CS388: Natural Language Processing

Lecture 6: NN Implementation

Greg Durrett

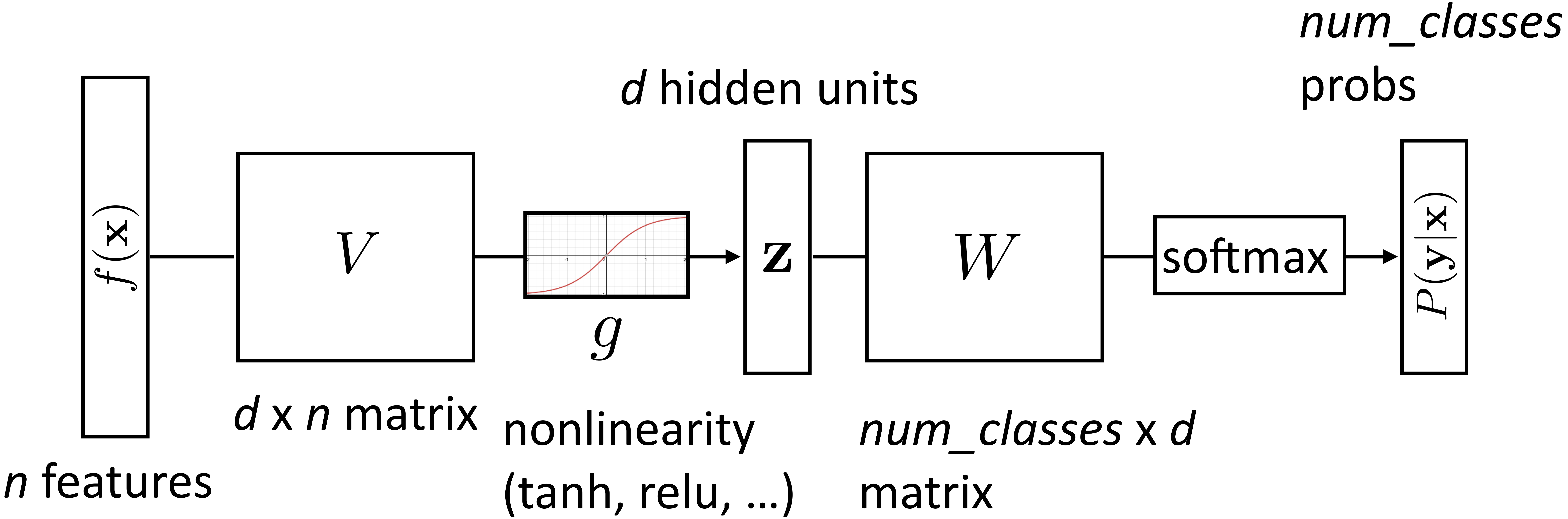
TEXAS
The University of Texas at Austin

Announcements

- A1 due today at midnight
- A2 out at midnight

Recall: Feedforward NNs

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



Recall: Training Feedforward NNs

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$

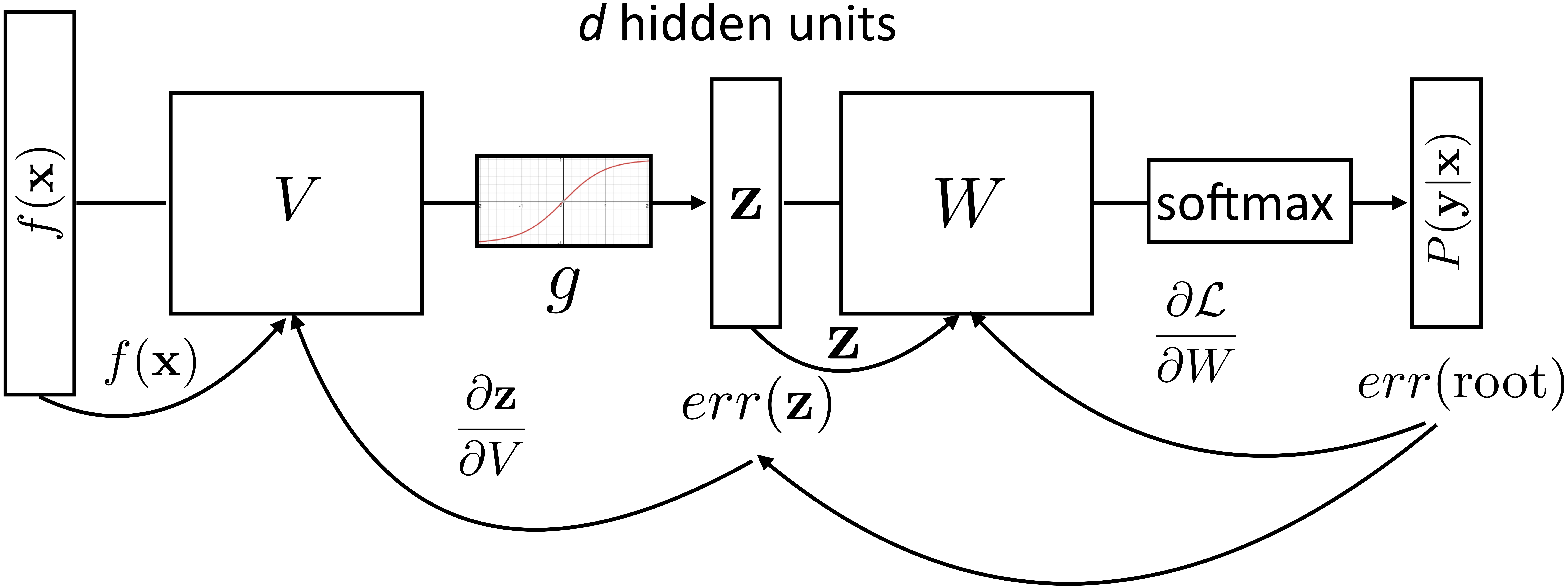
- Maximize log likelihood of training data. For one point:

$$\mathcal{L}(\mathbf{x}, i^*) = \log P(y = i^*|\mathbf{x}) = \log\left(\mathrm{softmax}(W\mathbf{z}) \cdot e_{i^*}\right)$$

- How to compute the gradient with respect to *W* and *V*?

Recall: Backpropagation

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



This Lecture

- Neural net implementation / PyTorch 101
- Neural net training tips
- Deep averaging networks

Implementing Neural Networks: PyTorch 101

Computation Graphs

- Computing gradients is hard!
- Automatic differentiation: instrument code to keep track of derivatives

```
y = x * x   --codegen-->   (y,dy) = (x * x, 2 * x * dx)
```

- Computation is now something we need to reason about symbolically; use a library like PyTorch (or Tensorflow)
- **Ensuing code examples are on the course website: ffnn_example.py under "Readings"**

PyTorch

- Framework for defining computations that provides easy access to derivatives
- Module: defines a neural network (can use wrap other modules which implement predefined layers)
- If forward() uses crazy stuff, you have to write backward yourself

```
torch.nn.Module

    # Takes an example x and computes result
    forward(x):
        ...
    # Computes gradient after forward() is called
    backward(): # produced automatically
        ...
```

Computation Graphs in Pytorch

- Define forward pass for $P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$

```
class FFNN(nn.Module):
    def __init__(self, input_size, hidden_size, out_size):
        super(FFNN, self).__init__()
        self.V = nn.Linear(input_size, hidden_size)
        self.g = nn.Tanh() # or nn.ReLU(), sigmoid()...
        self.W = nn.Linear(hidden_size, out_size)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

(syntactic sugar for forward)

Input to Network

- Whatever you define with torch.nn needs its input as some sort of tensor, whether it's integer word indices or real-valued vectors

```
def form_input(x) -> torch.Tensor:
  # Index words/embed words/etc.
  return torch.from_numpy(x).float()
```

- torch.Tensor is a different datastructure from a numpy array, but you can translate back and forth fairly easily
- Note that **translating out of PyTorch will break backpropagation**; don't do this inside your Module

Training and Optimization

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$

one-hot vector
of the label
(e.g., [0, 1, 0])

```
ffnn = FFNN(inp, hid, out)
optimizer = optim.Adam(ffnn.parameters(), lr=lr)
for epoch in range(0, num_epochs):
    for (input, gold_label) in training_data:
        ffnn.zero_grad() # clear gradient variables
        probs = ffnn.forward(input)
        loss = torch.neg(torch.log(probs)).dot(gold_label)
        loss.backward()
        optimizer.step()
```

negative log-likelihood of correct answer

Initialization in Pytorch

```
class FFNN(nn.Module):
    def __init__(self, inp, hid, out):
        super(FFNN, self).__init__()
        self.V = nn.Linear(inp, hid)
        self.g = nn.Tanh()
        self.W = nn.Linear(hid, out)
        self.softmax = nn.Softmax(dim=0)
        nn.init.uniform(self.V.weight)
```

- Initializing to a nonzero value is critical, more in a bit

Training a Model

Define modules, etc.

Initialize weights and optimizer

For each epoch:

- For each batch of data:
  - Zero out gradient
  - Compute loss on batch
  - Autograd to compute gradients and take step on optimizer
- [Optional: check performance on dev set to identify overfitting]

Run on dev/test set

Batching and Optimization
(blackboard)

Batching

- Batching: processing multiple examples in parallel (for training or test), gives speedups due to more efficient matrix operations
- Need to make the computation graph process a batch at the same time

```
# input is [batch_size, num_feats]
# gold_label is [batch_size, num_classes]
def make_update(input, gold_label)
    ...
    probs = ffnn.forward(input) # [batch_size, num_classes]
    loss = torch.sum(torch.neg(torch.log(probs)).dot(gold_label))
    ...
```

- Batch sizes range from 1-64 or so (depending on GPU memory, etc.)

Optimization Takeaways

- Need to initialize to values that aren't 0 but aren't too large
  - Can do random uniform / normal initialization with appropriate scale; also fancier initializers (Xavier Glorot initializer, Kaiming He) to match variances across layers
- Use Adam as your optimizer
- Consider adding dropout layers (at input or hidden layers; never at output). Typically 0.2 - 0.5 are good ranges for dropout probability

DANs

Word Embeddings

- Currently we think of words as "one-hot" vectors

  the = [1, 0, 0, 0, 0, 0, …]

  good = [0, 0, 0, 1, 0, 0, …]

  great = [0, 0, 0, 0, 0, 1, …]

- *good* and *great* seem as dissimilar as *good* and *the*

- Neural networks are built to learn sophisticated nonlinear functions of continuous inputs; our inputs are weird and discrete

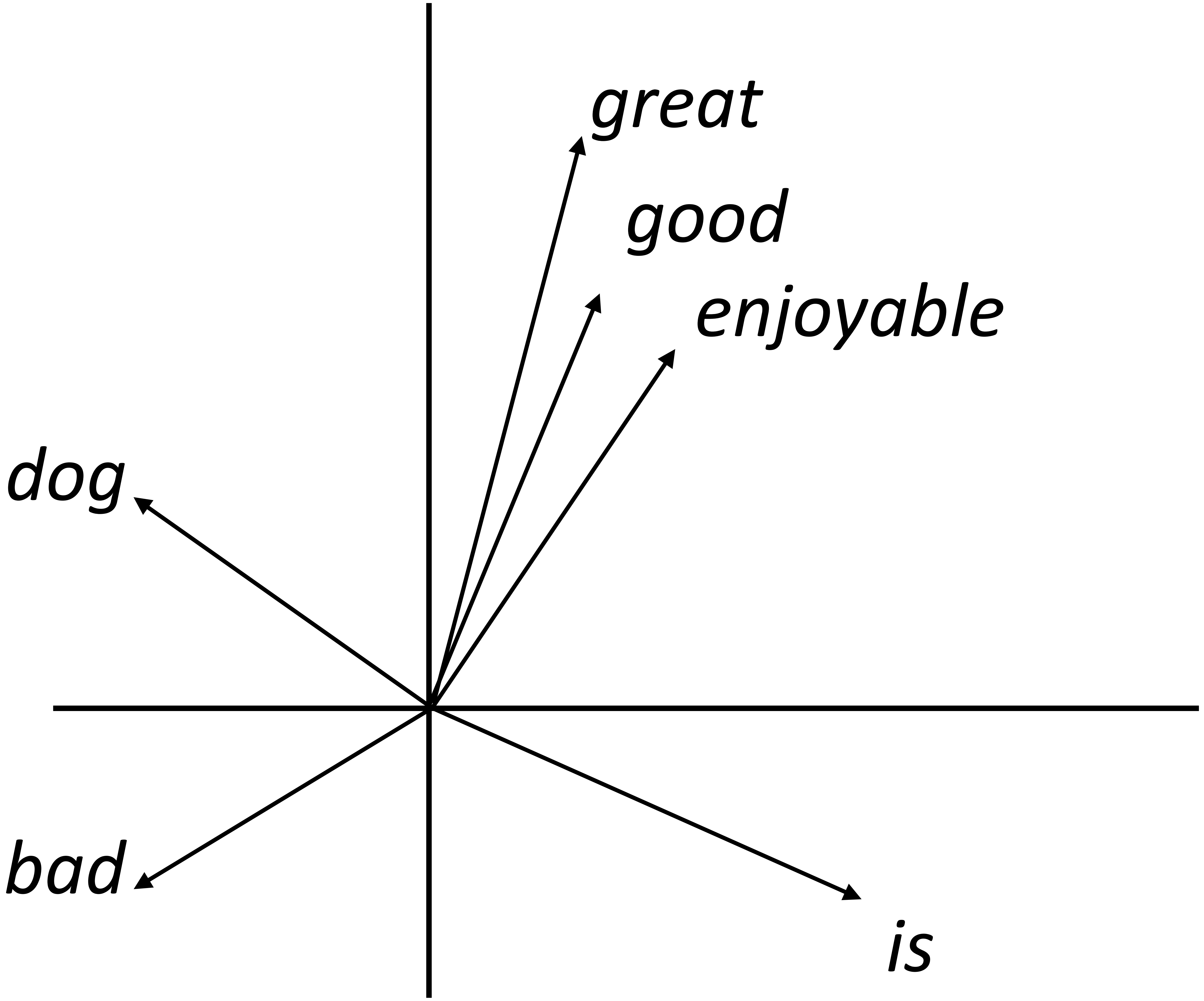
Word Embeddings

- Want a vector space where similar words have similar embeddings

  great ≈ good

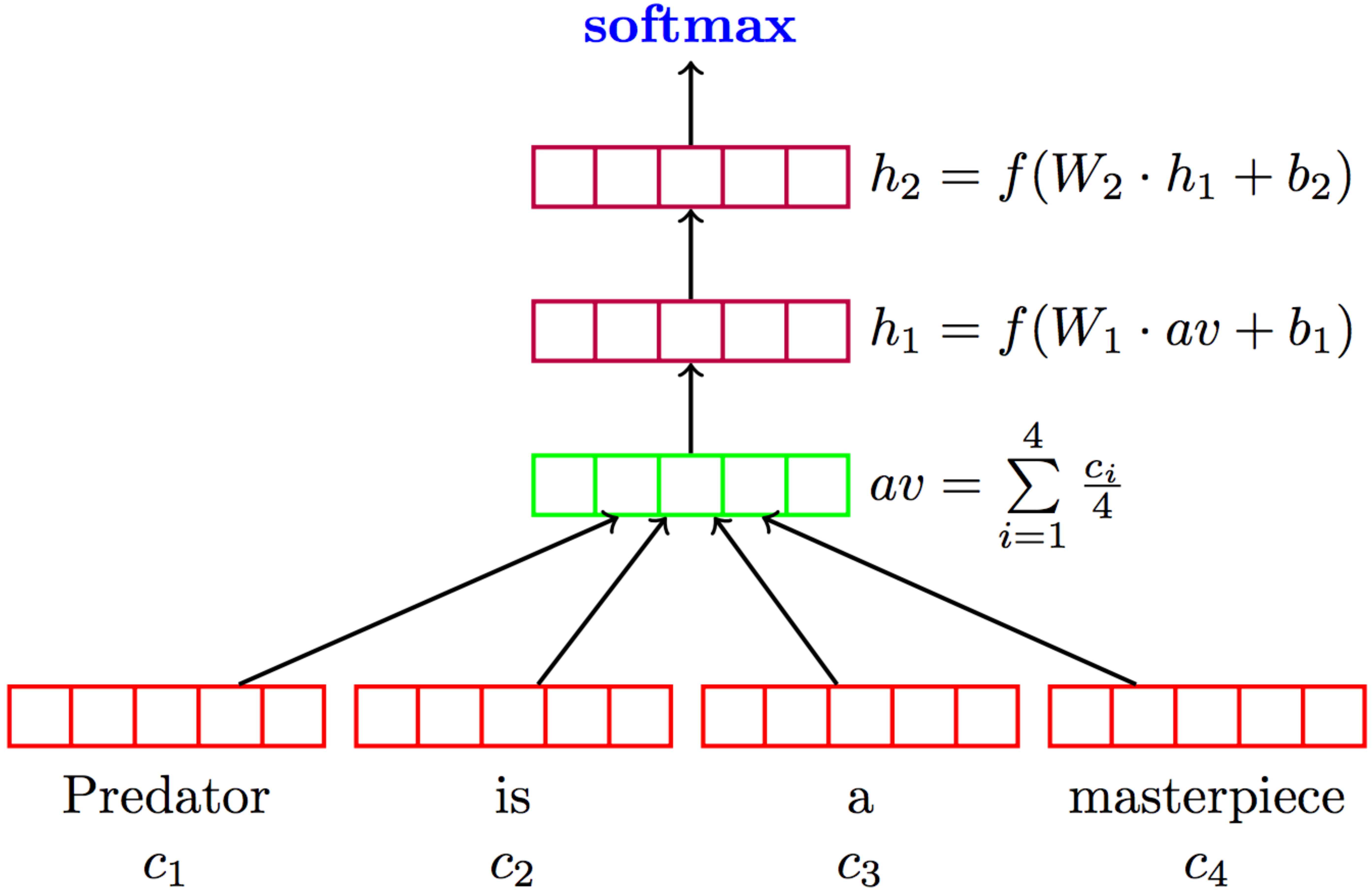
- Next lecture: come up with a way to produce these embeddings

- For each word, want "medium" dimensional vector (50-300 dims) representing it



Deep Averaging Networks

- Deep Averaging Networks: feedforward neural network on average of word embeddings from input

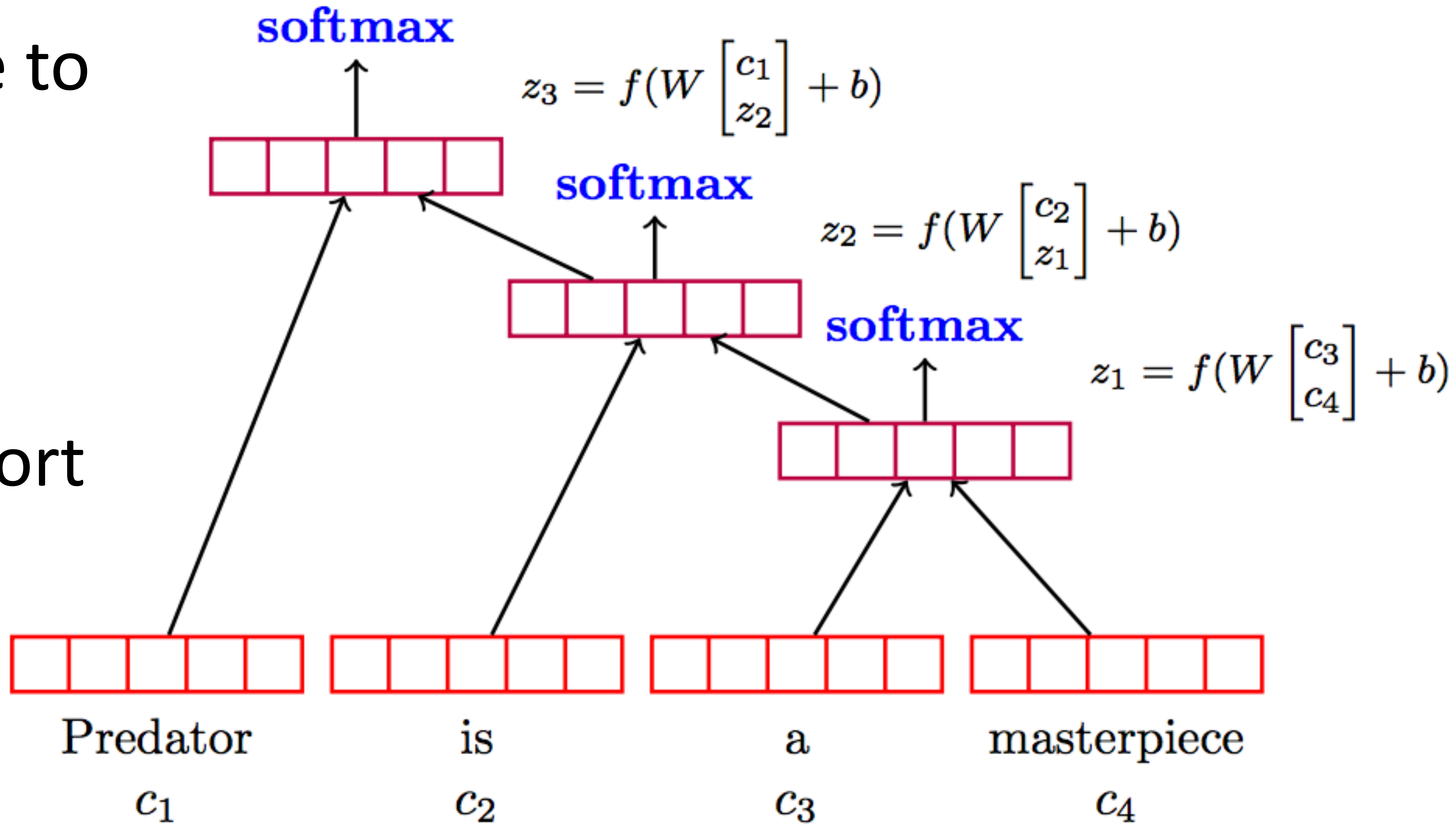


Iyyer et al. (2015)

Deep Averaging Networks

- Widely-held view: need to model syntactic structure to represent language

- Surprising that averaging can work as well as this sort of composition



Iyyer et al. (2015)

Sentiment Analysis

Model	RT	SST fine	SST bin	IMDB	Time (s)
DAN-ROOT	—	46.9	85.7	—	**31**
DAN-RAND	77.3	45.4	83.2	88.8	136
DAN	80.3	47.7	86.3	89.4	136
NBOW-RAND	76.2	42.3	81.4	88.9	91
NBOW	79.0	43.6	83.6	89.0	91
BiNB	—	41.9	83.1	—	—
NBSVM-bi	79.4	—	—	91.2	—
RecNN*	77.7	43.2	82.4	—	—
RecNTN*	—	45.7	85.4	—	—
DRecNN	—	49.8	86.6	—	431
TreeLSTM	—	**50.6**	86.9	—	—
DCNN*	—	48.5	86.9	89.4	—
PVEC*	—	48.7	87.8	**92.6**	—
CNN-MC	**81.1**	47.4	**88.1**	—	2,452
WRRBM*	—	—	—	89.2	—

No pretrained embeddings

Iyyer et al. (2015)

Bag-of-words

Wang and Manning (2012)

Tree-structured neural networks

Kim (2014)

Deep Averaging Networks

Sentence	DAN	DRecNN	Ground Truth
who knows what exactly godard is on about in this film, but his words and images do n't have to add up to mesmerize you.	positive	positive	positive
it's so good that its relentless, polished wit can withstand not only inept school productions, but even oliver parker's movie adaptation	negative	positive	positive
too bad, but thanks to some lovely comedic moments and several fine performances, it's not a total loss	negative	negative	positive
this movie was not good	negative	negative	negative
this movie was good	positive	positive	positive
this movie was bad	negative	negative	negative
the movie was not bad	negative	negative	positive

- Will return to compositionality with syntax and LSTMs

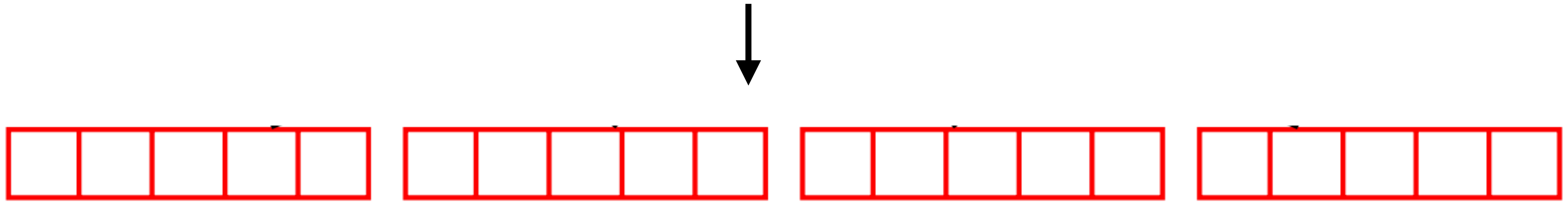
Iyyer et al. (2015)

Word Embeddings in PyTorch

- torch.nn.Embedding: maps vector of indices to matrix of word vectors

Predator is a masterpiece

1820 24 1 2047



- *n* indices => *n* x *d* matrix of *d*-dimensional word embeddings
- *b* x *n* indices => *b* x *n* x *d* tensor of *d*-dimensional word embeddings

Next Time

- Guest lecture
- Next Thursday: word embeddings