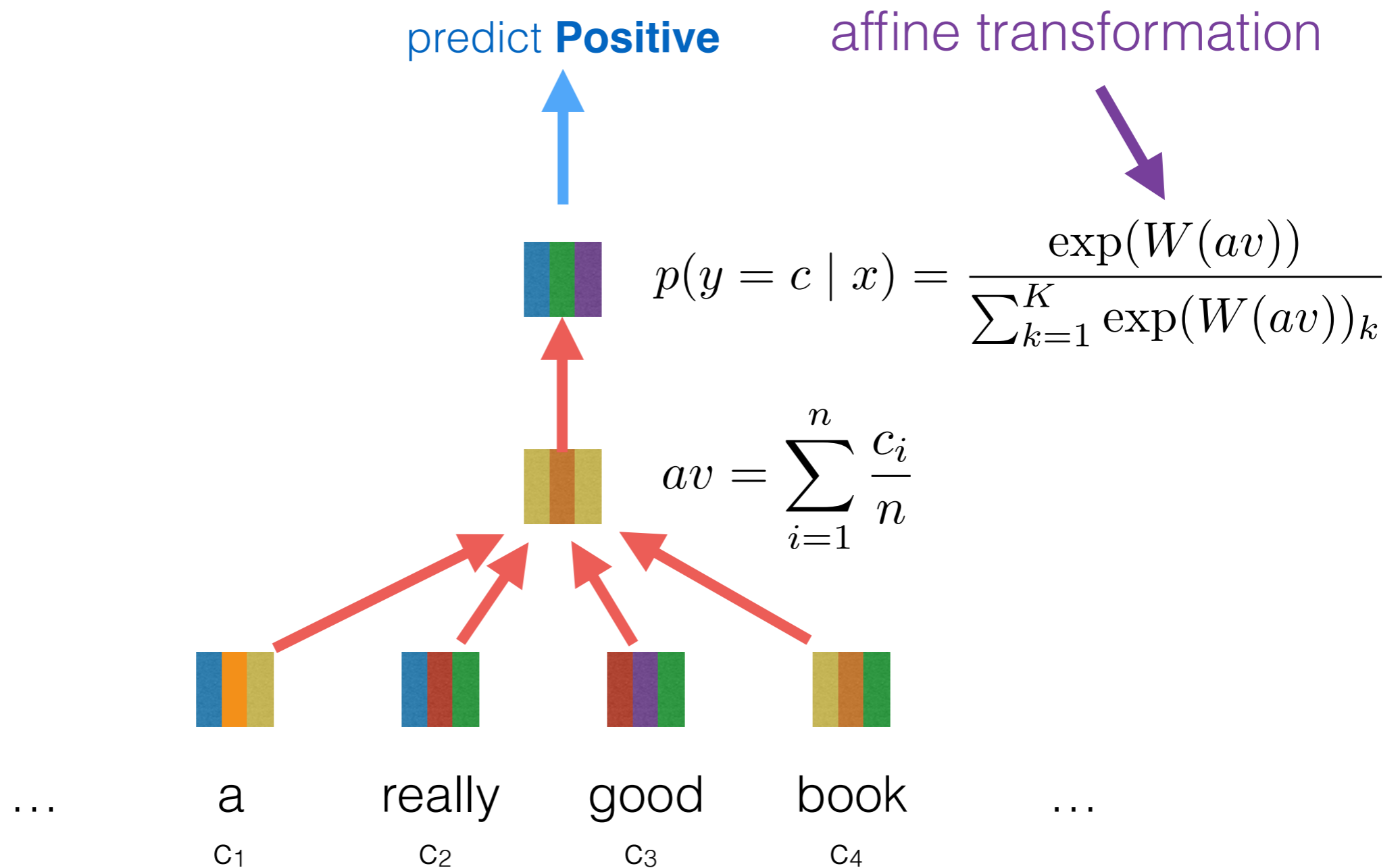
# Demo

- <https://playground.tensorflow.org/>

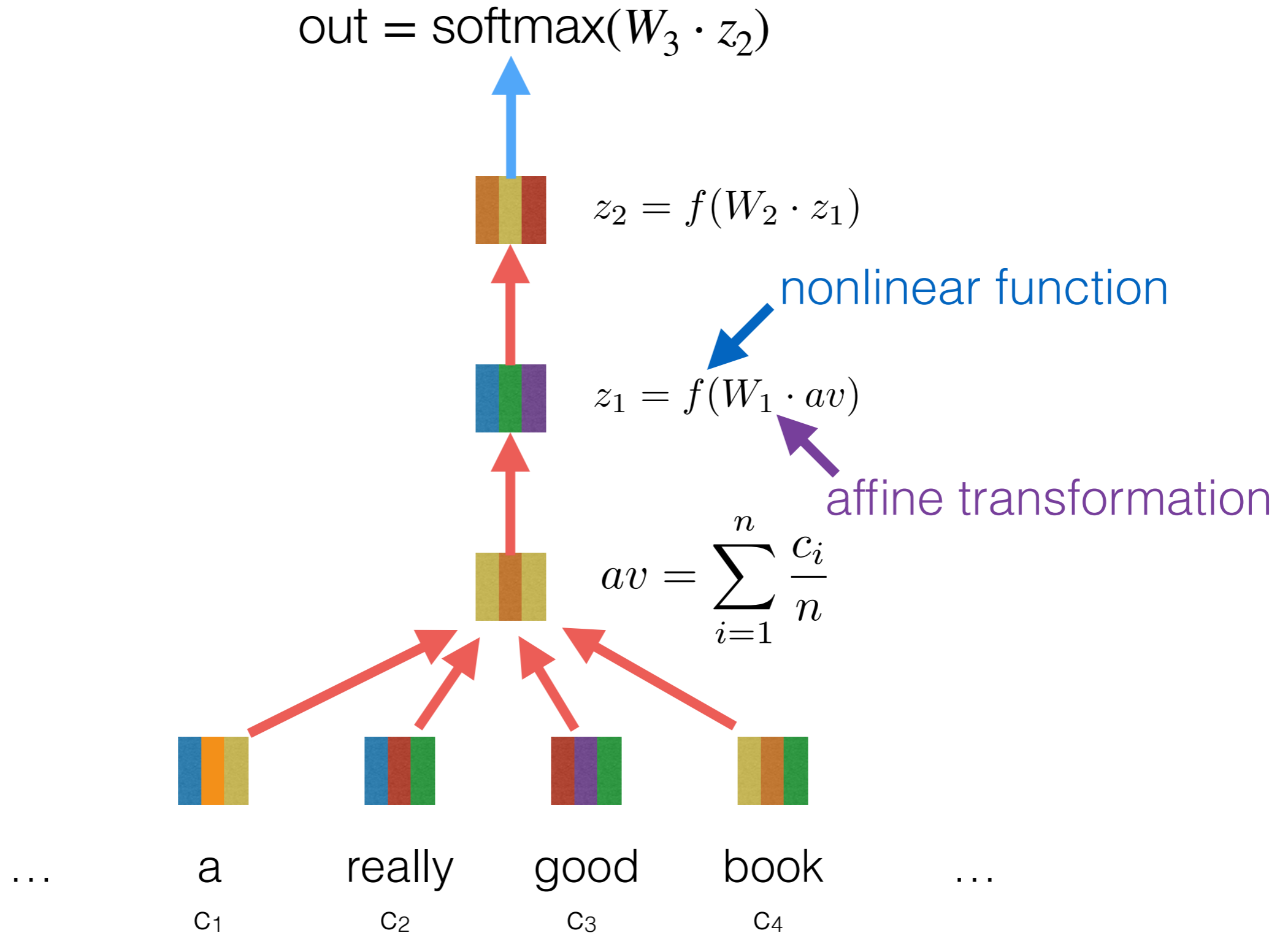


- It's easy to create different neural network architectures, and execute gradient descent learning for arbitrary networks, via *backpropagation*
- e.g. the PyTorch library for Python
- Illustration: deep averaging models for text classification

# “bag of embeddings”

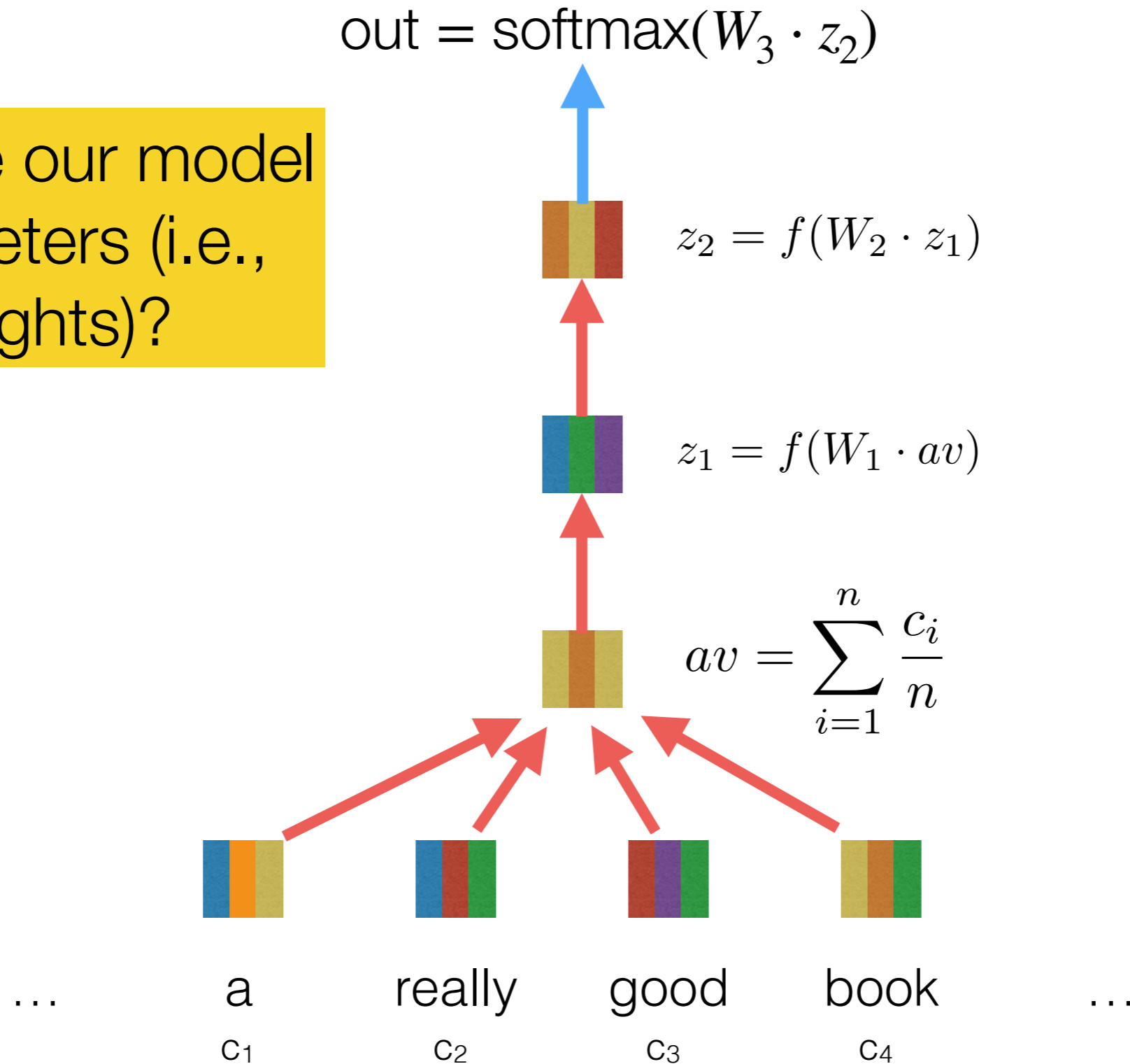


# deep averaging networks



# deep averaging networks

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

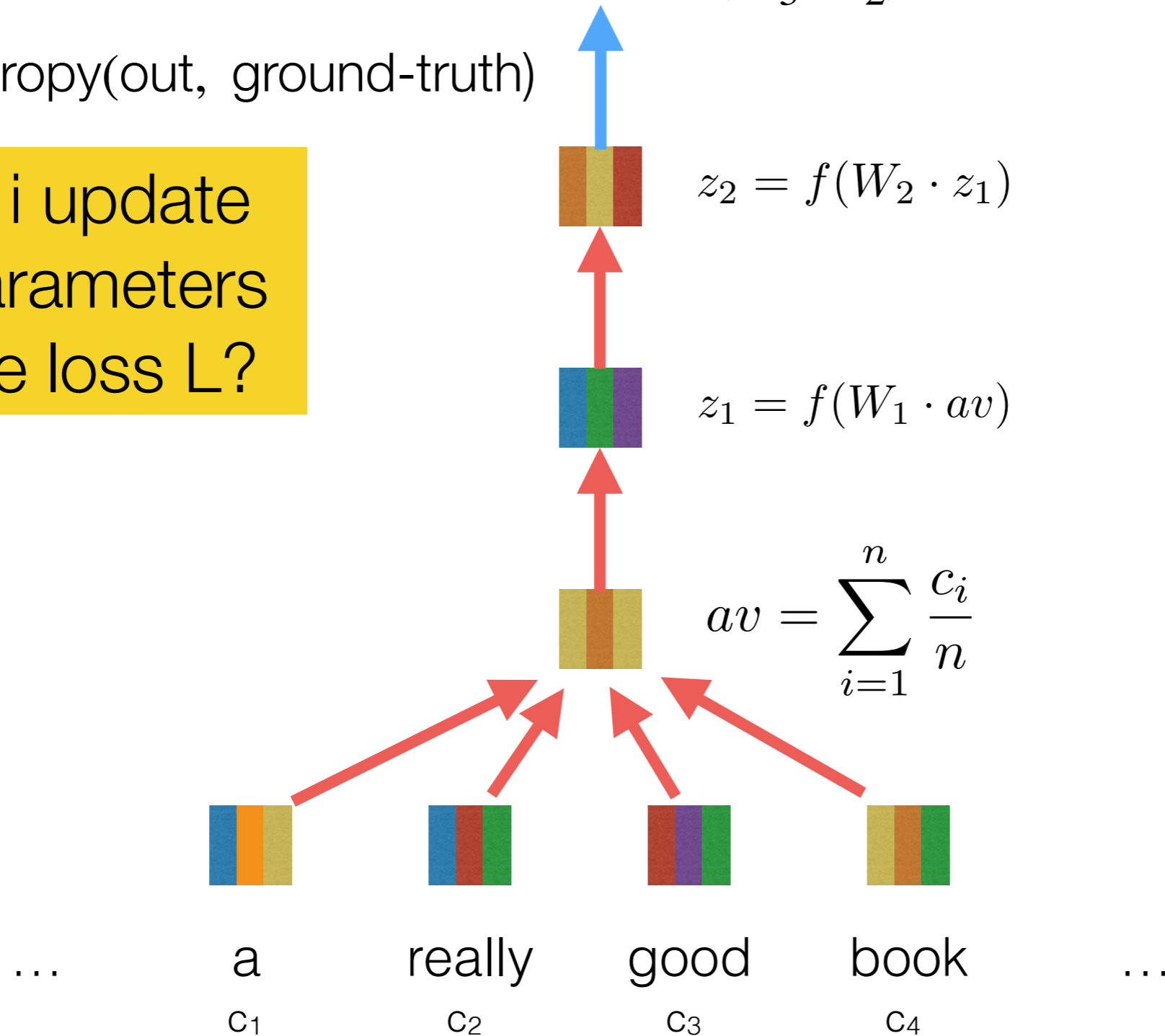


# deep averaging networks

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

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

how do i update  
these parameters  
given the loss L?





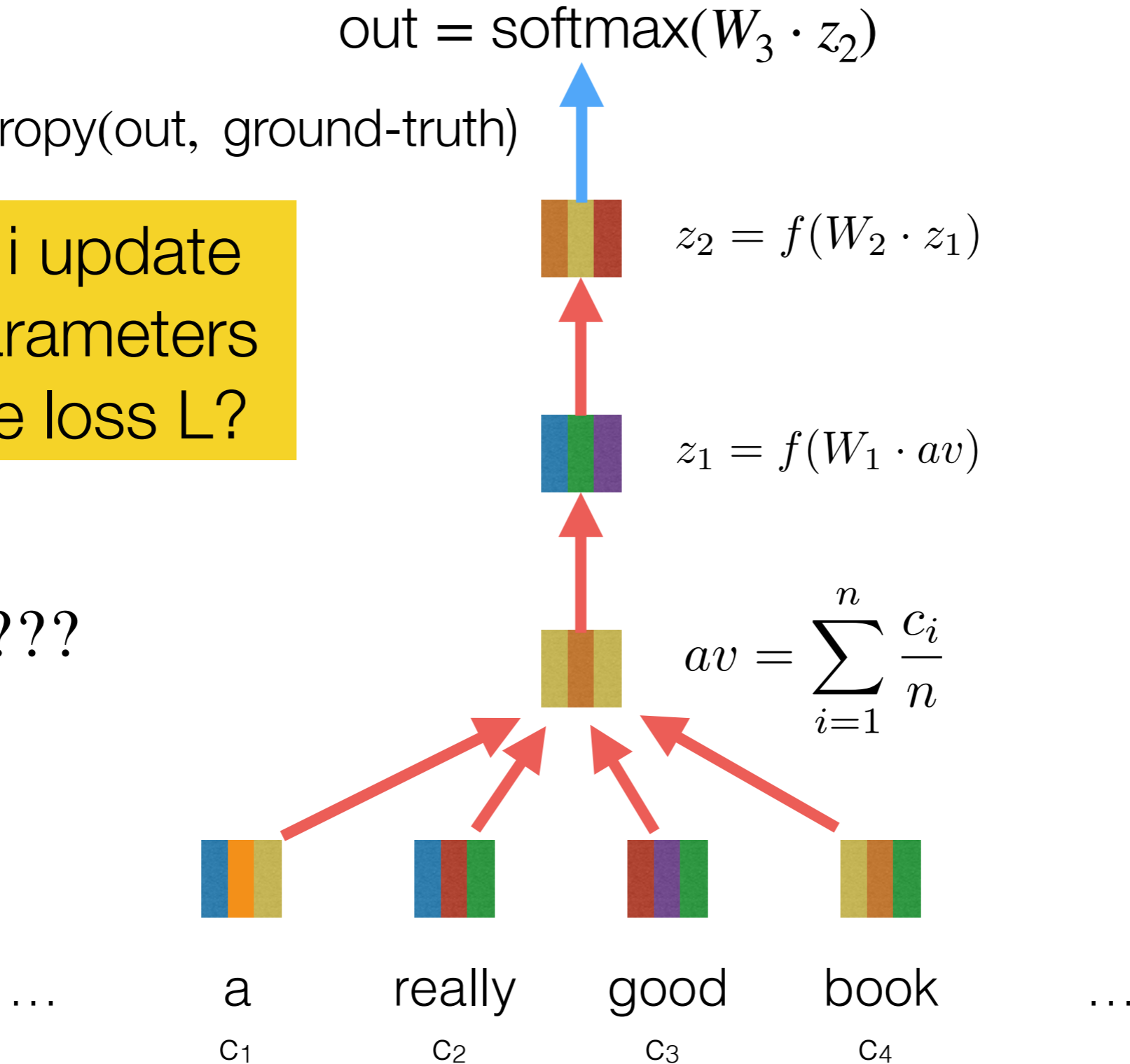
# deep averaging networks

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

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

how do i update these parameters given the loss L?

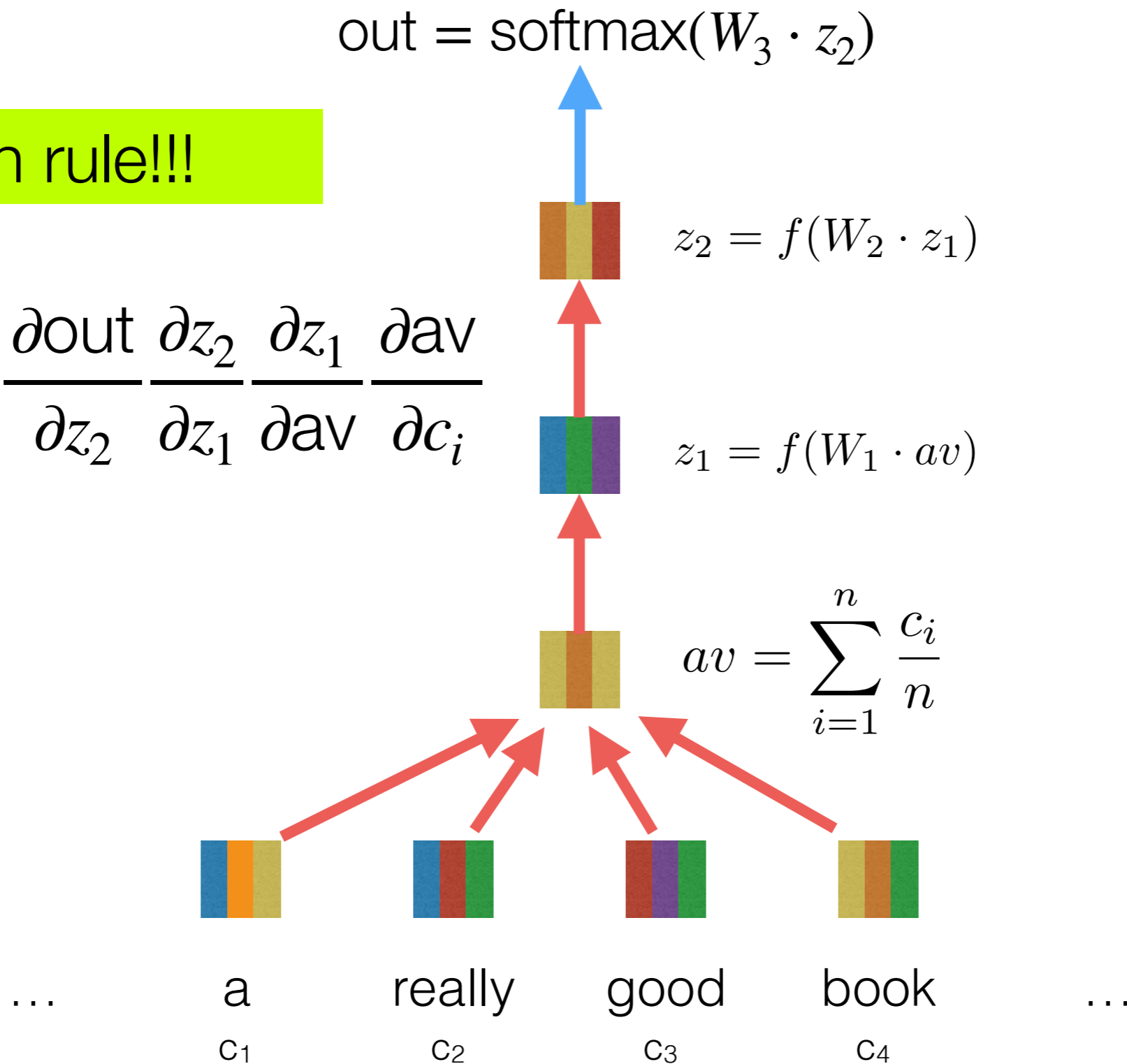
$$\frac{\partial L}{\partial c_i} = ???$$



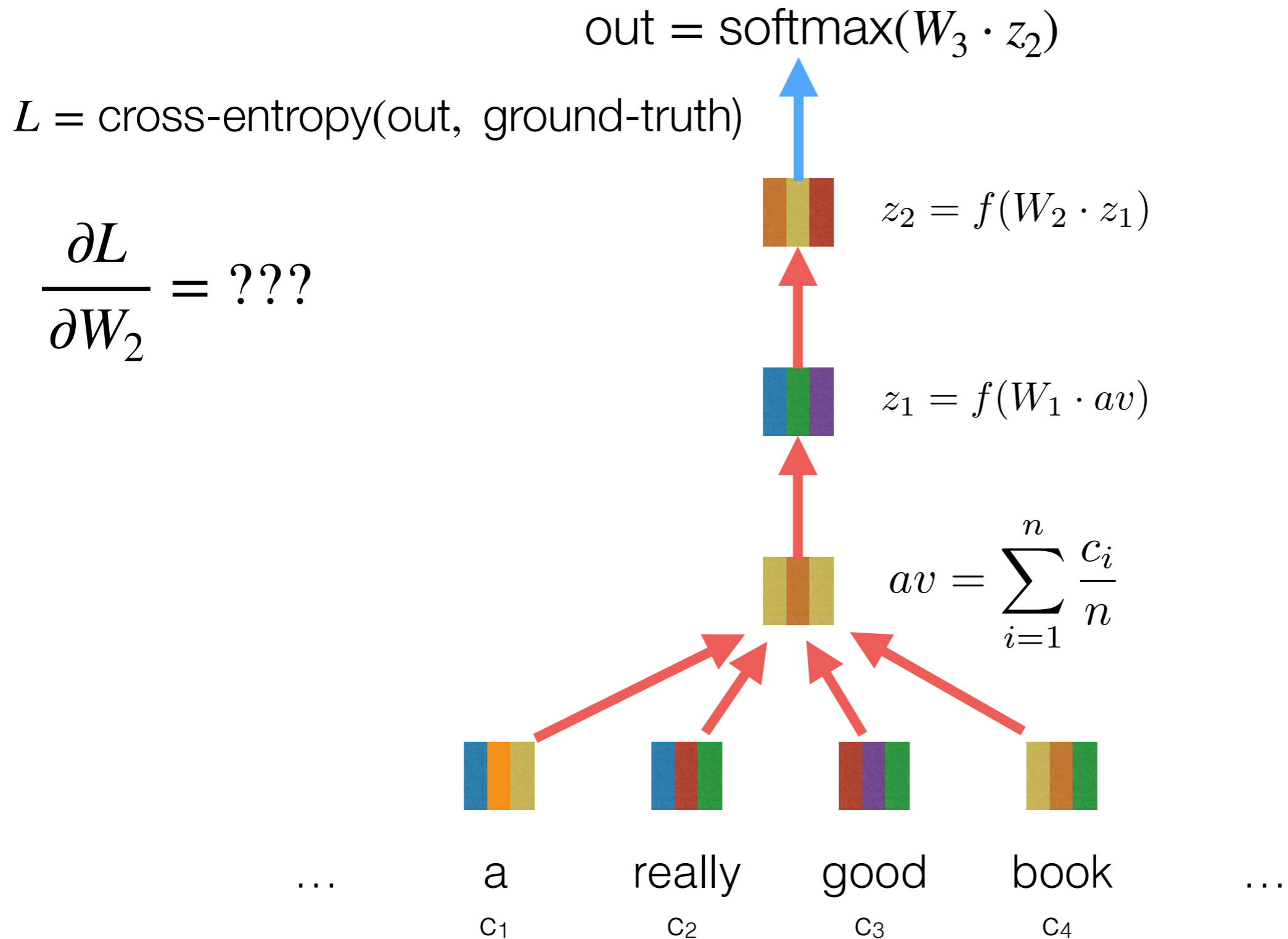
# deep averaging networks

chain rule!!!

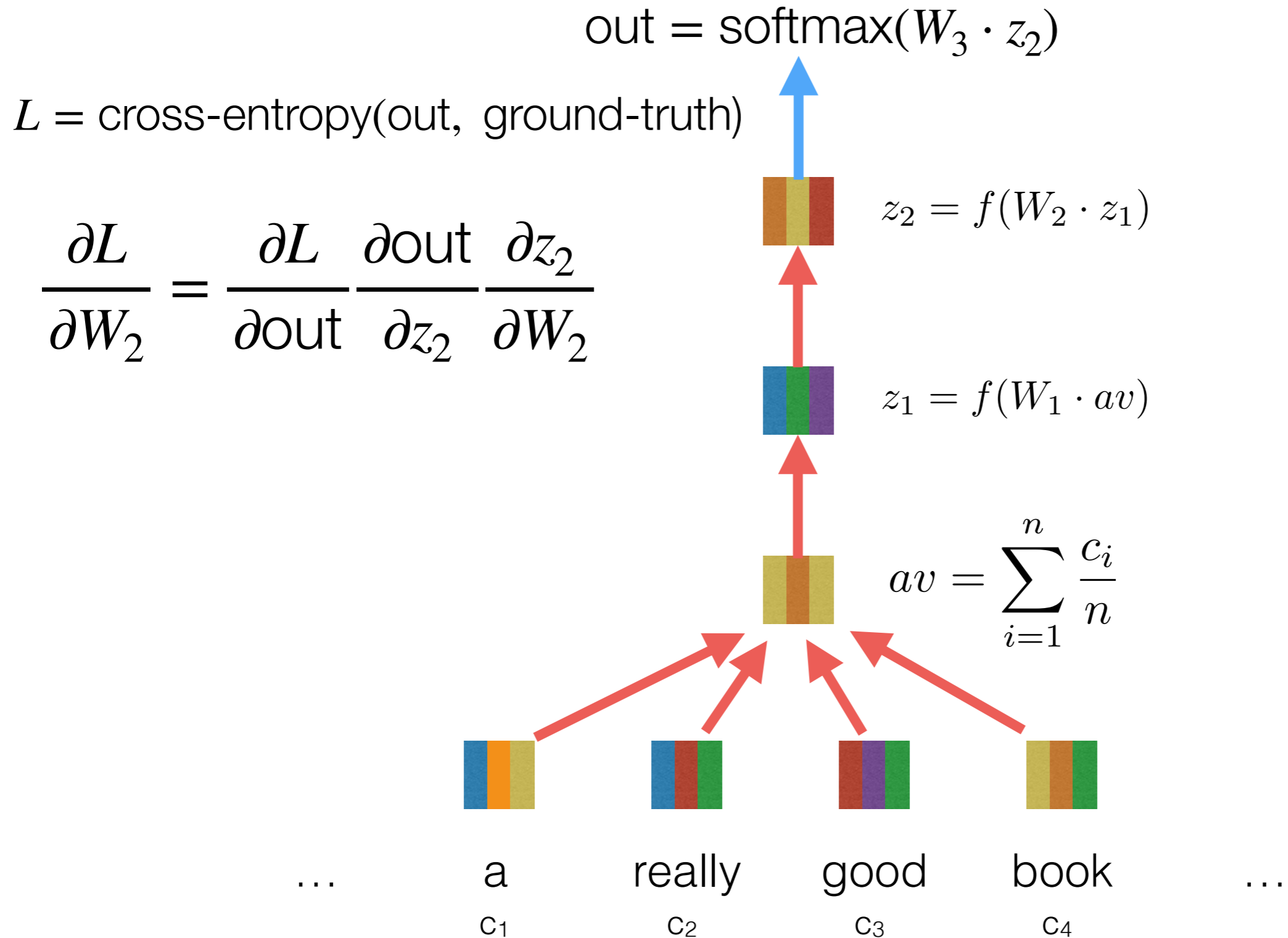
$$\frac{\partial L}{\partial c_i} = \frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2} \frac{\partial z_2}{\partial z_1} \frac{\partial z_1}{\partial av} \frac{\partial av}{\partial c_i}$$



# deep averaging networks



# deep averaging networks



# 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!

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

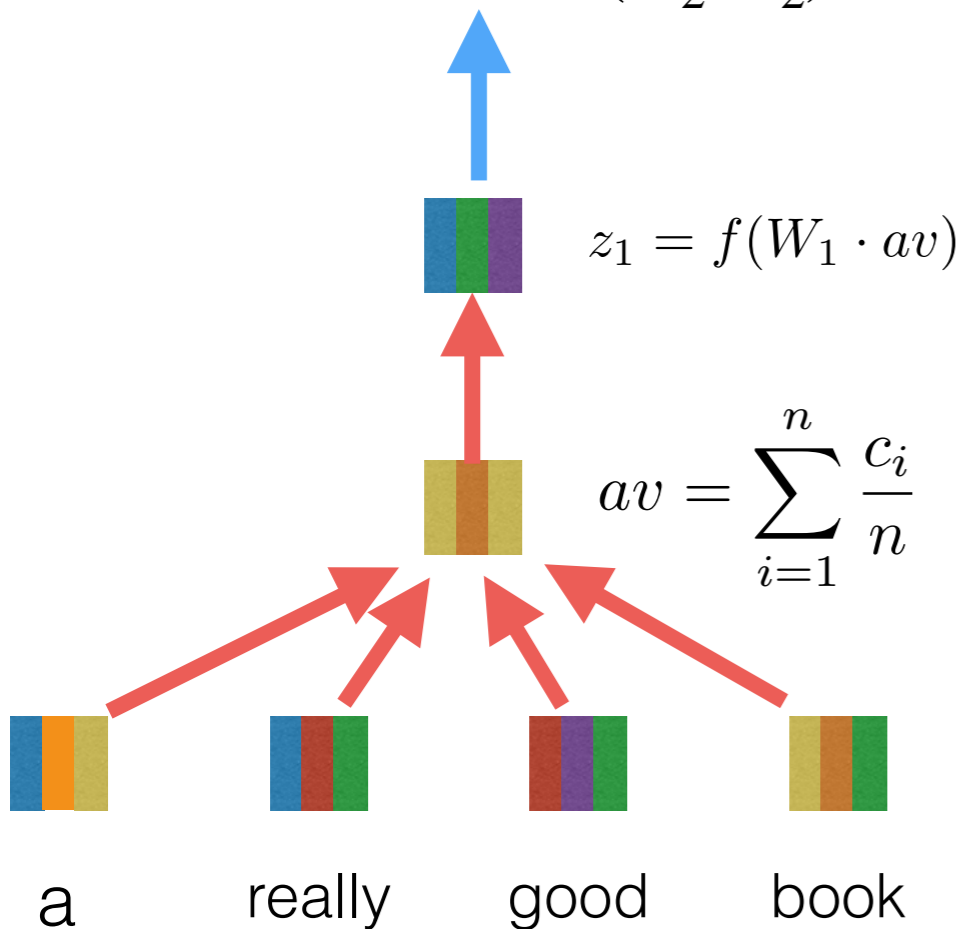
$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \text{out}} \frac{\partial \text{out}}{\partial z_2} \frac{\partial z_2}{\partial W_2}$$

# deep learning frameworks make building NNs super easy!

$$\text{out} = \text{softmax}(W_2 \cdot z_2)$$

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

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



## 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()
```

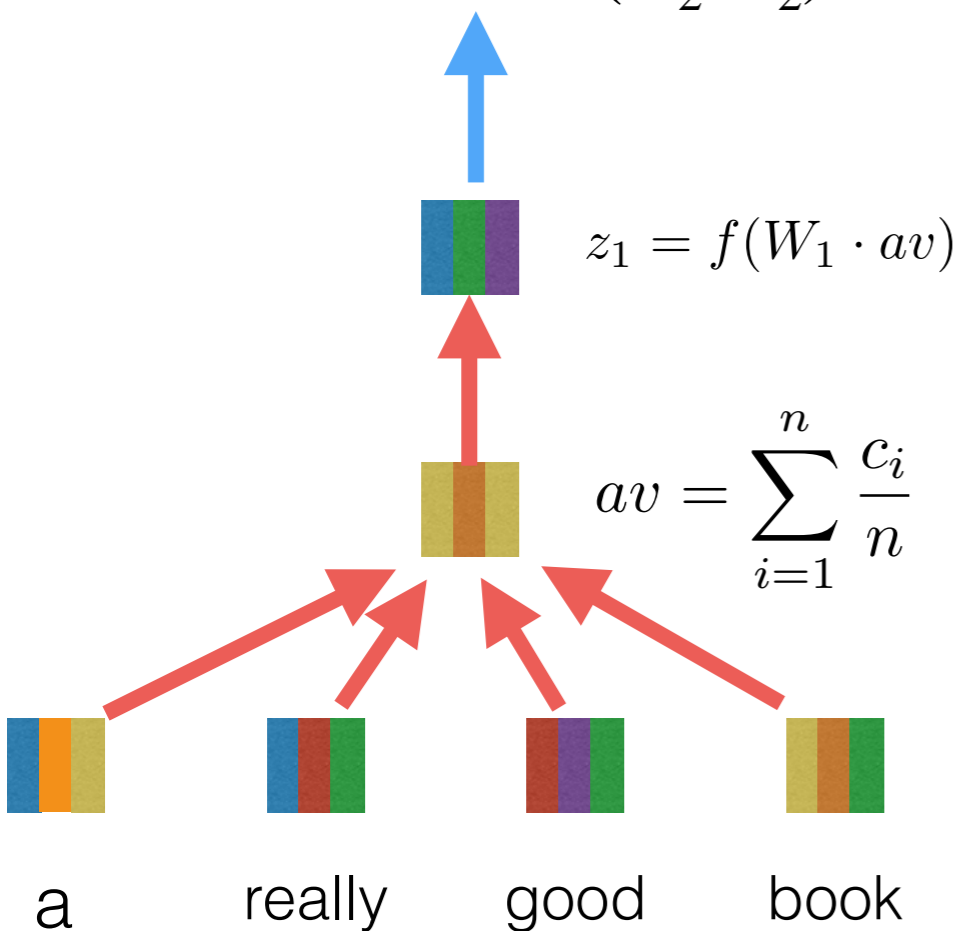
# deep learning frameworks make building NNs super easy!

do a forward pass to compute prediction

$$\text{out} = \text{softmax}(W_2 \cdot z_2)$$

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

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



```
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
```

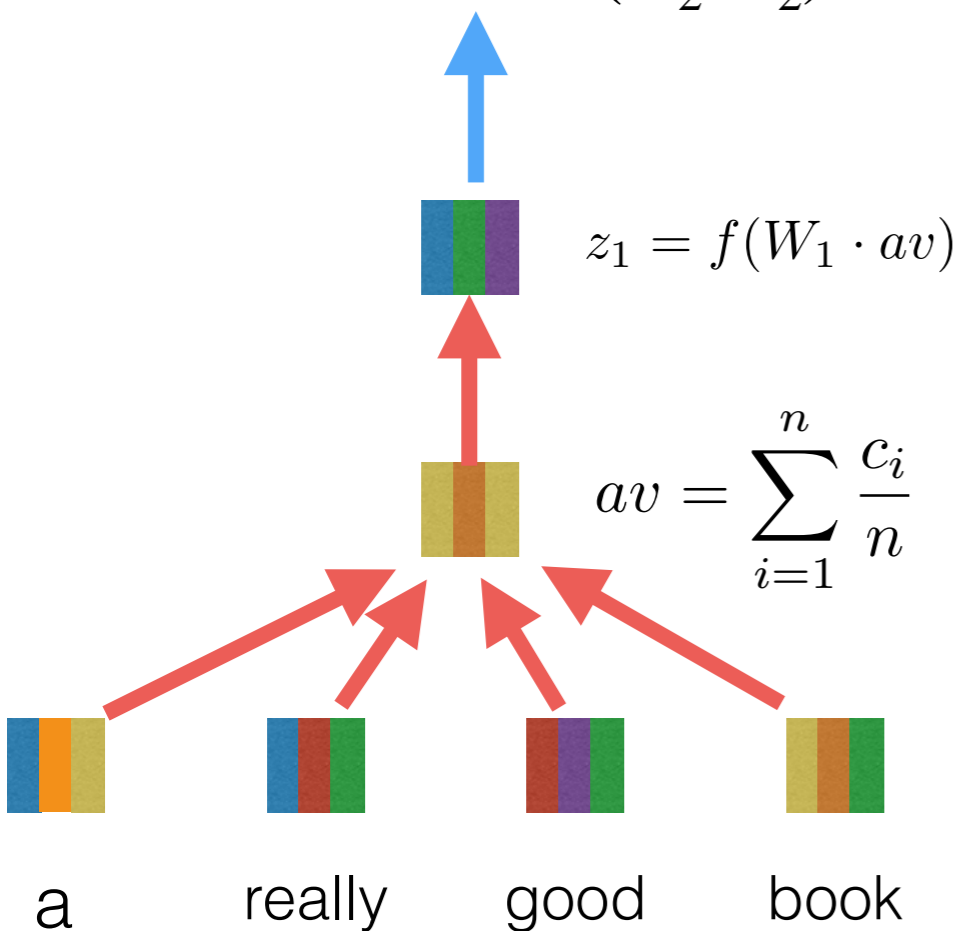
# deep learning frameworks make building NNs super easy!

do a backward pass to update weights

$$\text{out} = \text{softmax}(W_2 \cdot z_2)$$

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

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



```
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()
```

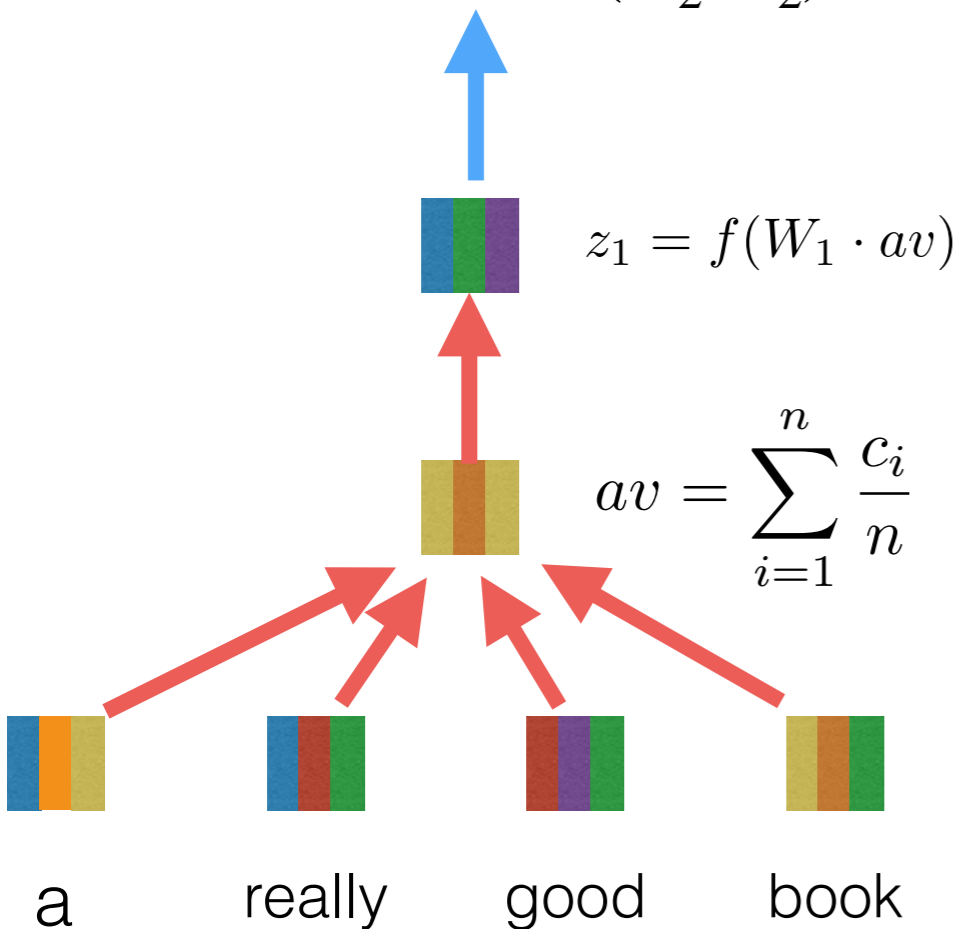


# deep learning frameworks make building NNs super easy!

$$\text{out} = \text{softmax}(W_2 \cdot z_2)$$

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

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



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!

# How NN models are used

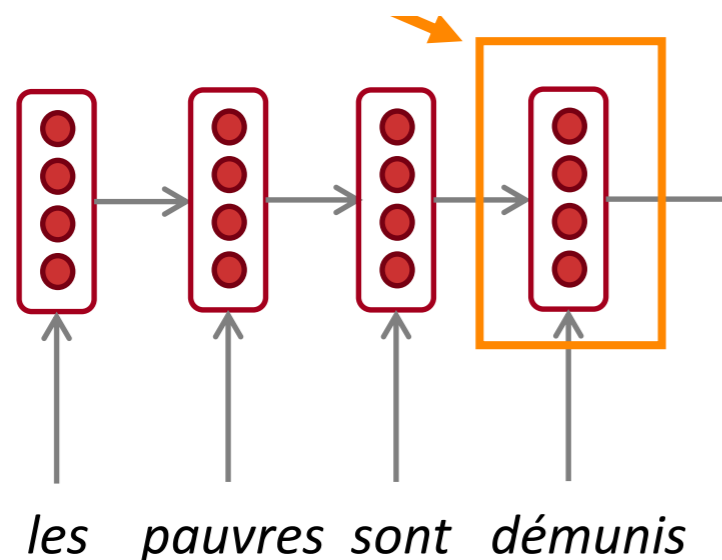
- For classification or tagging:
  - Direct supervised training
  - Pretrain then fine-tune pipeline
- Or: directly use an NN LM for generation

# NN architectures

- We need neural network models that can process token-by-token
- Major components
  - Recurrent neural networks (RNNs)
  - Attention mechanism (softmax over tokens)
  - Self-attention ("Transformers"; next lecture)
    - This is basically the entire model behind BERT and GPT, the best general-purpose NN NLP models today!

# Recurrent neural networks

- Idea: beyond the original word embedding, every token has its own hidden state vector  $h_t$ , influenced by the previous state!



- Many applications
  - Next word prediction
  - Text classification
  - Translation...

# Character LMs comparison: RNN (LSTM) vs. N-Gram

PANDARUS:

Alas, I think he shall be come approached and the day  
When little strain would be attain'd into being never fed,  
And who is but a chain and subjects of his death,  
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,  
Breaking and strongly should be buried, when I perish  
The earth and thoughts of many states.

First Citizen:

Nay, then, that was hers,  
It speaks against your other service:  
But since the  
youth of the circumstance be spoken:  
Your uncle and one Baptista's daughter.

SEBASTIAN:

Do I stand till the break off.

BIRON:

Hide thy head.

VENTIDIUS:

He purposeth to Athens: whither, with the vow  
I made to handle you.

# Structure awareness (one particular RNN hidden state)

Cell sensitive to position in line:

```
The sole importance of the crossing of the Berezina lies in the fact that it plainly and indubitably proved the fallacy of all the plans for cutting off the enemy's retreat and the soundness of the only possible line of action--the one Kutuzov and the general mass of the army demanded--namely, simply to follow the enemy up. The French crowd fled at a continually increasing speed and all its energy was directed to reaching its goal. It fled like a wounded animal and it was impossible to block its path. This was shown not so much by the arrangements it made for crossing as by what took place at the bridges. When the bridges broke down, unarmed soldiers, people from Moscow and women with children who were with the French transport, all--carried on by vis inertiae--pressed forward into boats and into the ice-covered water and did not, surrender.
```

Cell that turns on inside quotes:

```
"You mean to imply that I have nothing to eat out of.... On the contrary, I can supply you with everything even if you want to give dinner parties," warmly replied Chichagov, who tried by every word he spoke to prove his own rectitude and therefore imagined Kutuzov to be animated by the same desire.
```

```
Kutuzov, shrugging his shoulders, replied with his subtle penetrating smile: "I meant merely to say what I said."
```

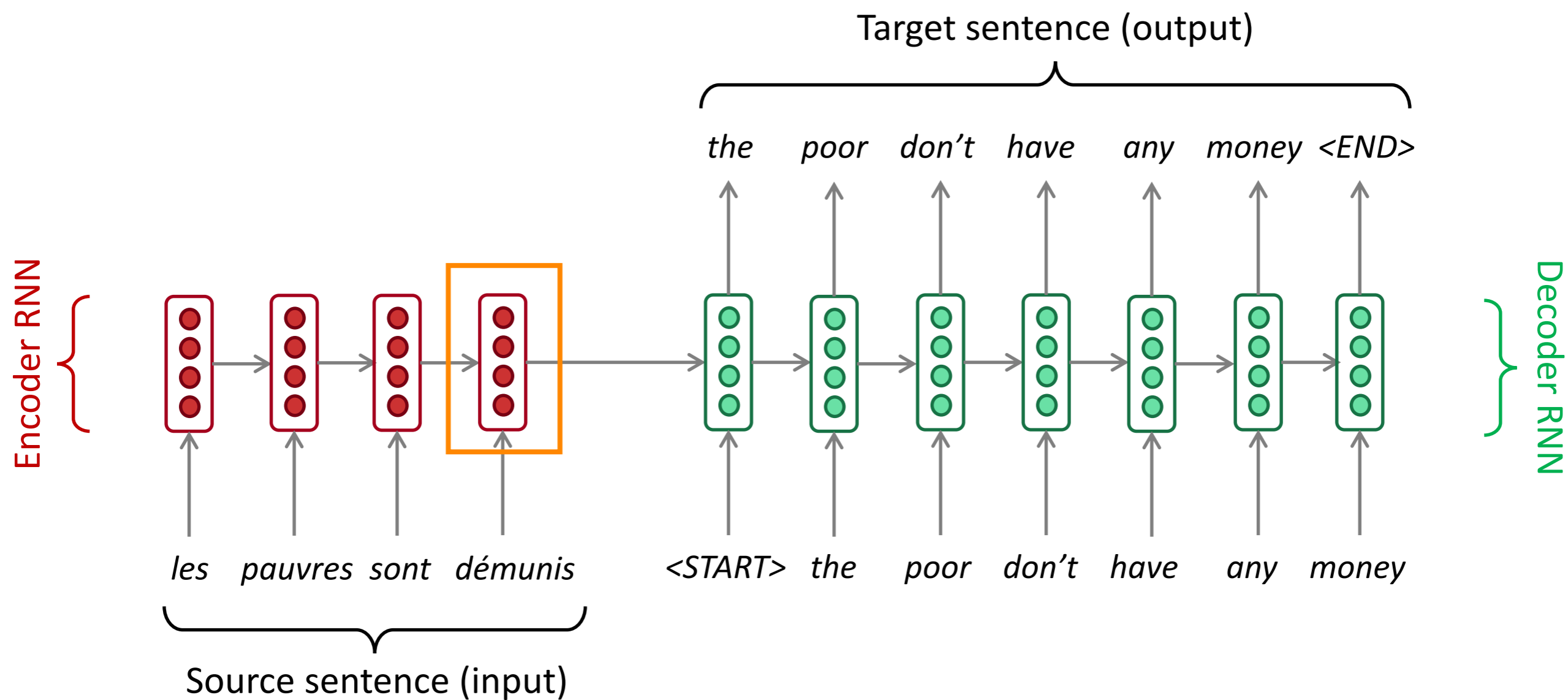
Cell that robustly activates inside if statements:

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
                           siginfo_t *info)
{
    int sig = next_signal(pending, mask);
    if (sig) {
        if (current->notifier) {
            if (sigismember(current->notifier_mask, sig)) {
                if (!(current->notifier)(current->notifier_data)) {
                    clear_thread_flag(TIF_SIGPENDING);
                    return 0;
                }
            }
        }
        collect_signal(sig, pending, info);
    }
    return sig;
}
```

A large portion of cells are not easily interpretable. Here is a typical example:

```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
    char *str;
    if (!*bufp || (len == 0) || (len > *remain))
        return ERR_PTR(-EINVAL);
    /* Of the currently implemented string fields, PATH_MAX
     * defines the longest valid length.
     */
}
```

# Sequence-to-sequence: the bottleneck problem



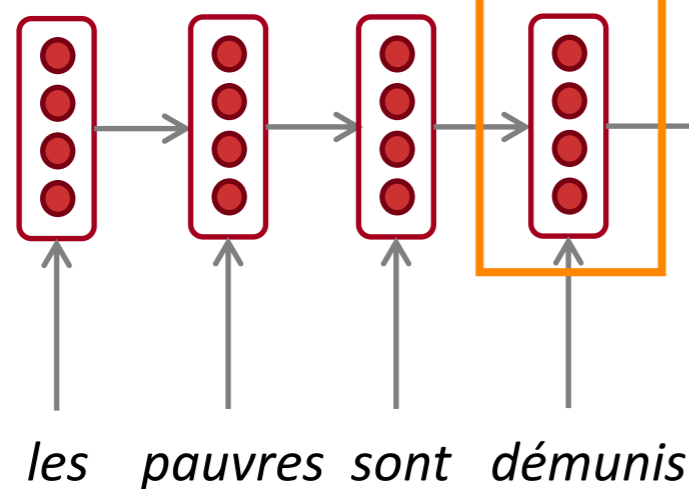
# Sequence-to-sequence: the bottleneck problem

Encoding of the source sentence.

This needs to capture *all information* about the source sentence.

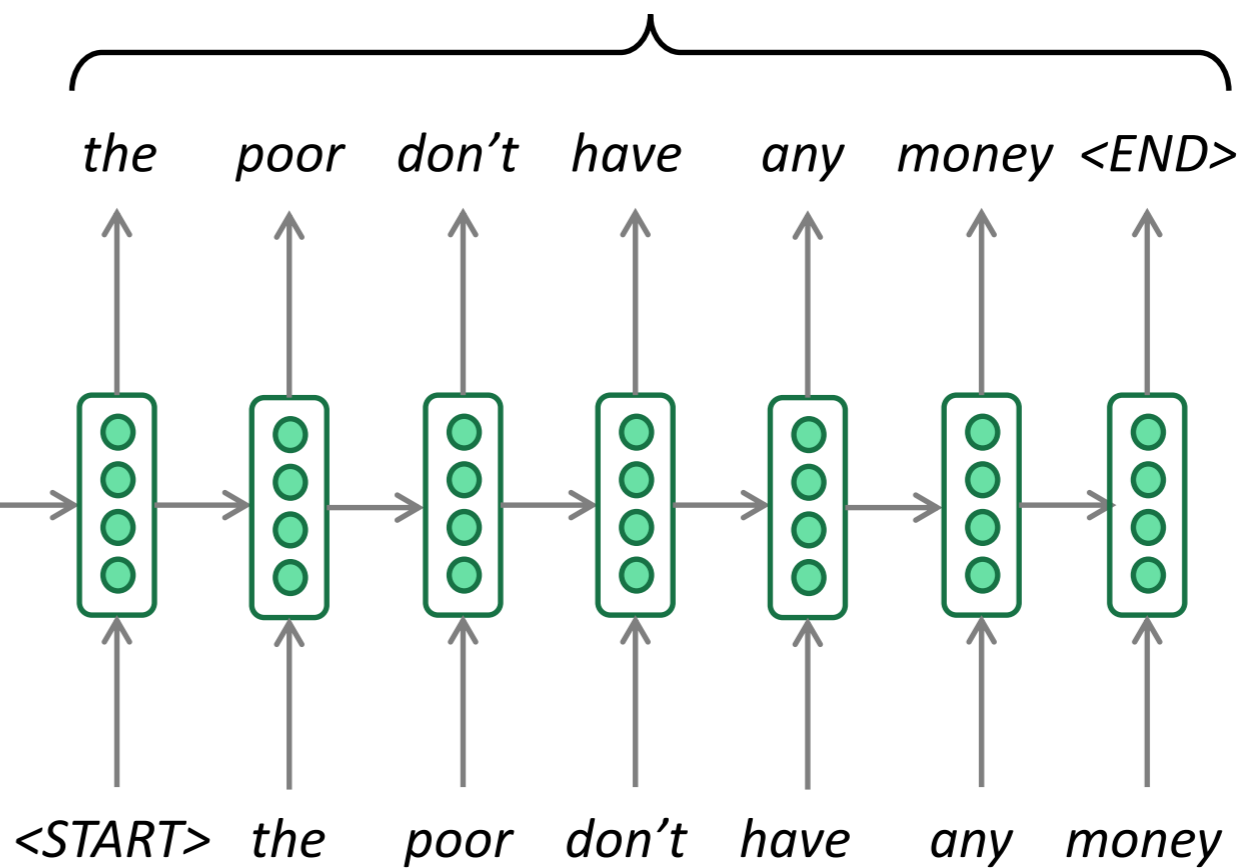
Information bottleneck!

Encoder RNN



Source sentence (input)

Target sentence (output)



Decoder RNN