

# CS378: Natural Language Processing

## Lecture 6: NN Implementation

Greg Durrett





# Announcements

---

- ▶ Assignment 1 due today
- ▶ Assignment 2 out today, due in two weeks
- ▶ Fairness response due Tuesday (submit on Canvas)
- ▶ Slip days: do not need to notify me
- ▶ Greg's OHs shifted 15 minutes later today



# Recap

---



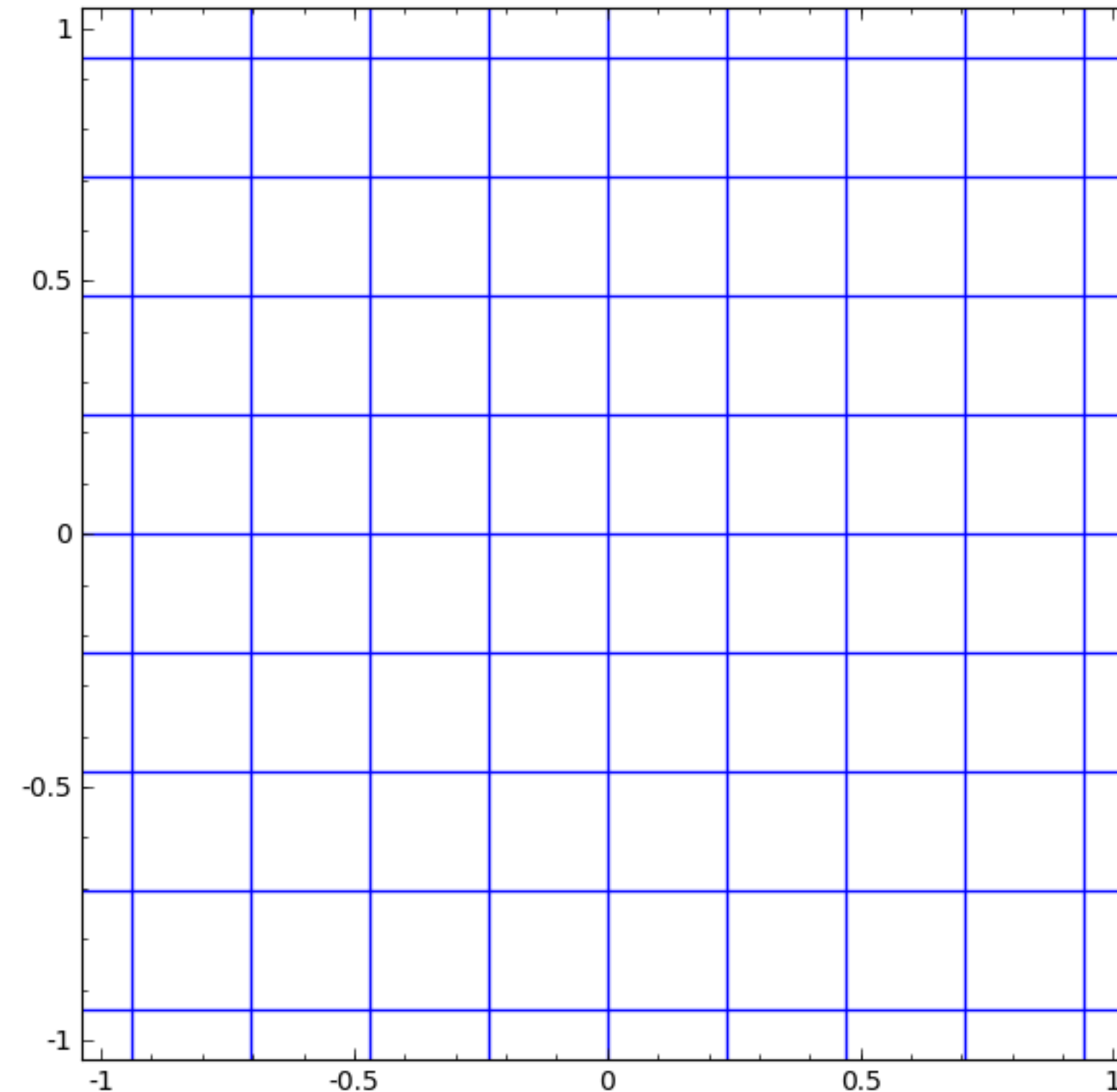
# Neural Networks

$$\mathbf{z} = g(V f(\mathbf{x}) + \mathbf{b})$$

Nonlinear transformation      Warp space      Shift

$$y_{\text{pred}} = \operatorname{argmax}_y \mathbf{w}_y^\top \mathbf{z}$$

- ▶ Ignore shift /  $+\mathbf{b}$  term for the rest of the course





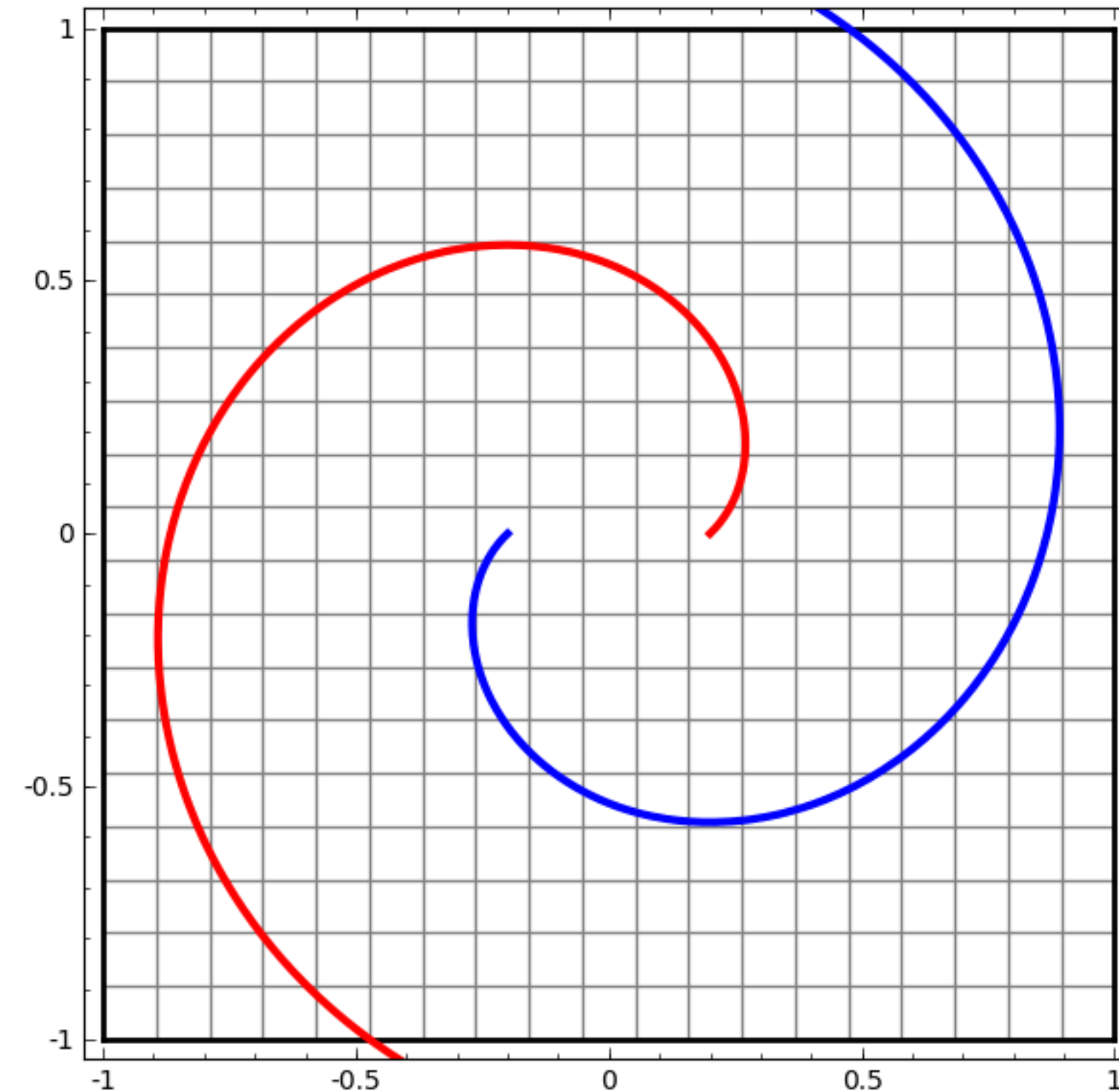
# Deep Neural Networks

$$\mathbf{z}_1 = g(V_1 f(\mathbf{x}))$$

$$\mathbf{z}_2 = g(V_2 \mathbf{z}_1)$$

...

$$y_{\text{pred}} = \operatorname{argmax}_y \mathbf{w}_y^\top \mathbf{z}_n$$





# Classification Review

---

- ▶ See Instapoll

# Feedforward Networks



# Vectorization and Softmax

$$P(y|\mathbf{x}) = \frac{\exp(\mathbf{w}_y^\top f(\mathbf{x}))}{\sum_{y' \in \mathcal{Y}} \exp(\mathbf{w}_{y'}^\top f(\mathbf{x}))}$$

- ▶ Single scalar probability

- ▶ Three classes, “different weights”

$\mathbf{w}_1^\top f(\mathbf{x})$	-1.1	softmax →	0.036	class probs
$\mathbf{w}_2^\top f(\mathbf{x}) =$	2.1		0.89	
$\mathbf{w}_3^\top f(\mathbf{x})$	-0.4		0.07	

- ▶ Softmax operation = “exponentiate and normalize”
- ▶ We write this as:  $\text{softmax}(W f(\mathbf{x}))$





# Logistic Regression as a Neural Net

---

$$P(y|\mathbf{x}) = \frac{\exp(\mathbf{w}_y^\top f(\mathbf{x}))}{\sum_{y' \in \mathcal{Y}} \exp(\mathbf{w}_{y'}^\top f(\mathbf{x}))}$$

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W f(\mathbf{x}))$$

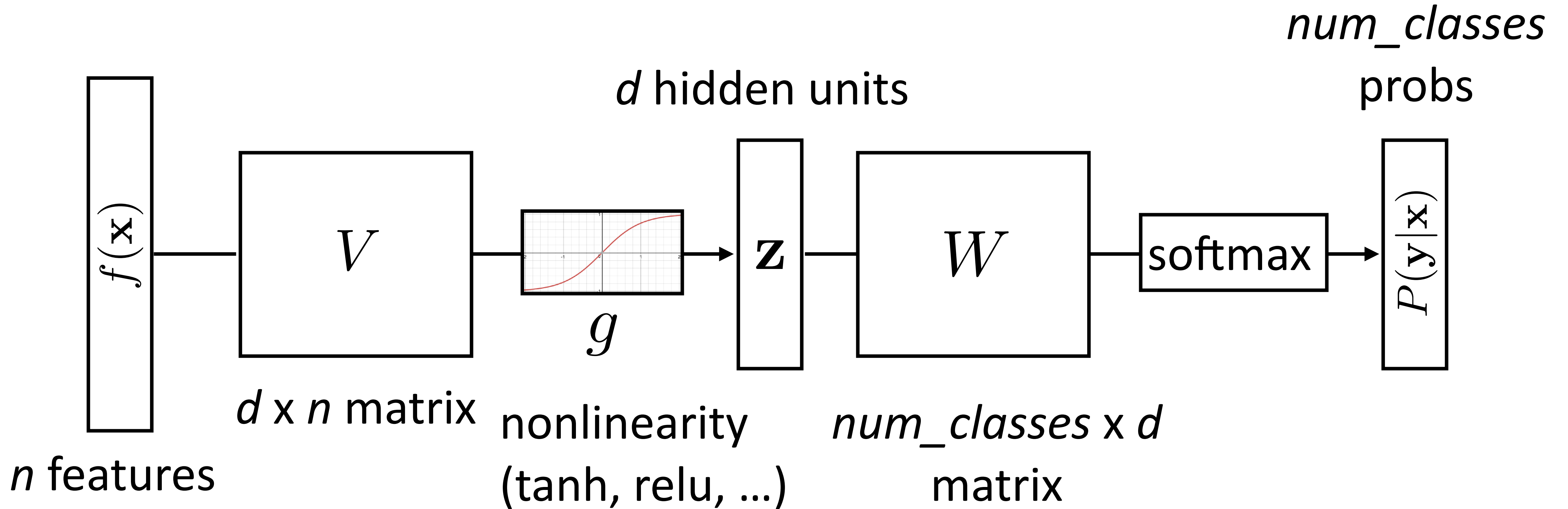
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W g(V f(\mathbf{x})))$$

- ▶ Single scalar probability
- ▶ Weight vector per class;  
 $W$  is [num classes x num feats]
- ▶ Now one hidden layer



# Neural Networks for Classification

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



# Backpropagation (with pictures)



# Training Objective

---

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

- ▶ Consider the log likelihood of a single training example:

$$\mathcal{L}(\mathbf{x}, i^*) = \log P(y = i^* | \mathbf{x})$$

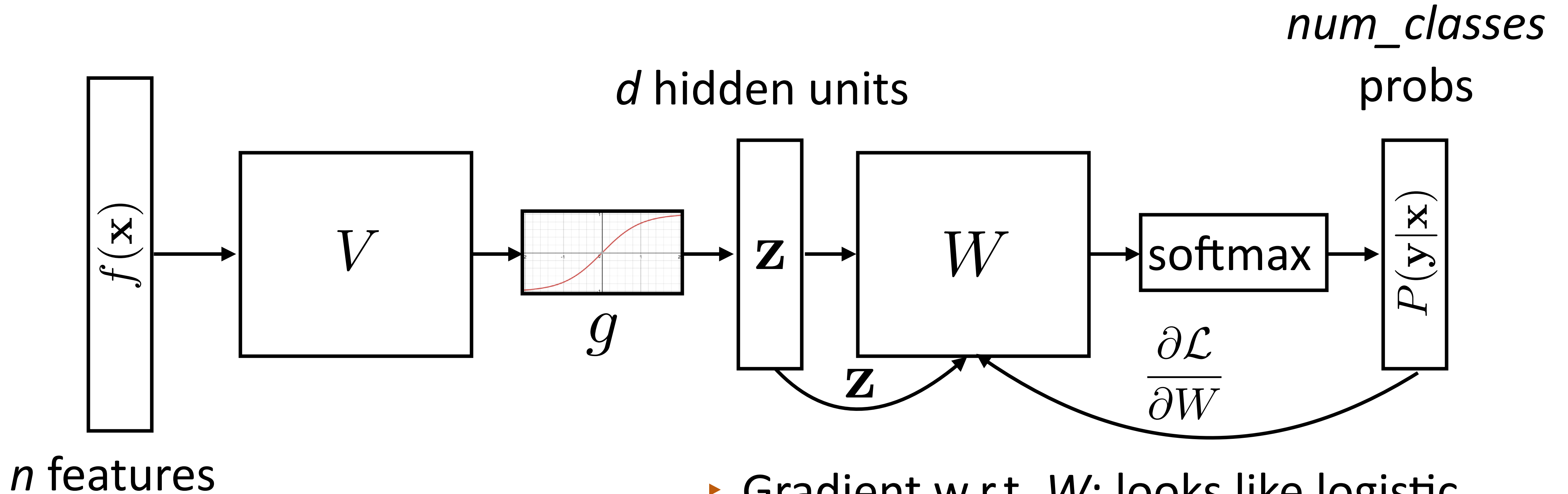
where  $i^*$  is the index of the gold label for an example

- ▶ Backpropagation is an algorithm for computing gradients of  $W$  and  $V$  (and in general any network parameters)



# Backpropagation: Picture

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

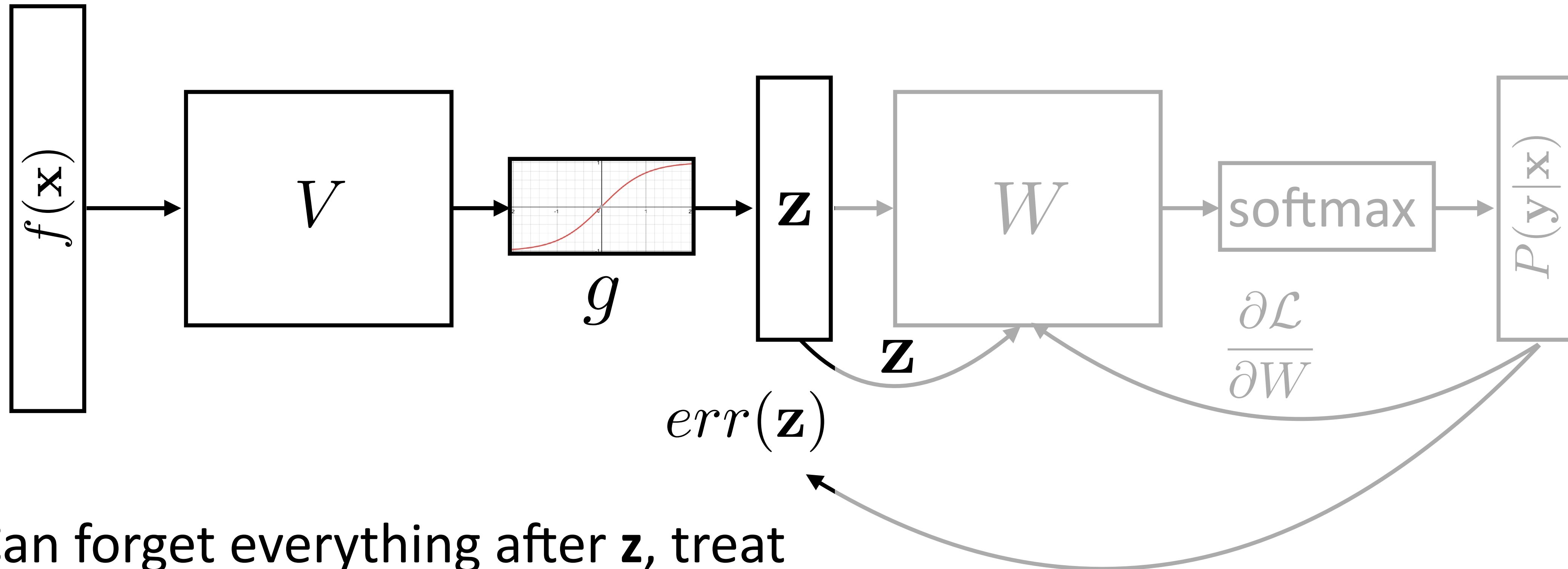


- ▶ Gradient w.r.t.  $W$ : looks like logistic regression, can be computed treating  $\mathbf{z}$  as the features



# Backpropagation: Picture

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

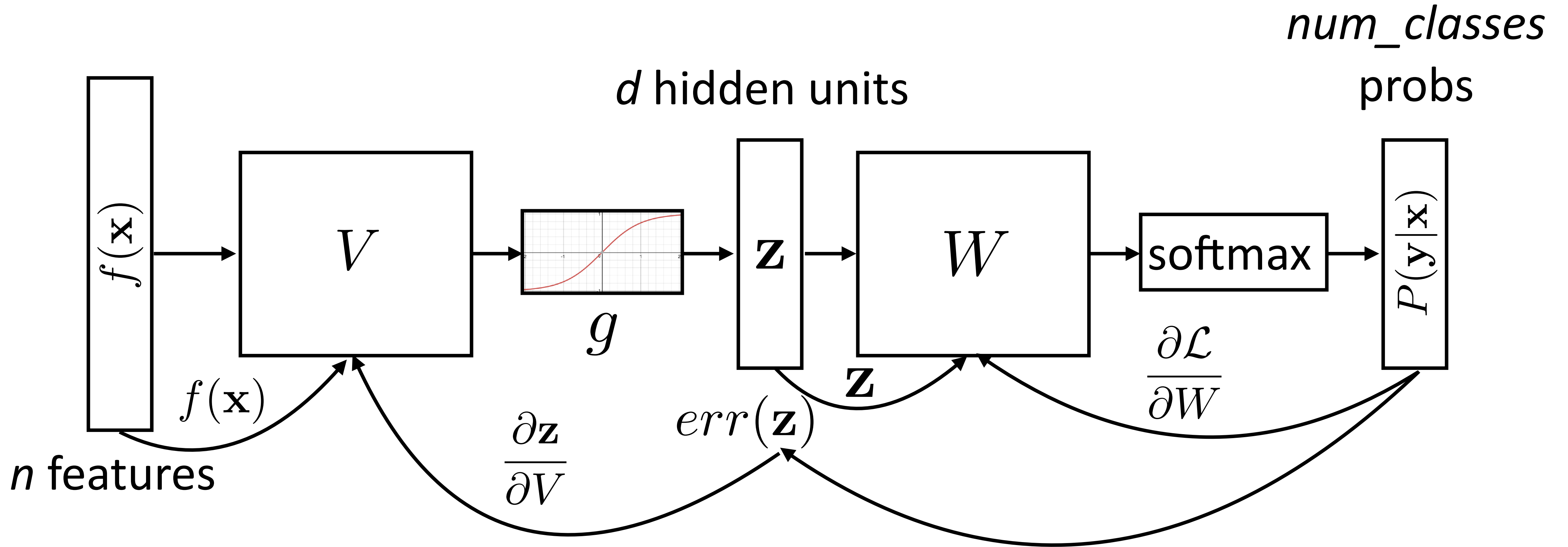


- ▶ Can forget everything after  $\mathbf{z}$ , treat it as the output and keep backpropping



# Backpropagation: Picture

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W g(V f(\mathbf{x})))$$



- ▶ Combine backward gradients with forward-pass products

# Pytorch Basics

**(code examples are on the course website: [ffnn\\_example.py](#) )**





# 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

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

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  
(can also use NLLLoss)



# 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. See optimization video on course website. **(Pytorch does this by default so you don't actually have to include it.)**



# 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

Pytorch example

# Batching





# Batching

---

- ▶ Modify the training loop to run over multiple examples at once

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

- ▶ Batch sizes from 1-100 often work well
- ▶ Can use the same network as before **without modification**

# DANs



Credit: Stephen Roller



# 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

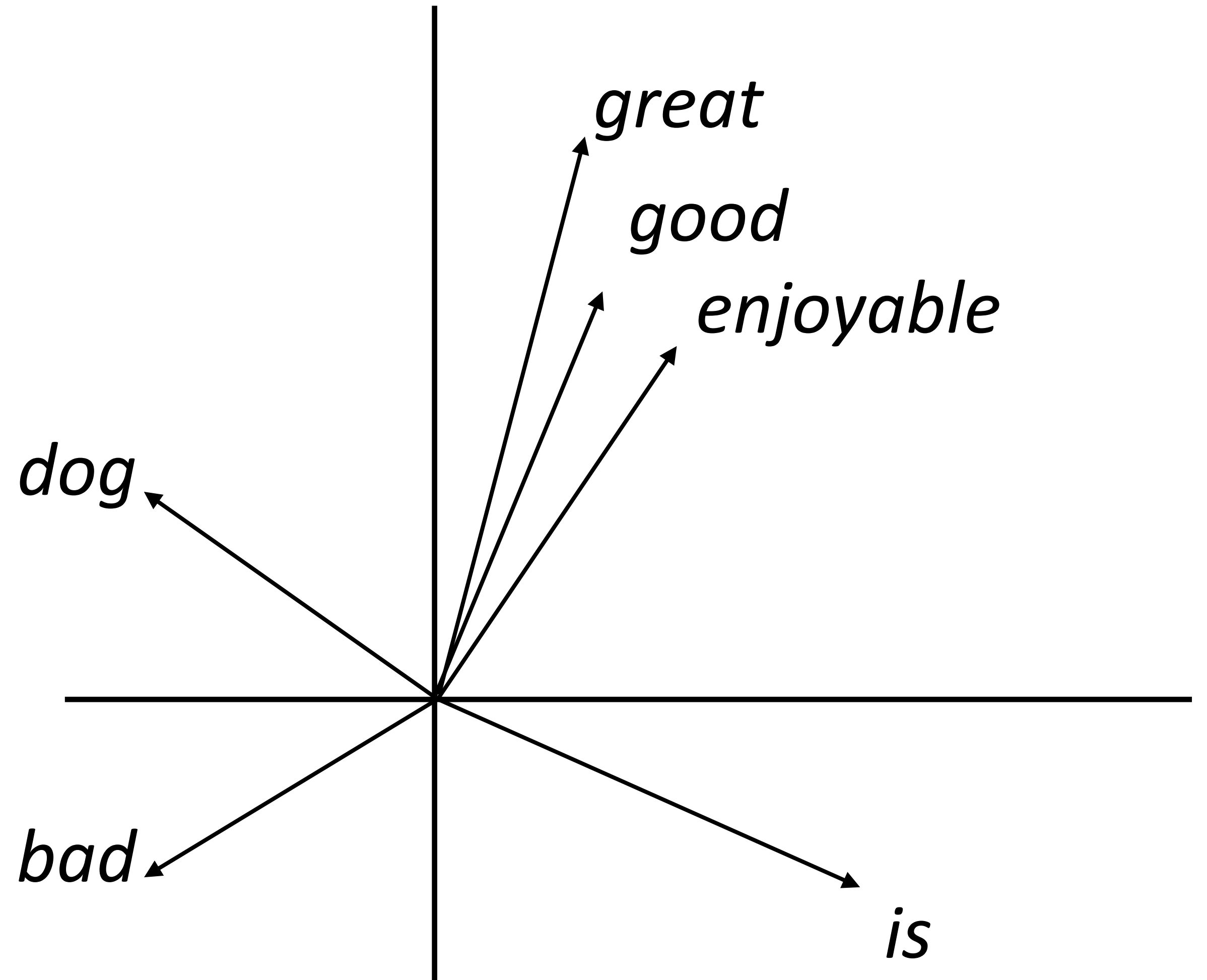


# Word Embeddings

- ▶ Want a vector space where similar words have similar embeddings

*great*  $\approx$  *good*

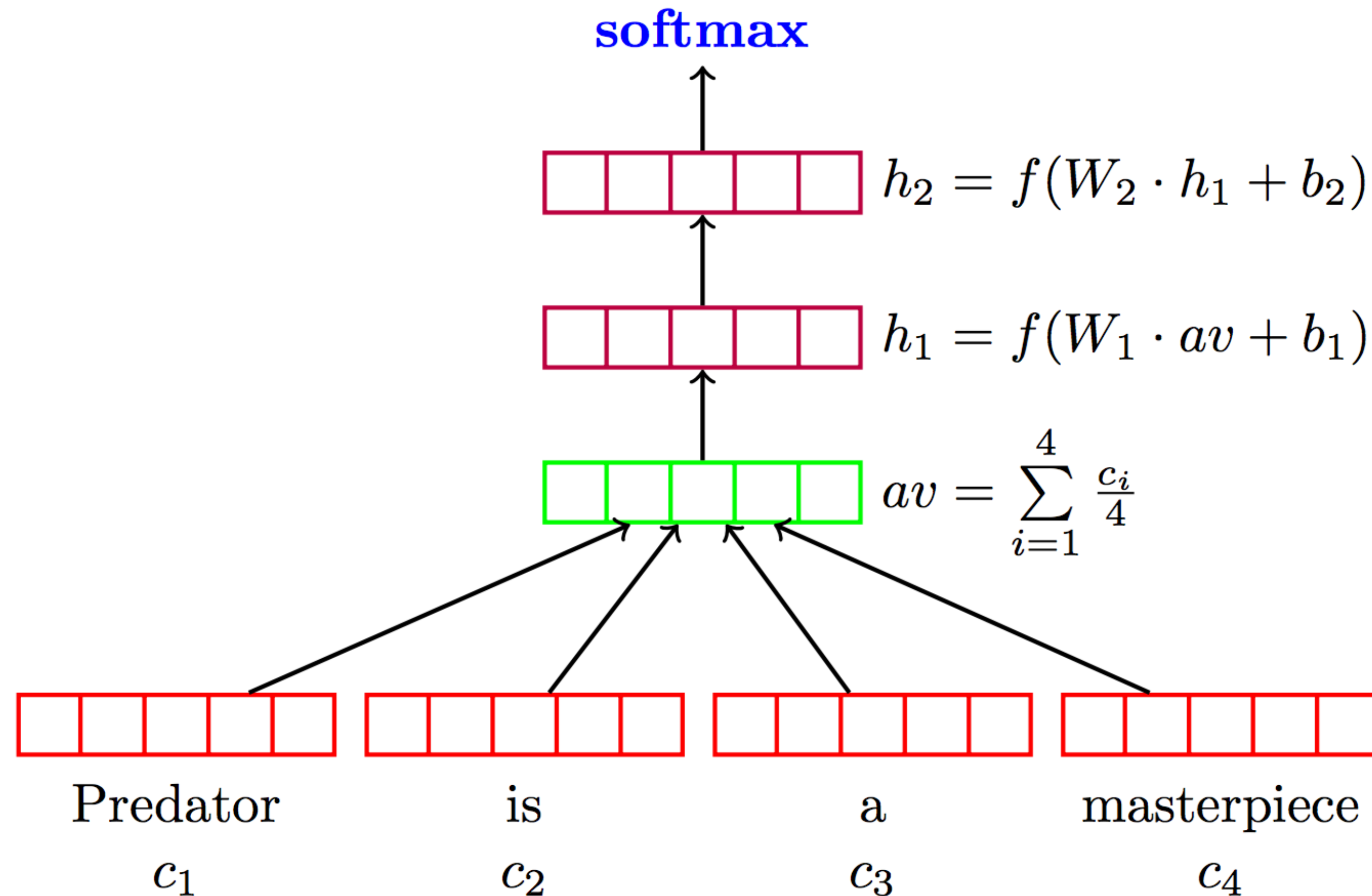
- ▶ Next lecture: come up with a way to produce these embeddings
- ▶ For each word, want “medium” dimensional vector (50-300 dims) representing it





# Deep Averaging Networks

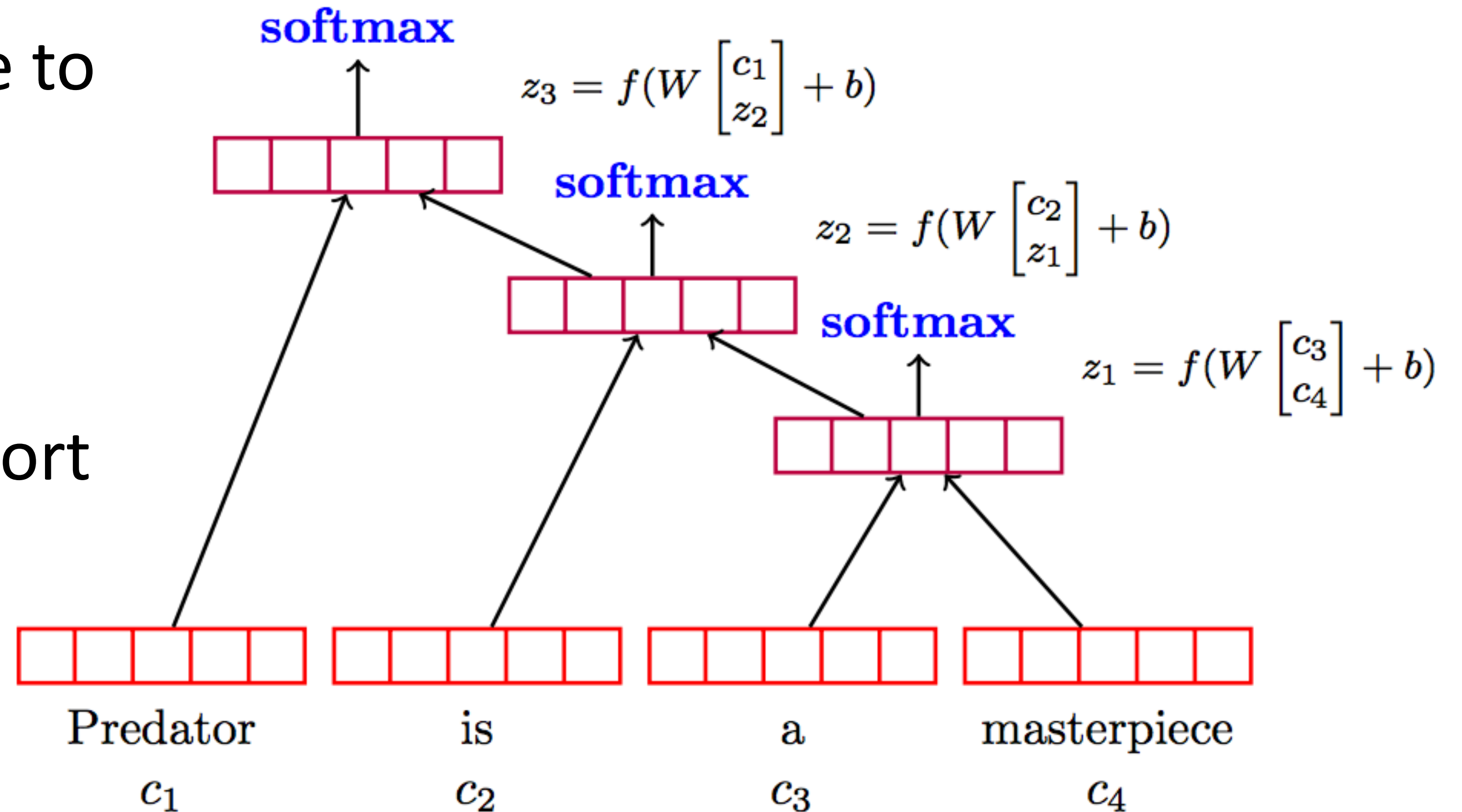
- ▶ Deep Averaging Networks: feedforward neural network on average of word embeddings from input





# 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





# Sentiment Analysis

No pretrained embeddings

Model	RT	SST fine	SST bin	IMDB	Time (s)
DAN-ROOT	—	46.9	85.7	—	<b>31</b>
DAN-RAND	77.3	45.4	83.2	88.8	136
DAN	80.3	47.7	86.3	89.4	136

Iyyer et al. (2015)

Bag-of-words

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	—

Wang and Manning (2012)

Tree-structured neural networks

RecNN*	77.7	43.2	82.4	—	—
RecNTN*	—	45.7	85.4	—	—
DRecNN	—	49.8	86.6	—	431
TreeLSTM	—	<b>50.6</b>	86.9	—	—
DCNN*	—	48.5	86.9	89.4	—
PVEC*	—	48.7	87.8	<b>92.6</b>	—
CNN-MC	<b>81.1</b>	47.4	<b>88.1</b>	—	2,452
WRRBM*	—	—	—	89.2	—

Kim (2014)



# Deep Averaging Networks

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

- ▶ Will return to compositionality with syntax and LSTMs

Iyyer et al. (2015)

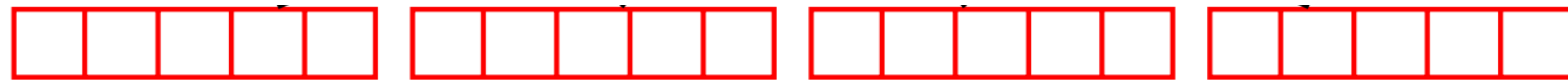




# Word Embeddings in PyTorch

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

Predator is a masterpiece  
1820 24 1 2047



- ▶  $n$  indices  $\Rightarrow n \times d$  matrix of  $d$ -dimensional word embeddings
- ▶  $b \times n$  indices  $\Rightarrow b \times n \times d$  tensor of  $d$ -dimensional word embeddings



# Word Embeddings

---



# Word Embeddings

---