CS388: Natural Language Processing

Lecture 4: Neural Networks

TEXAS
The University of Texas at Austin

Administration

- Auditing
  - I will add you to the canvas, send me your EID
  - Some restrictions
    - No assignments/quizzes/midterms
    - Needs to be enough room in the class
    - On quiz days, please wait until after the quiz is over to enter
  - HW1: Due on Monday.

Recall: Multiclass Classification

- Logistic regression: $$P_{\mathbf{w}}(y = \hat{y} \mid \mathbf{x}) = \frac{\exp\left(\mathbf{w}_{\hat{y}}^{\top}\mathbf{f}(\mathbf{x})\right)}{\sum_{y'} \exp\left(\mathbf{w}_{y'}^{\top}\mathbf{f}(\mathbf{x})\right)}$$

Gradient of log likelihood:

$$\frac{\partial}{\partial \mathbf{w}_{y^{(i)}}} \mathcal{L}(\mathbf{x}^{(i)}, y^{(i)}) = \mathbf{f}(\mathbf{x}^{(i)})(P_{\mathbf{w}}(y^{(i)} \mid \mathbf{x}^{(i)}) - 1)$$

"increase value for gold weight vector, decrease for other weight vectors"

$$\frac{\partial}{\partial \mathbf{w}_{\tilde{y}}} \mathcal{L}(\mathbf{x}^{(i)}, y^{(i)}) = \mathbf{f}(\mathbf{x}^{(i)}) P_{\mathbf{w}}(y^{(i)} \mid \mathbf{x}^{(i)})$$

This Lecture

- Neural network basics
- Feedforward neural networks + backpropagation
- Most basic neural language model

Neural Net Basics

Neural Networks

- Linear classification: $\mathrm{argmax}_y w^\top f(x, y)$
- Want to learn intermediate conjunctive features of the input

*the movie was **not** all that **good***

I[contains *not* & contains *good*]

- How do we learn this if our feature vector is just the unigram indicators?

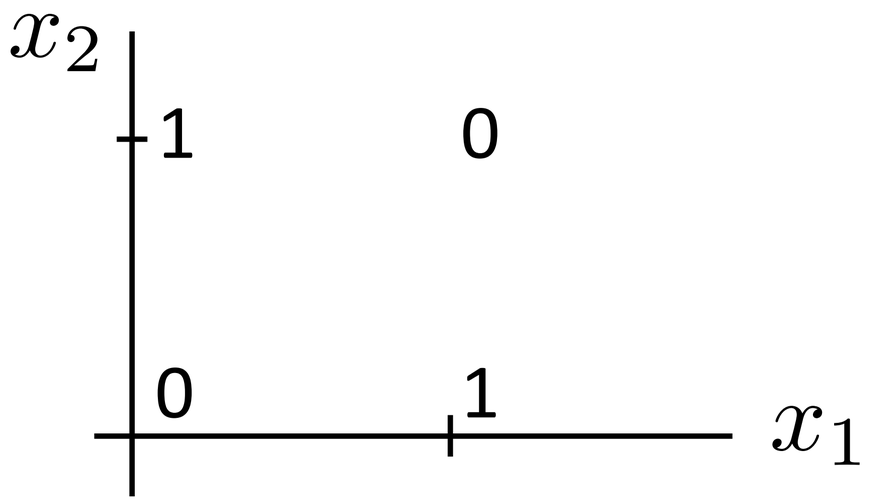
I[contains *not*], I[contains *good*]

Neural Networks: XOR

- Let's see how we can use neural nets to learn a simple nonlinear function
- Inputs $x_1, \; x_2$

  $(\text{generally } \mathbf{x} = (x_1, \ldots, x_m))$
- Output y

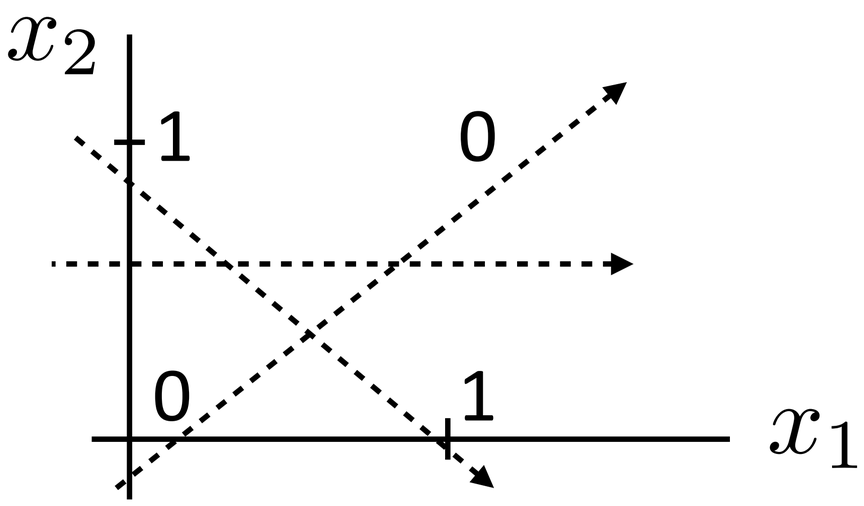
  $(\text{generally } \mathbf{y} = (y_1, \ldots, y_n))$



x_1	x_2	$y = x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Minsky and Papert (1969): a single-layer NN cannot learn XOR

Neural Networks: XOR



$$y = a_1 x_1 + a_2 x_2$$

X

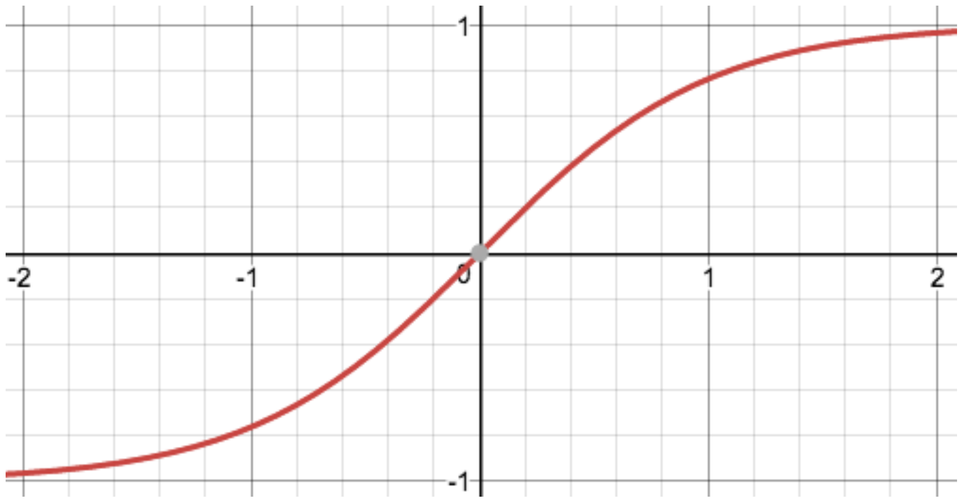
$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$

✓

"or"

(looks like action potential in neuron)

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0



Neural Networks

Linear model: $y = \mathbf{w} \cdot \mathbf{x} + b$

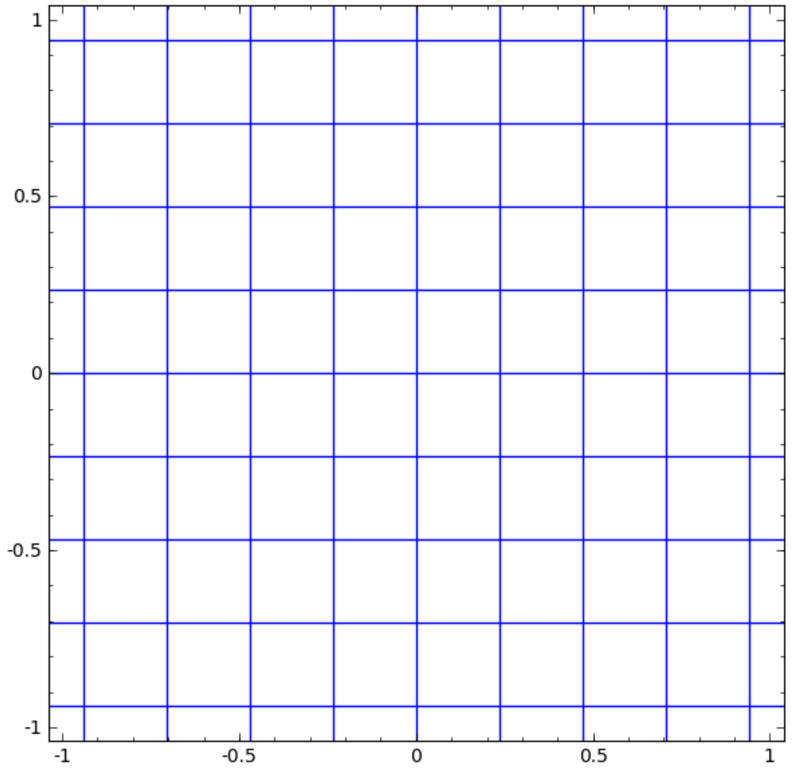
$$y = g(\mathbf{w} \cdot \mathbf{x} + b)$$

$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Nonlinear transformation

Warp space

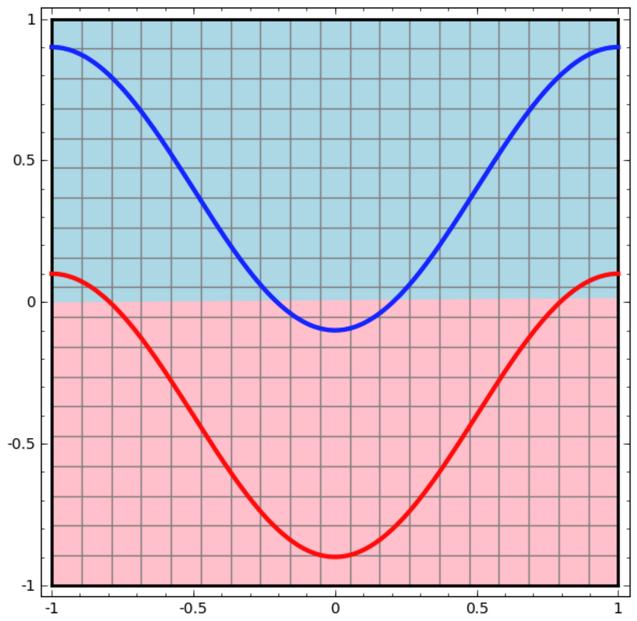
Shift



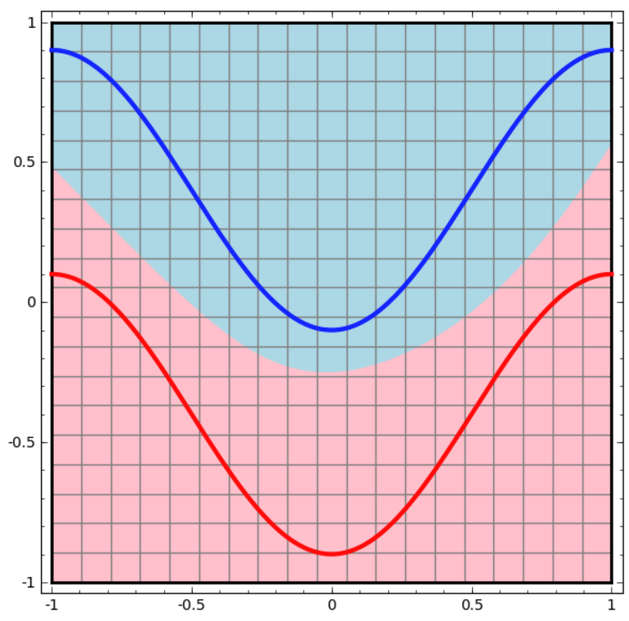
Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Neural Networks

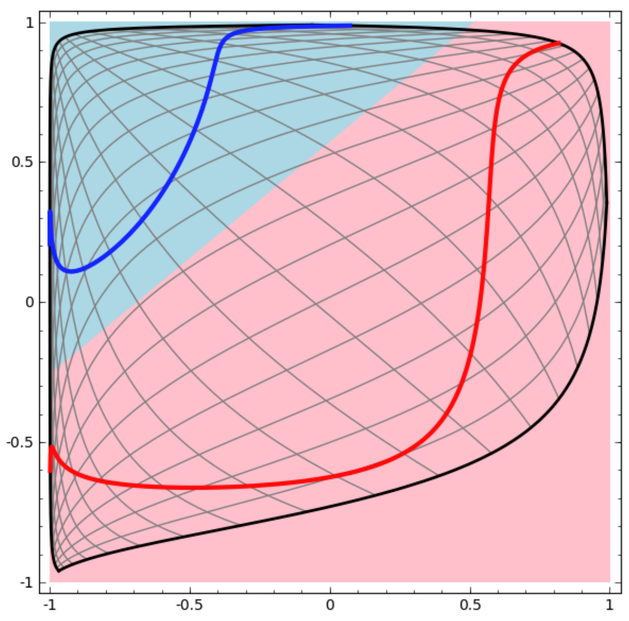
Linear classifier



Neural network



…possible because we transformed the space!



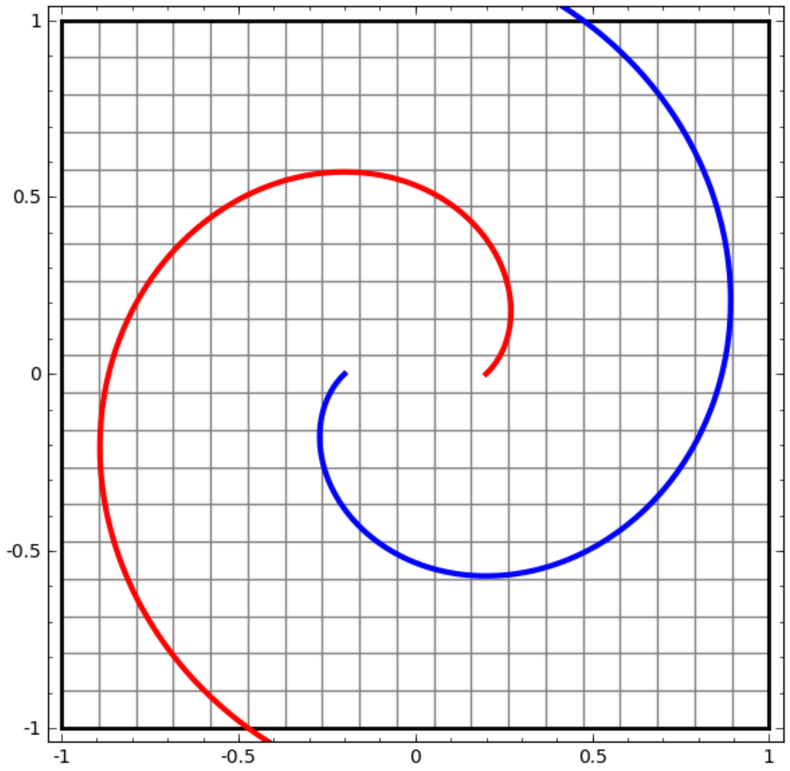
Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Deep Neural Networks

$$\boldsymbol{y} = g(\mathbf{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\mathbf{z} = g(\mathbf{V}\mathbf{y} + \mathbf{c})$$

$$\mathbf{z} = g(\mathbf{V}\underbrace{g(\mathbf{W}\mathbf{x} + \mathbf{b})}_{\text{output of first layer}} + \mathbf{c})$$



Check: what happens if no nonlinearity?

More powerful than basic linear models?

$$\mathbf{z} = \mathbf{V}(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Feedforward Networks, Backpropagation

Logistic Regression with NNs

$$P_{\mathbf{w}}(y = \hat{y} \mid \mathbf{x}) = \frac{\exp\left(\mathbf{w}_{\hat{y}}^{\top}\mathbf{f}(\mathbf{x})\right)}{\sum_{y'} \exp\left(\mathbf{w}_{y'}^{\top}\mathbf{f}(\mathbf{x})\right)}$$

- Single scalar probability

$$P(\mathbf{y} \mid \mathbf{x}) = \mathrm{softmax}([\mathbf{w}_{\hat{y}}^{\top}\mathbf{f}(\mathbf{x})]_{y \in \mathcal{Y}})$$

- Compute scores for all possible labels at once (returns vector)

$$\mathrm{softmax}(p)_i = \frac{\exp(p_i)}{\sum_{i'} \exp(p_{i'})}$$

- softmax: exps and normalizes a given vector

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

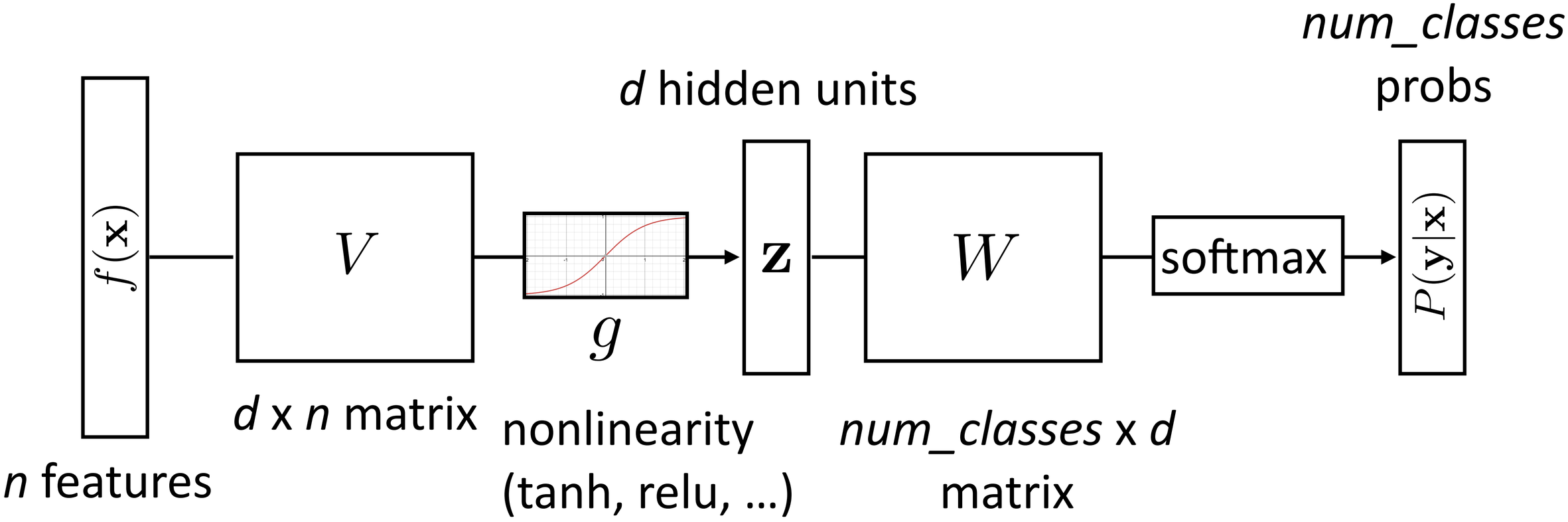
- Weight vector per class;
  W is [num classes x num feats]

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

- Now one hidden layer

Neural Networks for Classification

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



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

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

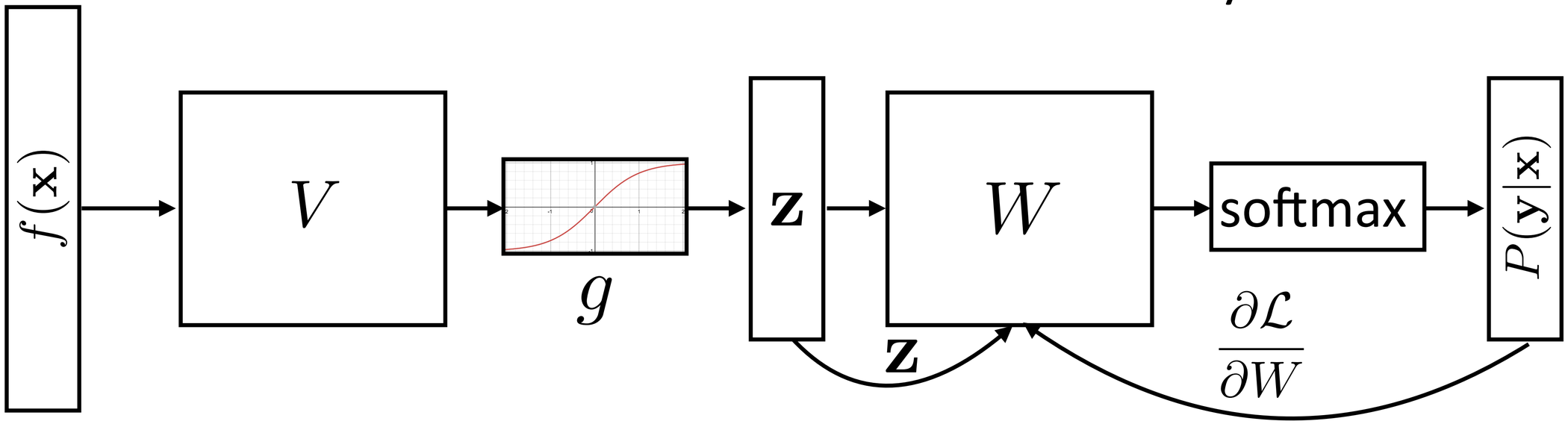
- i^*: index of the gold label
- e_i: 1 in the ith row, zero elsewhere. Dot by this = select ith index

  $[0, 0, 0, 0, \ldots, 1, 0, 0, \ldots]^T$
- This is exactly the same as logistic regression with ***z*** as the features!

Neural Networks for Classification

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

- Two sets of params to update: *W* and *V*. *W* is easy!



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

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

[some math…]

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

dim = m

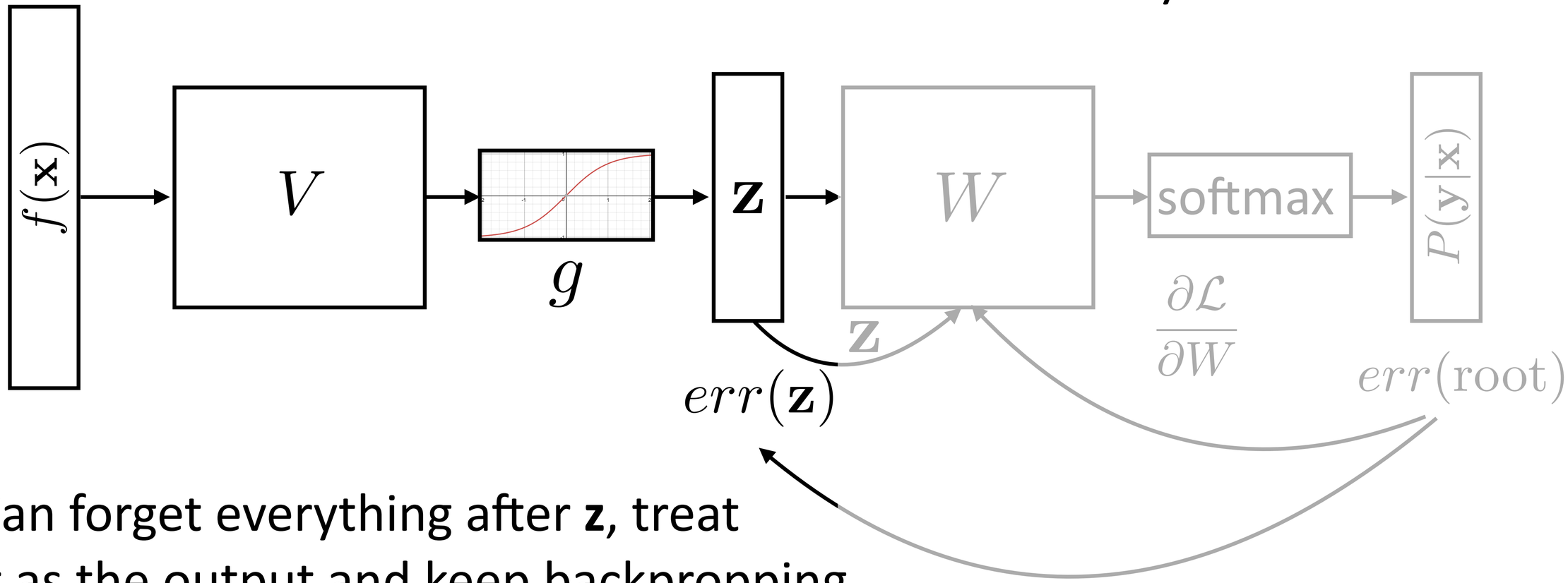
$$\boxed{\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = err(\mathbf{z}) = W^\top err(\text{root})}$$

dim = d

Backpropagation: Picture

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

- Two sets of params to update: *W* and *V*. *W* is easy!



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

Backpropagation: Takeaways

- Gradients of output weights *W* are easy to compute — looks like logistic regression with hidden layer ***z*** as feature vector
- Can compute derivative of loss with respect to ***z*** to form an "error signal" for backpropagation
- Easy to update parameters based on "error signal" from next layer, keep pushing error signal back as backpropagation
- Need to remember the values from the forward computation

Implementing NNs

Computation Graphs

- Computing gradients is hard! Computation graph abstraction allows us to define a computation symbolically and will do this for us

- Automatic differentiation: keep track of derivatives / be able to backpropagate through each function:

```
y = x * x    ──codegen──▶    (y,dy) = (x * x, 2 * x * dx)
```

- Use a library like Pytorch or Tensorflow. This class: Pytorch

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, 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)

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

Computation Graphs in Pytorch

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

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

```
ffnn = FFNN()
def make_update(input, gold_label):
    ffnn.zero_grad() # clear gradient variables
    probs = ffnn.forward(input)
    loss = torch.neg(torch.log(probs)).dot(gold_label)
    loss.backward()
    optimizer.step()
```

Training a Model

Define a computation graph

For each epoch:

For each batch of data:

Compute loss on batch

Autograd to compute gradients

Take step with optimizer

Decode test set

Simplest Neural LM

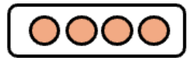
- For now: one-hot input features
- Later: word embeddings

V-dimensional vectors:

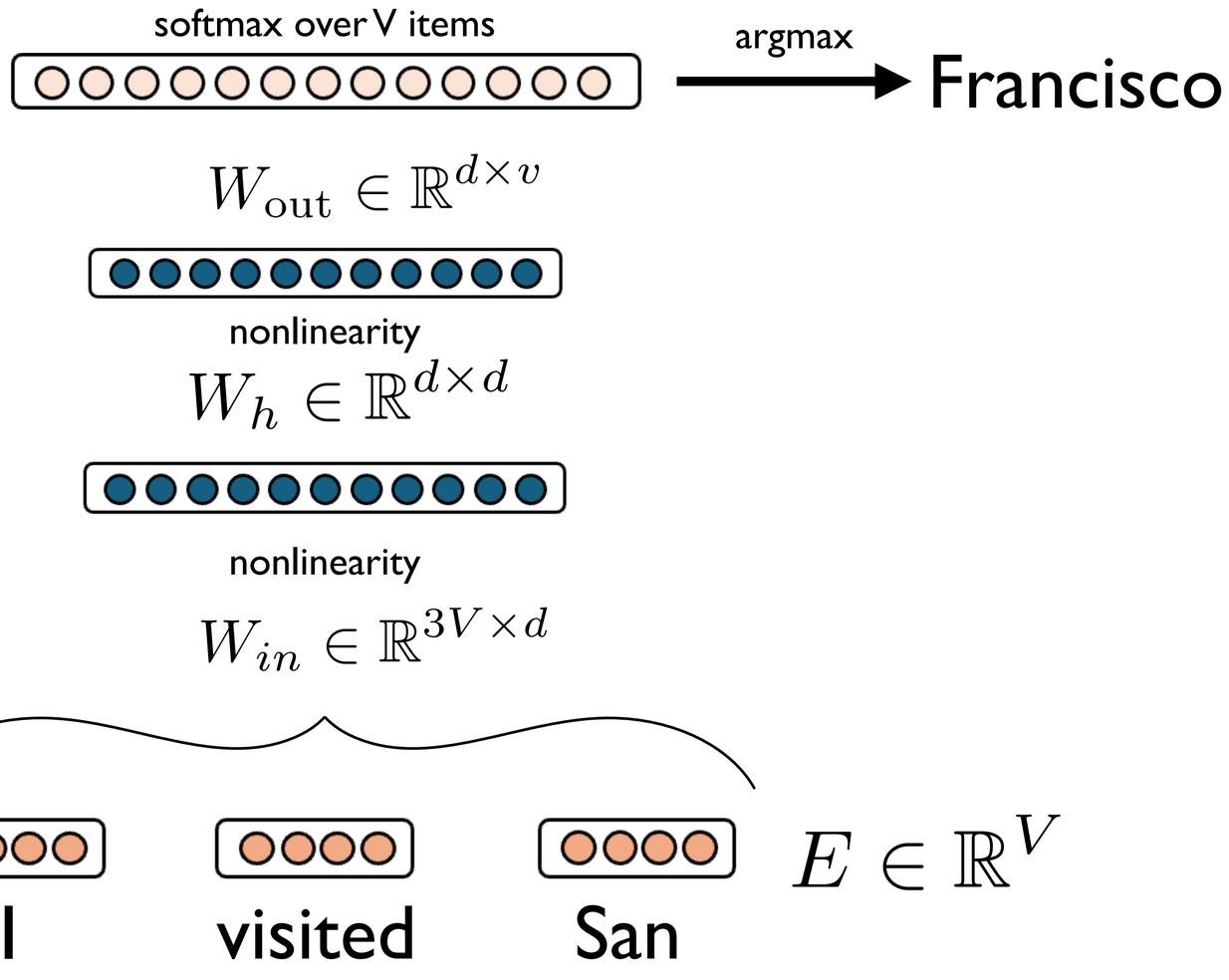
word 1: [1,0,0,…]

word 2: [0,1,0,…]

word 3: [0,0,1,…]



Simplest Neural LM



What is one key problem with this approach?

Bengio et al. (2000)

Fixed window

- We need to define a dimensionality for the input weight matrix

$$W_{in} \in \mathbb{R}^{3V \times d}$$

- If we want more context —> bigger weight matrix, more expensive computation $W_{in} \in \mathbb{R}^{4V \times d}$ $W_{in} \in \mathbb{R}^{5V \times d}$
- Fundamentally: Language involves dealing with variable-length inputs!
- Later: recurrence as a way of addressing this, embeddings as a way of reducing dimensionality

Training Tips

Adam

- Most common optimizer in practice so we will spend more time on understanding intuitions behind it
- Key components to understand:
  - Momentum
  - RMSProp

RMSProp

- Core problem: sign of the gradient tells us where to go, but not how big of a step
- Gradient magnitudes swing wildly during training
- Idea: normalize the magnitude along each dimension of the gradient

$$s_k \leftarrow \beta s_{k-1} + (1-\beta)(\nabla_\theta \mathcal{L}(\theta_k))^2$$

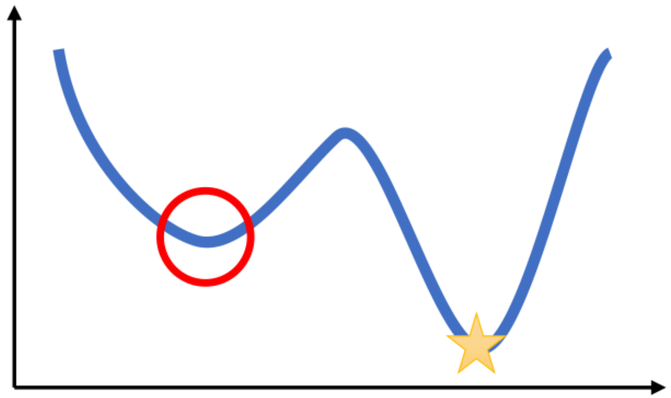
$$\theta_{k+1} \leftarrow \theta_k - \alpha \frac{\nabla_\theta \mathcal{L}(\theta_k)}{\sqrt{s_k}}$$

Momentum

- "Blend-in" a running average of the gradient direction
- Gradient "keeps going" a bit in the direction it was headed, preventing wild swings and local optima
  - Think of a rolling ball

$$g_k = \nabla_\theta \mathcal{L}(\theta_k) + \mu g_{k-1}$$

"blend in" some of the previous gradient



from https://cs182sp21.github.io/static/slides/lec-4.pdf

Adam

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)

- *m*: exponentially-weighted moving average of gradients (momentum)
- *v*: exponentially-weighted moving average of gradients squared (RMS)
- $\beta_1 = 0.9$, $\beta_2 = 0.999$, so these average over many steps
- Update is based on normalized corrected mean, incorporates *momentum*

Kingma and Ba (2015)

Batching

- Batching data 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))
    ...
```

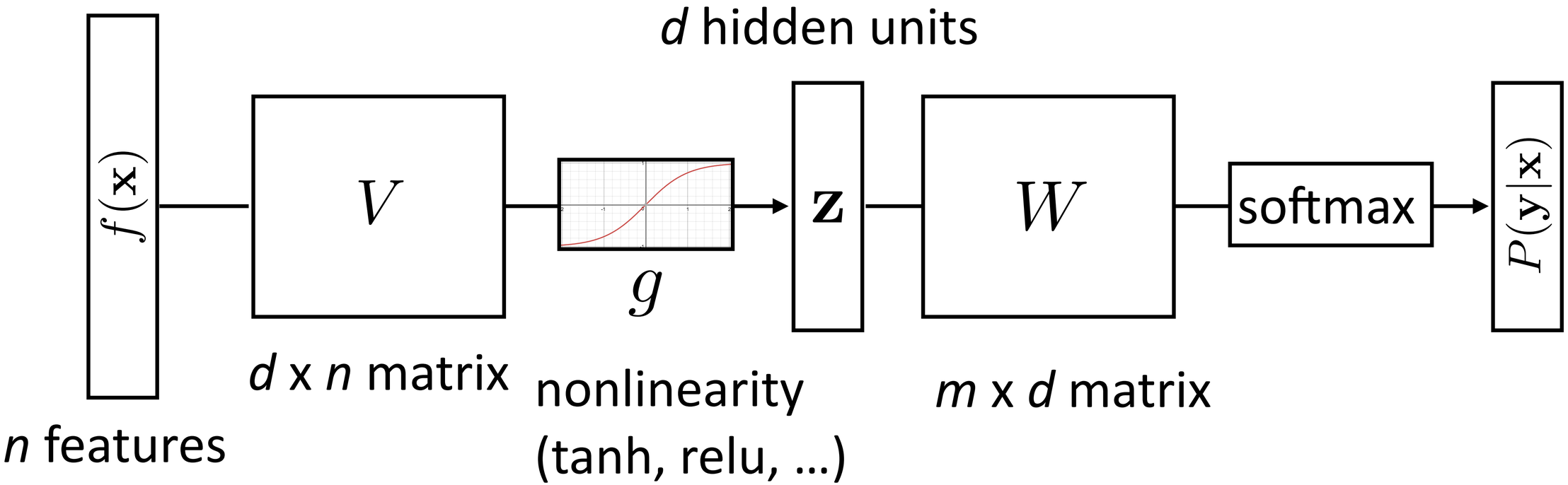
- Batch sizes from 1-100 often work well

Training Basics

- Basic formula: compute gradients on batch, use first-order optimization method (SGD, Adagrad, etc.)
- How to initialize? How to regularize? What optimizer to use?

How does initialization affect learning?

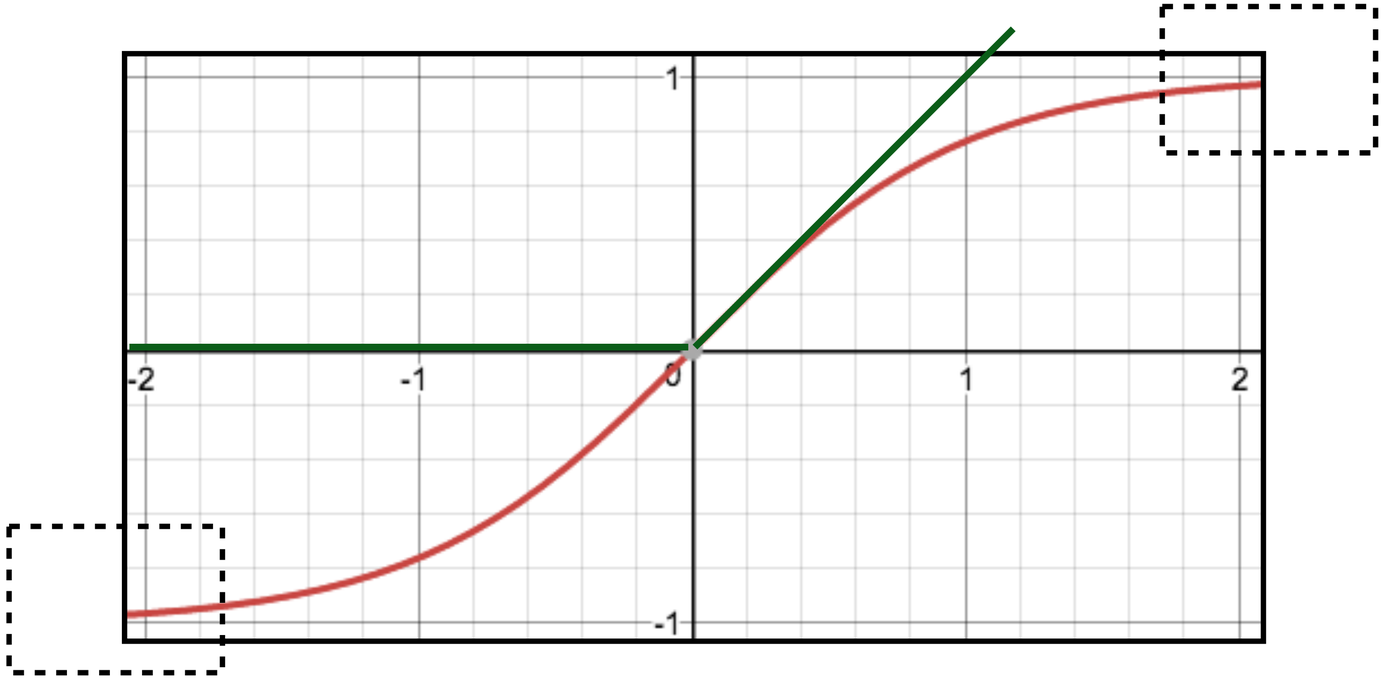
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W g(V f(\mathbf{x})))$$



- How do we initialize V and W? What consequences does this have?
- *Nonconvex* problem, so initialization matters!

How does initialization affect learning?

- Nonlinear model…how does this affect things?



- If cell activations are too large in absolute value, gradients are small
- ReLU: larger dynamic range (all positive numbers), but can produce big values, can break down if everything is too negative

Initialization

1) Can't use zeroes for parameters to produce hidden layers: all values in that hidden layer are always 0 and have gradients of 0, never change

2) Initialize too large and cells are saturated

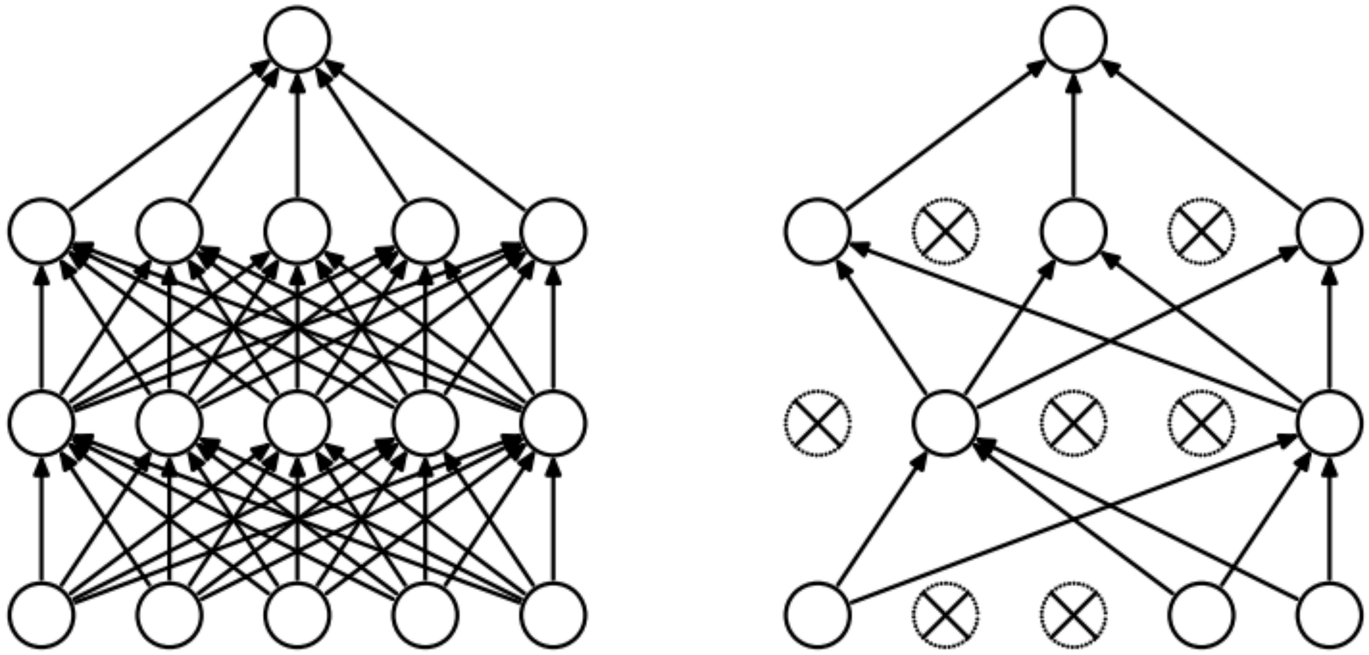
- Can do random uniform / normal initialization with appropriate scale
- Glorot initializer: $U\left[-\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}, +\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}\right]$
  - Want variance of inputs and gradients for each layer to be the same
- Batch normalization (Ioffe and Szegedy, 2015): periodically shift+rescale each layer to have mean 0 and variance 1 over a batch (useful if net is deep)

Dropout

- Probabilistically zero out parts of the network during training to prevent overfitting, use whole network at test time

- Form of stochastic regularization

- Similar to benefits of ensembling: network needs to be robust to missing signals, so it has redundancy



(a) Standard Neural Net (b) After applying dropout.

- One line in Pytorch/Tensorflow

Srivastava et al. (2014)

Next Time: Word Representations

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

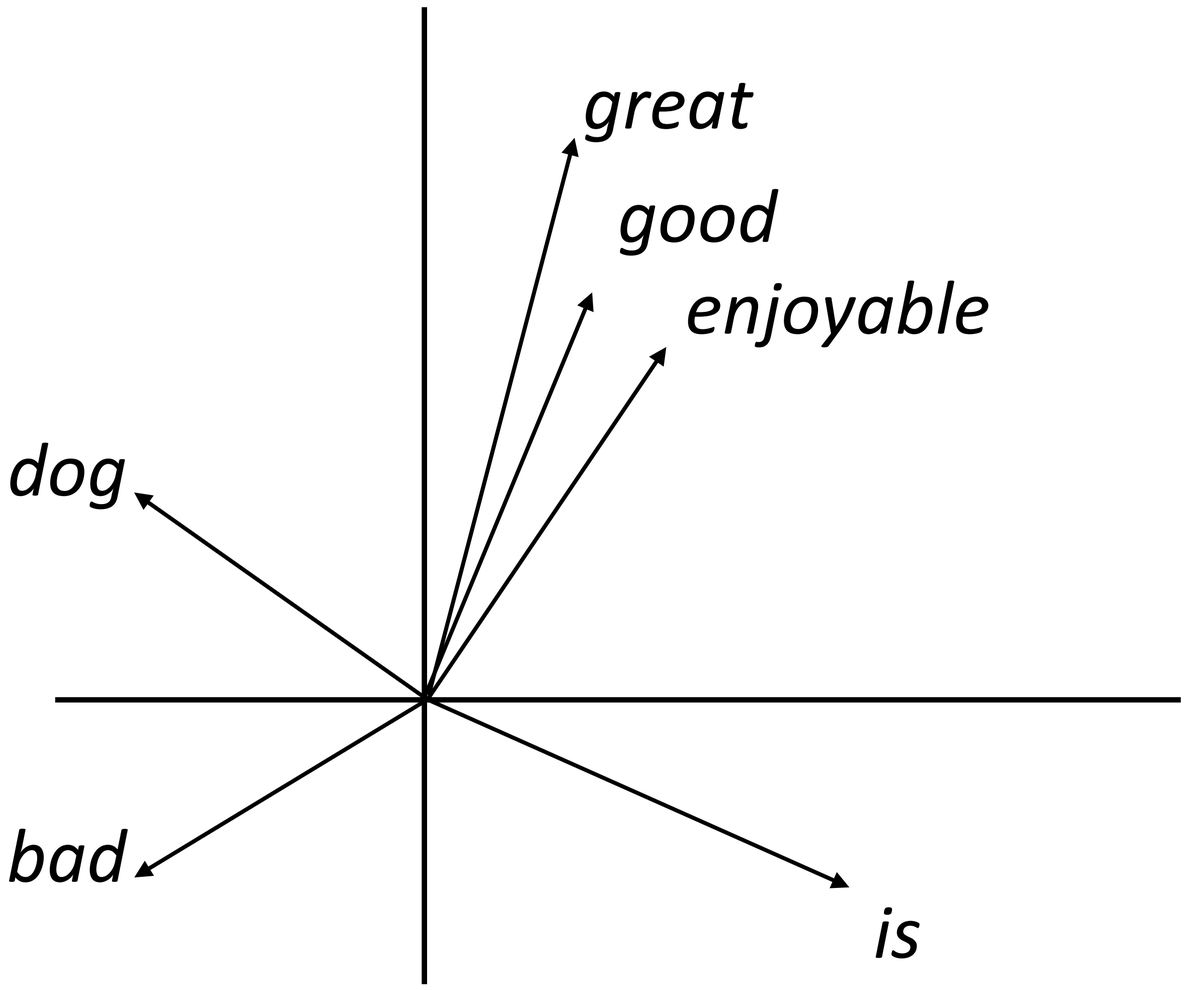
- Want a vector space where similar words have similar embeddings

the movie was great

≈

the movie was good

- Goal: come up with a way to produce these embeddings
- For each word, want "medium" dimensional vector (50-300 dims) representing it



Takeaways

- Feedforward neural networks can be implemented easily in PyTorch
  - We saw that these are basically logistic regression
  - Easy to implement backpropagation (you don't have to do anything!) and use the standard tricks to get good performance

- Next class: thinking about the feature representations: word representations / word vectors (word2vec and GloVe)