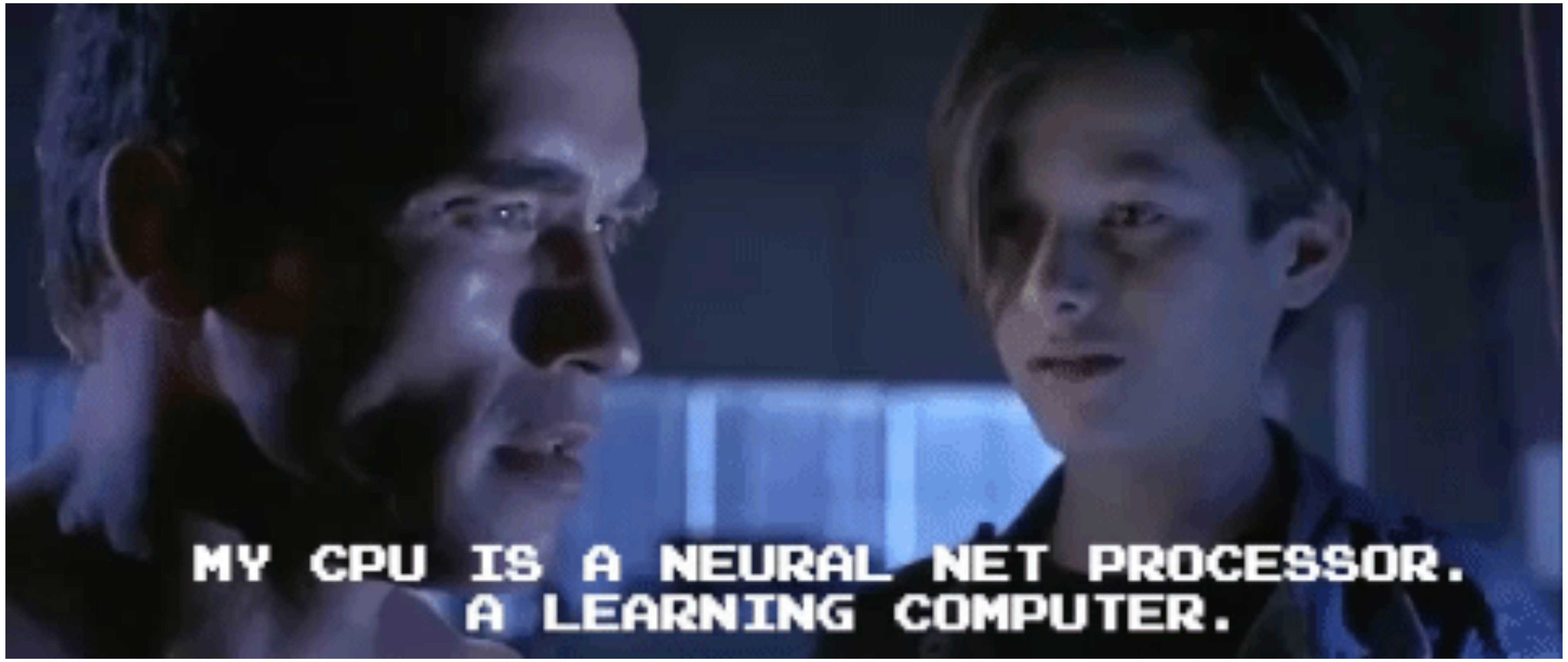
# CS378: Natural Language Processing

# Lecture 6: NN Implementation




Greg Durrett

TEXAS
The University of Texas at Austin

# Announcements

- Assignment 1 due today
- Assignment 2 out today, due in two weeks
- Fairness response due Thursday (submit on Canvas)
- Seating chart

# Recap

# 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"

$$\begin{matrix} \mathbf{w}_1^\top f(\mathbf{x}) \\ \mathbf{w}_2^\top f(\mathbf{x}) \\ \mathbf{w}_3^\top f(\mathbf{x}) \end{matrix} = \begin{matrix} -1.1 \\ 2.1 \\ -0.4 \end{matrix} \xrightarrow{\text{softmax}} \begin{matrix} 0.036 \\ 0.89 \\ 0.07 \end{matrix} \quad \text{class probs}$$

- Softmax operation = "exponentiate and normalize"
- We write this as: $\mathrm{softmax}(W f(\mathbf{x}))$

# Logistic Regression as a Neural Net

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

- Single scalar probability

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

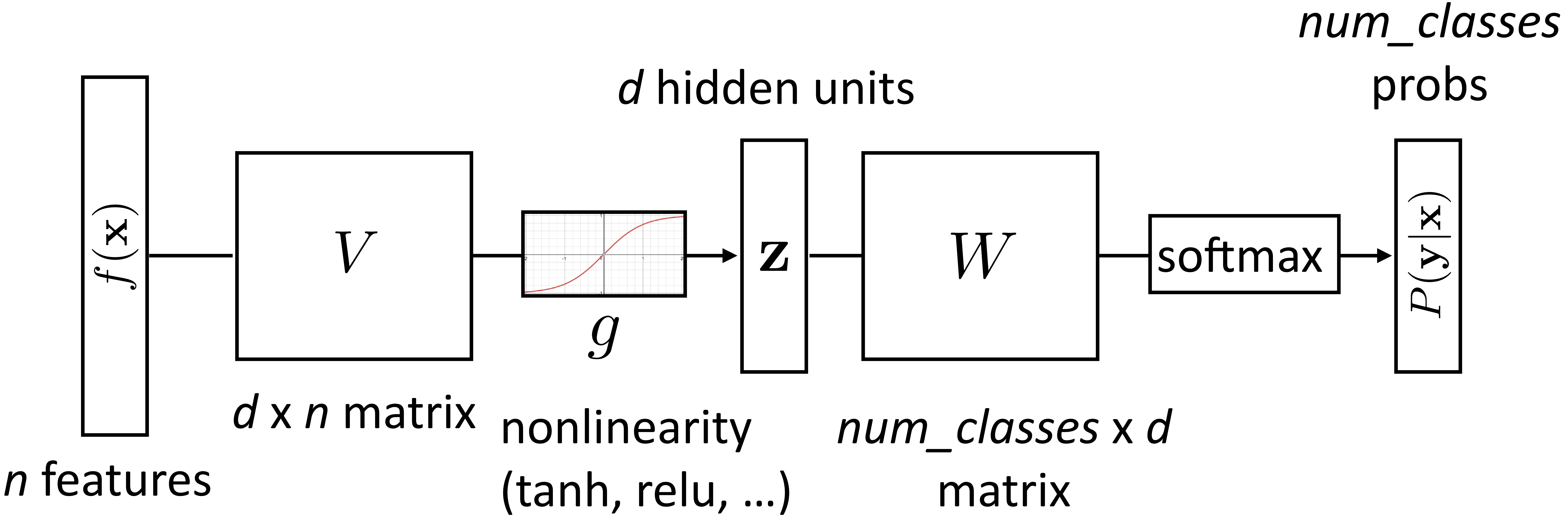
- Weight vector per class;
  *W* is [num classes x num feats]

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

- Now one hidden layer

# Neural Networks for Classification

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



# Backpropagation
# (with pictures! Full derivations at the end of the slides)

# Training Objective

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

- Consider the log likelihood of a single training example:

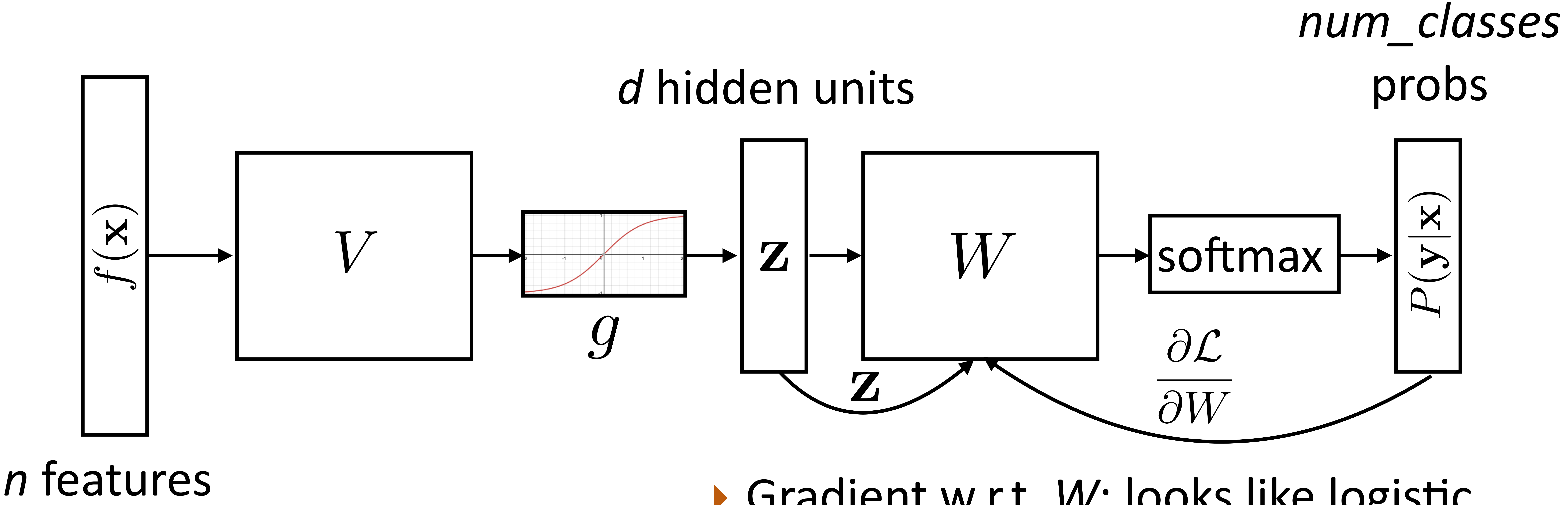
$$\mathcal{L}(\mathbf{x}, i^*) = \log P(y = i^*|\mathbf{x})$$

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

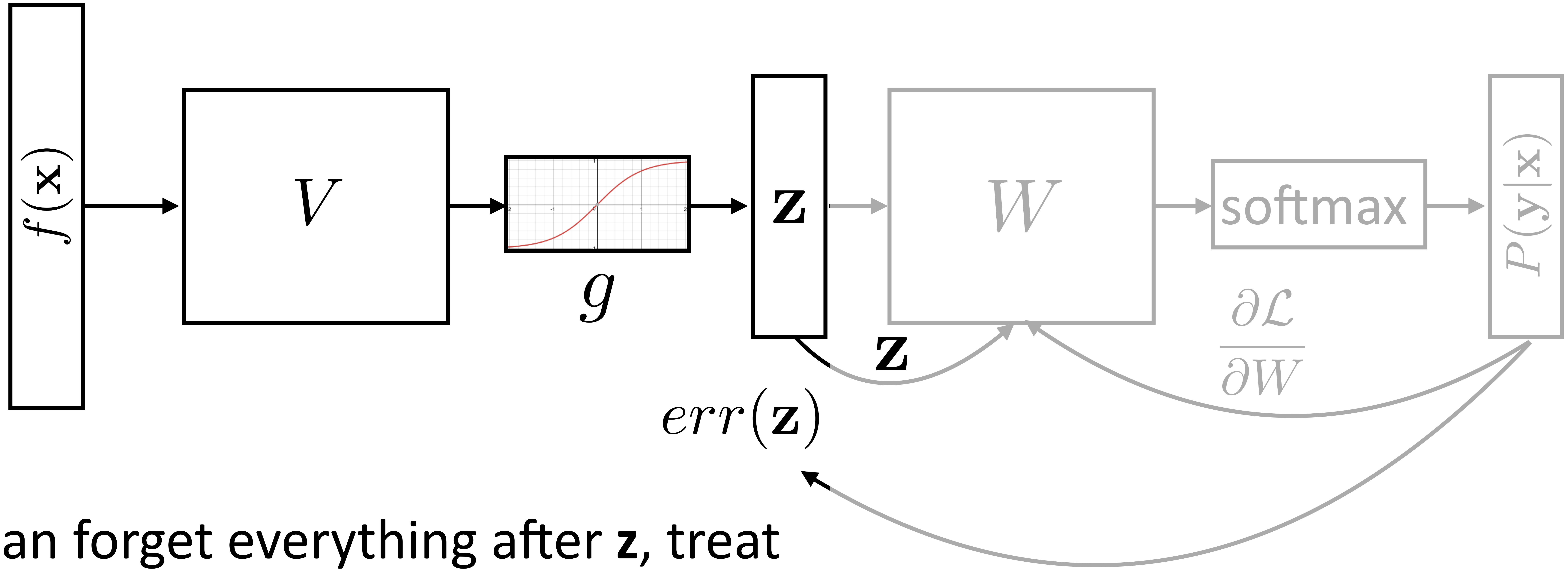
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$




- Gradient w.r.t. *W*: looks like logistic regression, can be computed treating **z** as the features

# Backpropagation: Picture

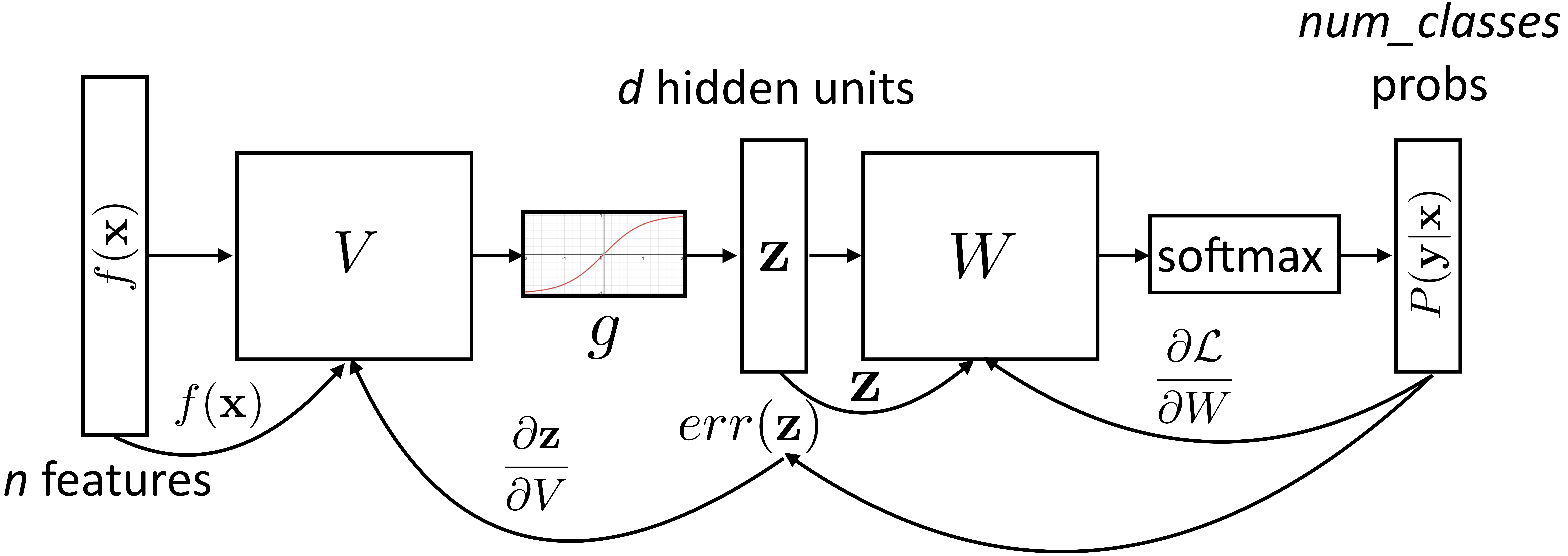
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$




- Can forget everything after **z**, treat it as the output and keep backpropping

# Backpropagation: Picture

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




- 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

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

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. See optimization video on course website

# 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

```
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)
```

- Can run this in a batched fashion without modification!

# 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 = ffnn.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

# 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

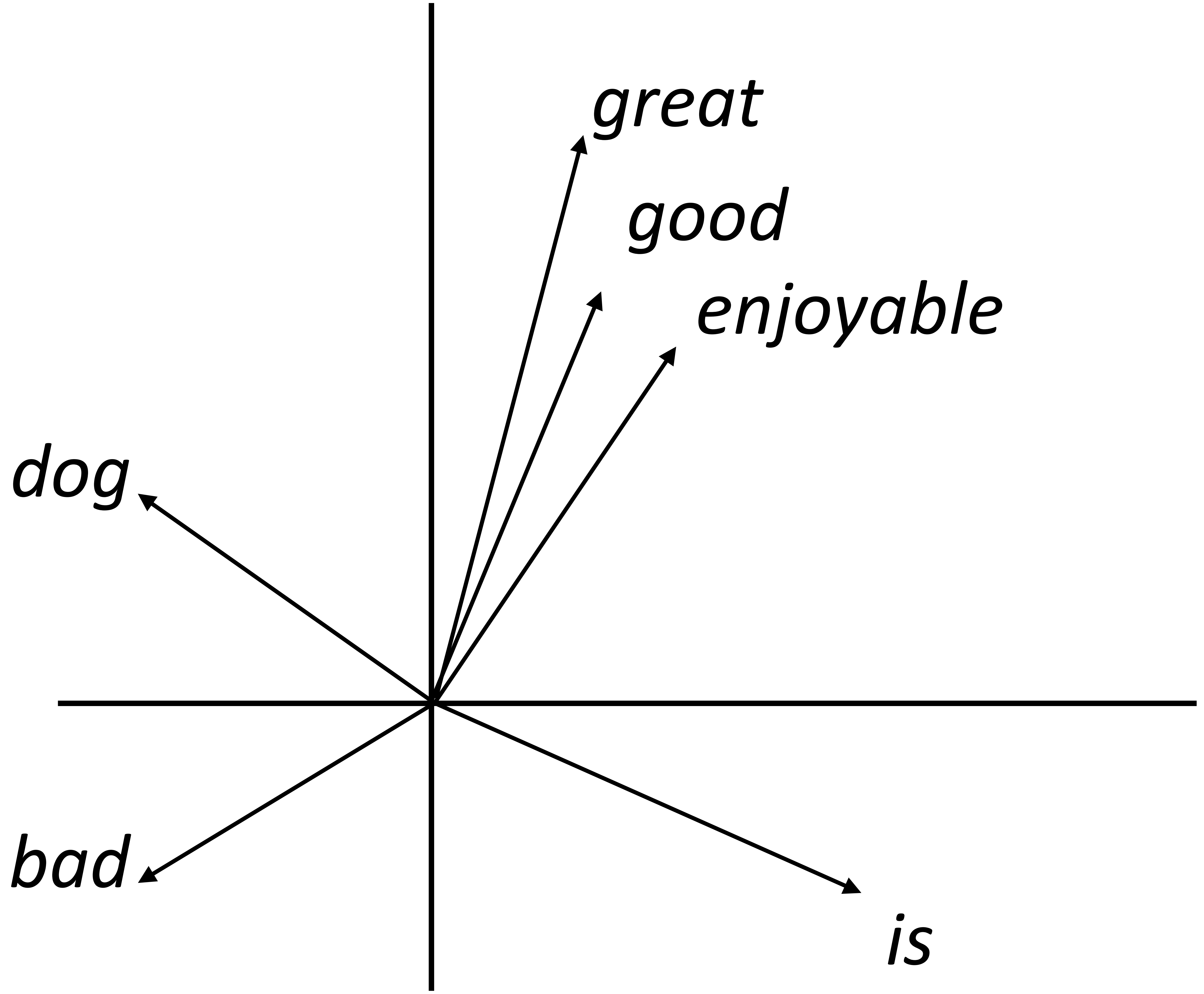
# Word Embeddings

- Want a vector space where similar words have similar embeddings

*great* ≈ *good*

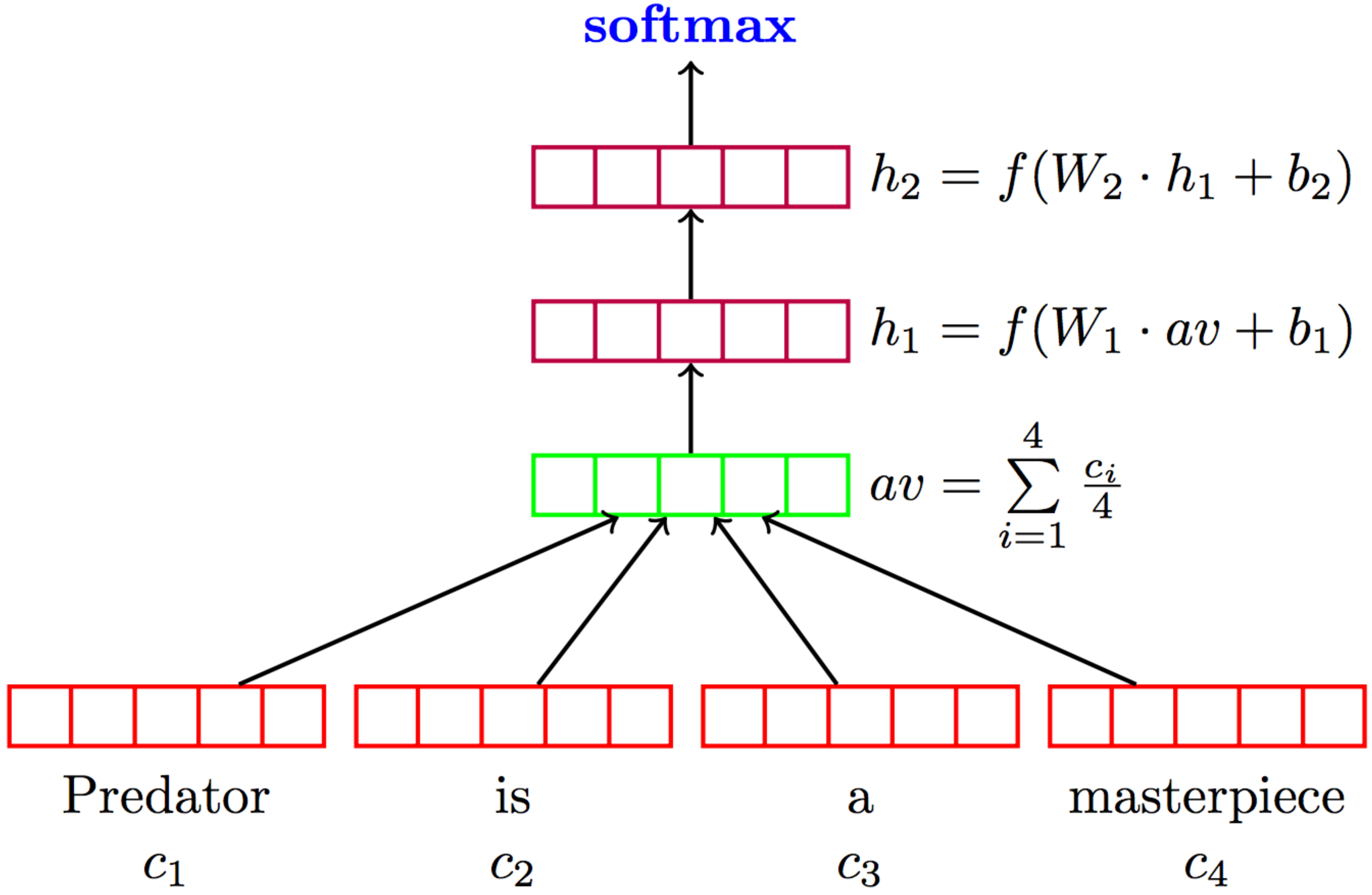
- Next lecture: come up with a way to produce these embeddings

- 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

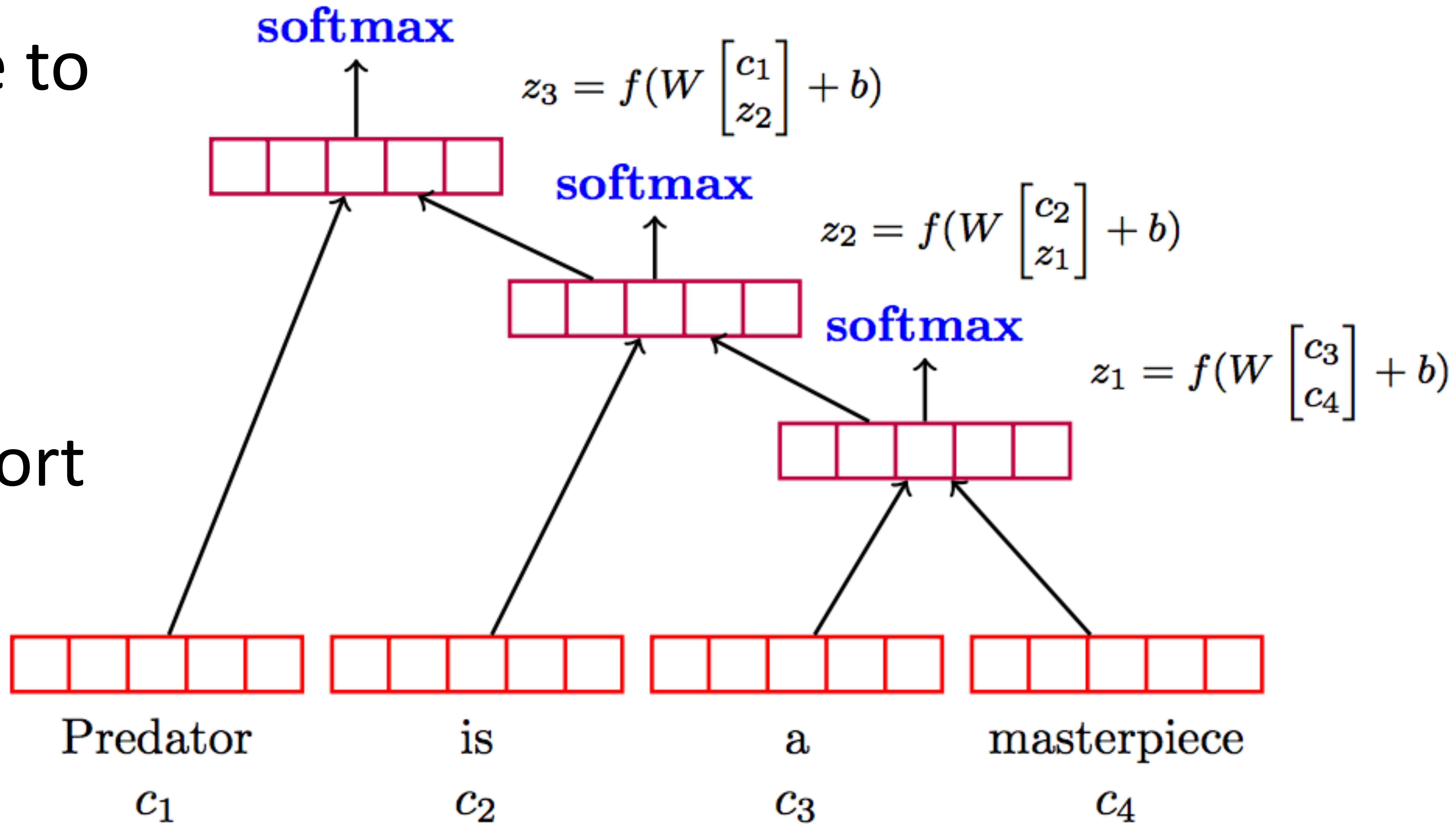



Iyyer et al. (2015)

# 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 (→ DAN-RAND)

Iyyer et al. (2015) (DAN)

Bag-of-words (NBOW-RAND, NBOW, BiNB, NBSVM-bi)

Wang and Manning (2012) (NBSVM-bi)

Tree-structured neural networks (RecNN* through WRRBM*)

Kim (2014) (CNN-MC)

# 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

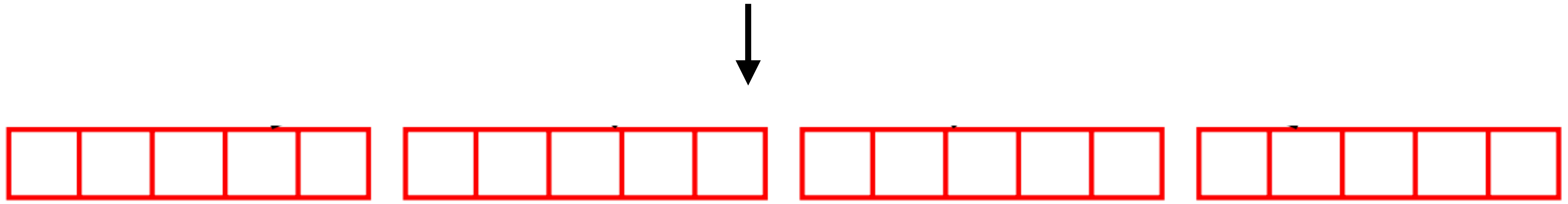
Iyyer et al. (2015)

# 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

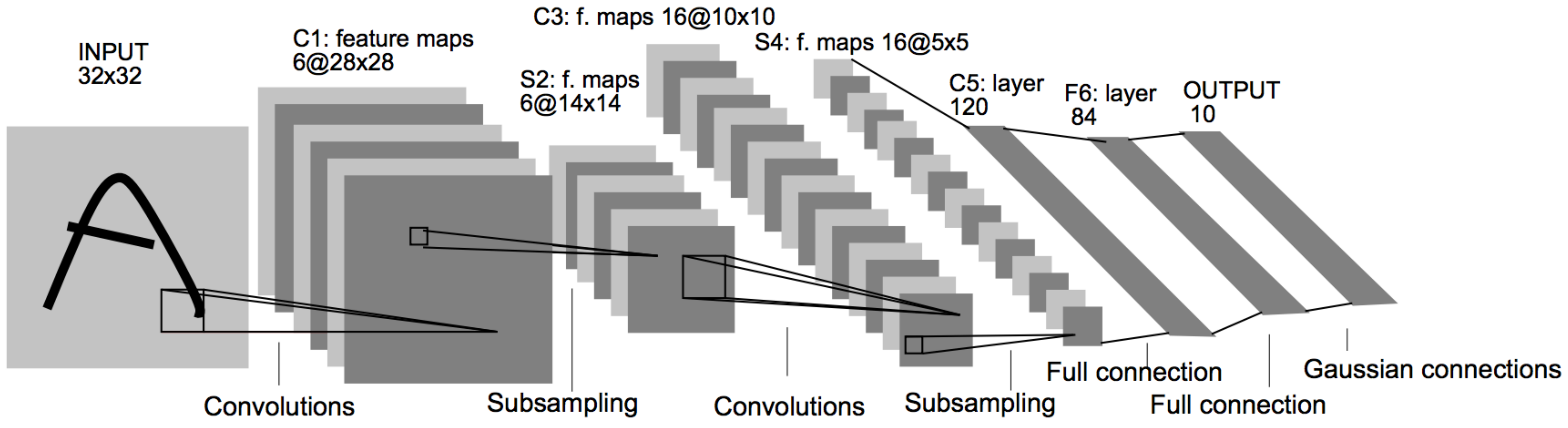
# Word Embeddings

# Word Embeddings

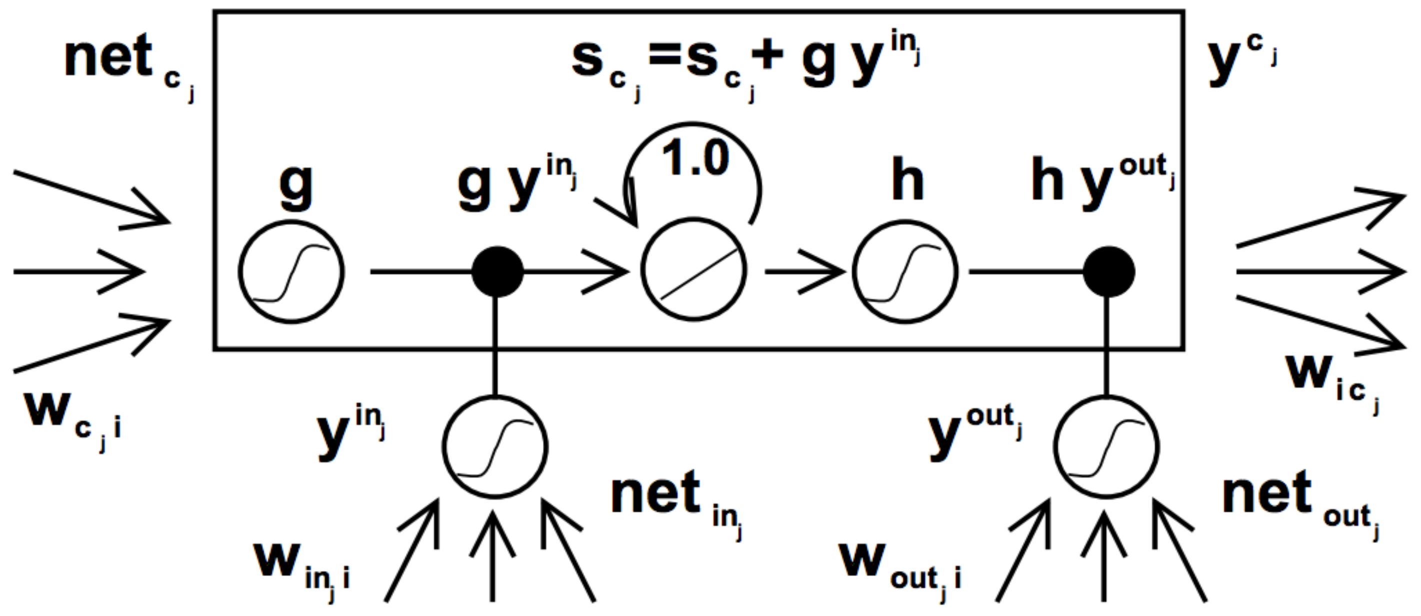
# Neural Nets History

# History: NN “dark ages”

- Convnets: applied to MNIST by LeCun in 1998



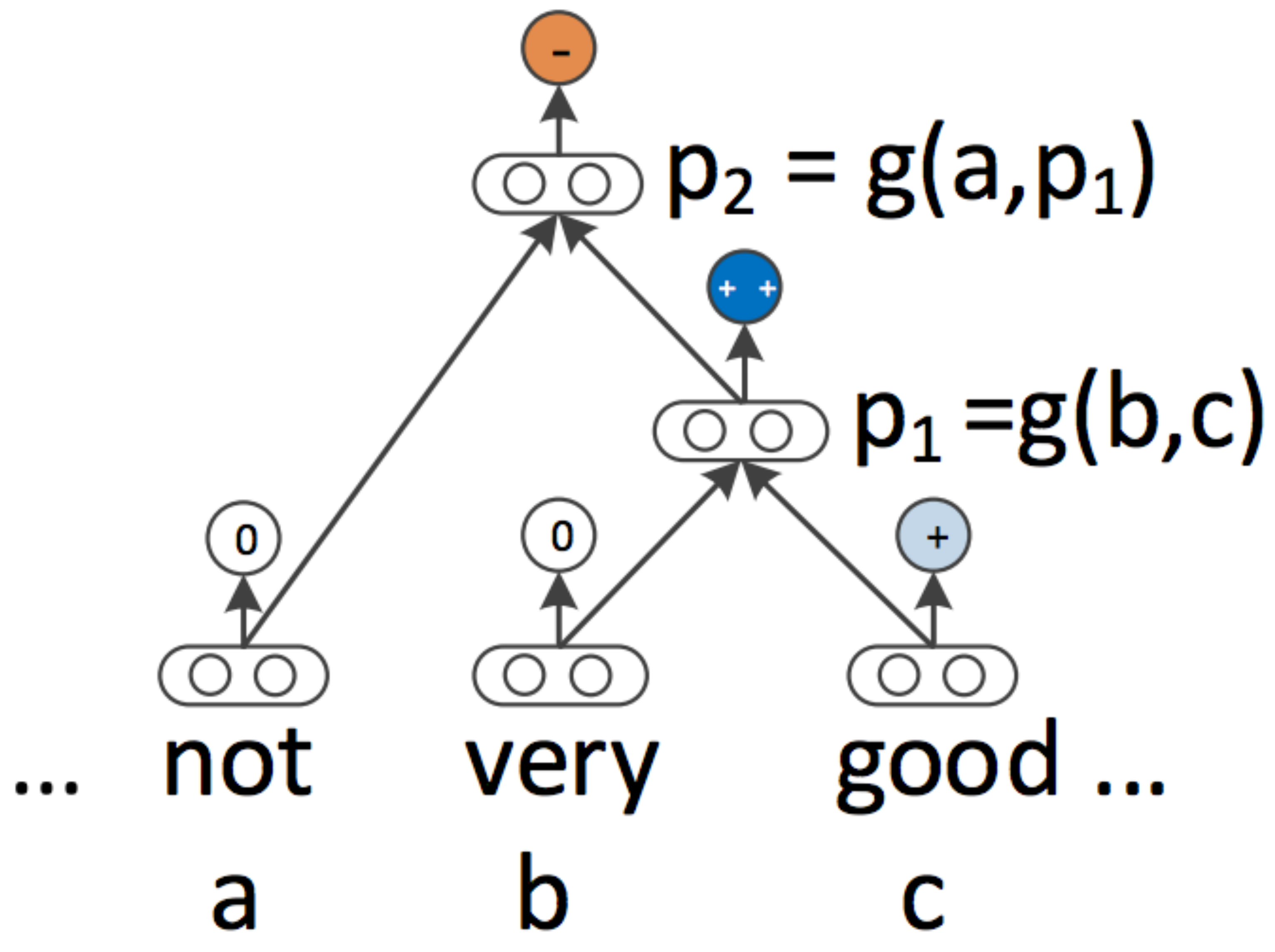

- LSTMs: Hochreiter and Schmidhuber (1997)




- Henderson (2003): neural shift-reduce parser, not SOTA

# 2008-2013: A glimmer of light…

- Collobert and Weston 2011: "NLP (almost) from scratch"
  - Feedforward neural nets induce features for sequential CRFs ("neural CRF")
- Krizhevskey et al. (2012): AlexNet for vision
- Socher 2011-2014: tree-structured RNNs working okay



# 2014: Stuff starts working

- Kim (2014) + Kalchbrenner et al. (2014): sentence classification / sentiment (convnets work for NLP?)
- Sutskever et al. (2014) + Bahdanau et al. (2014): seq2seq for neural MT (LSTMs work for NLP?)
- Chen and Manning transition-based dependency parser (feedforward)
- 2015: explosion of neural nets for everything under the sun

# Why didn't they work before?

- **Datasets too small**: for MT, not really better until you have 1M+ parallel sentences (and really need a lot more)
- **Optimization not well understood**: good initialization, per-feature scaling + momentum (Adagrad / Adadelta / Adam) work best out-of-the-box
  - **Regularization**: dropout is pretty helpful
  - **Computers not big enough**: can't run for enough iterations
- **Inputs**: need word representations to have the right continuous semantics

# Backpropagation — Derivations (not covered in lecture, optional)

# Training Neural Networks

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W\mathbf{z}) \qquad \mathbf{z} = g(Vf(\mathbf{x}))$$

- Maximize log likelihood of training data

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

- *i**: index of the gold label

- $e_i$: 1 in the *i*th row, zero elsewhere. Dot by this = select *i*th index

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

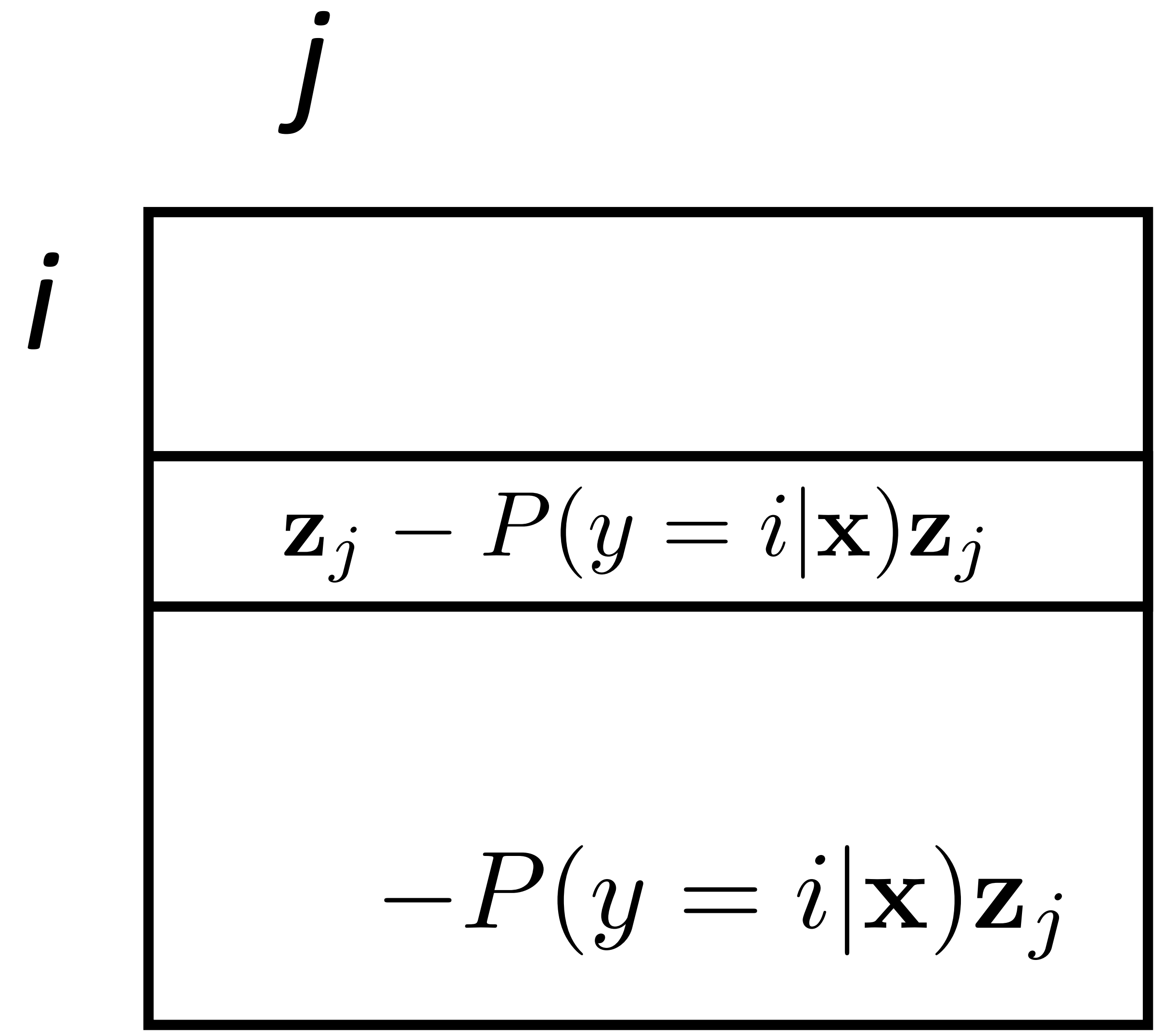
# Computing Gradients

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

- Gradient with respect to *W:*

$$\frac{\partial}{\partial W_{ij}} \mathcal{L}(\mathbf{x}, i^*) = \begin{cases} \mathbf{z}_j - P(y = i|\mathbf{x})\mathbf{z}_j & \text{if } i = i^* \\ -P(y = i|\mathbf{x})\mathbf{z}_j & \text{otherwise} \end{cases}$$

gradient w.r.t. *W*




- Looks like logistic regression with ***z*** as the features!

# Computing Gradients: Backpropagation

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

$$\mathbf{z} = g(Vf(\mathbf{x}))$$

Activations at hidden layer

- Gradient with respect to *V*: apply the chain rule

$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V_{ij}}$$

[some math…]

$$err(\text{root}) = e_{i^*} - P(\mathbf{y}|\mathbf{x})$$

dim = *num_classes*

$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = err(\mathbf{z}) = W^\top err(\text{root})$$

dim = *d*

# Computing Gradients: Backpropagation

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

$\mathbf{z} = g(Vf(\mathbf{x}))$

Activations at hidden layer

- Gradient with respect to *V*: apply the chain rule

$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V_{ij}} \qquad \frac{\partial \mathbf{z}}{\partial V_{ij}} = \frac{\partial g(\mathbf{a})}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial V_{ij}}$$

$\mathbf{a} = Vf(\mathbf{x})$

- First term: gradient of nonlinear activation function at ***a*** (depends on current value)
- Second term: gradient of linear function
- First term: *err*(**z**); represents gradient w.r.t. **z**

# Backpropagation

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

- Step 1: compute $err(\text{root}) = e_{i^*} - P(\mathbf{y}|\mathbf{x})$ (vector)
- Step 2: compute derivatives of *W* using *err*(root) (matrix)
- Step 3: compute $\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = err(\mathbf{z}) = W^\top err(\text{root})$ (vector)
- Step 4: compute derivatives of *V* using *err*(**z**) (matrix)
- Step 5+: continue backpropagation if necessary