Neural Networks

Wei Xu

(many slides from Greg Durrett)

Administrivia

- Reading: Eisenstein 2.6, 3.1-3.3, J+M 7, Goldberg 1-4
- PS1 is released.
- PyTorch Tutorial can be found on the course website!

This and Next Lectures

- Neural network history
- Neural network basics
- Feedforward neural networks
- Applications
- Training of neural networks - backpropagation, more optimization
- Implementing neural networks

A Bit of History

- The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.
- Perceptron (Frank Rosenblatt, 1957)
- Artificial Neuron (McCulloch & Pitts, 1943)

McCulloch Pitts Neuron
(assuming no inhibitory inputs)

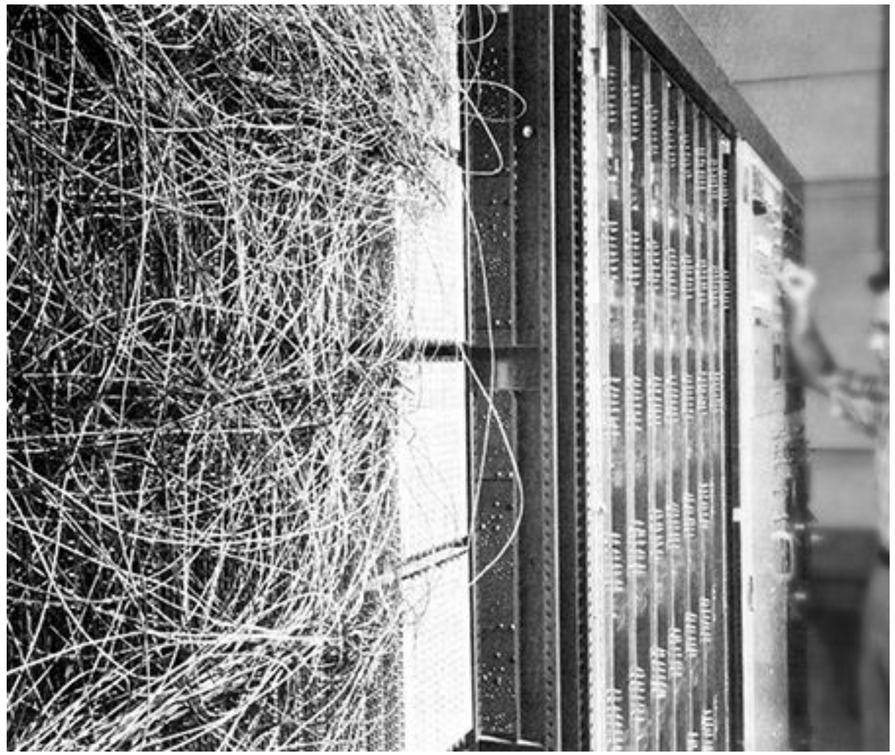
$$y = 1 \quad if \sum_{i=0}^{n} x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} x_i < 0$$

Perceptron

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

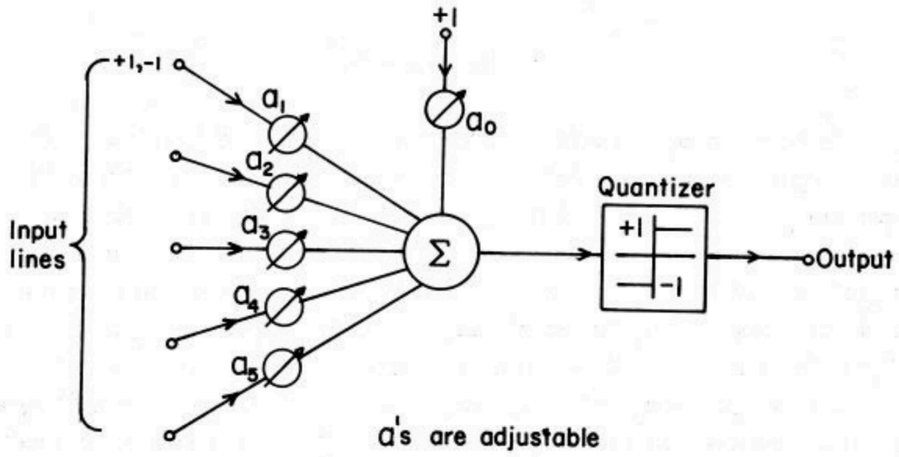


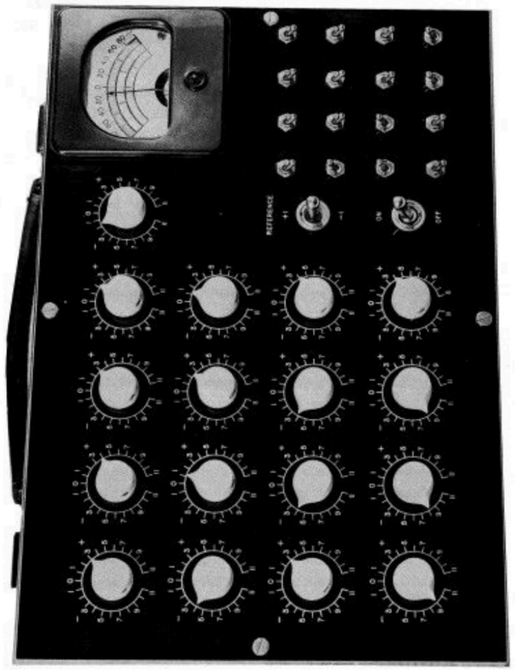
The IBM Automatic Sequence Controlled Calculator, called Mark I by Harvard University's staff. It was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors.

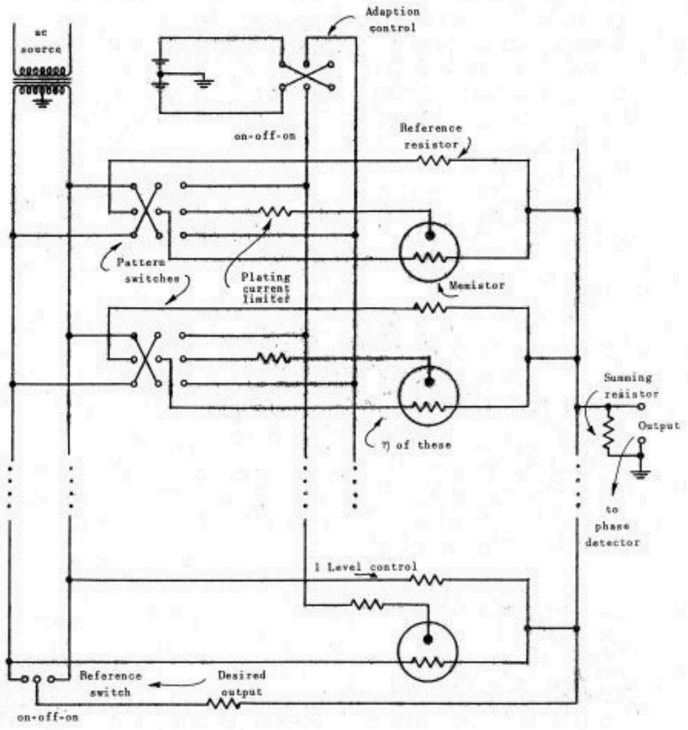
https://www.youtube.com/watch?time_continue=71&v=cNxadbrN_aI&feature=emb_logo

A Bit of History

- Adaline/Madeline - single and multi-layer “artificial neurons” (Widrow and Hoff, 1960)







A Bit of History

- First time back-propagation became popular (Rumbelhart et al, 1986)

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \quad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

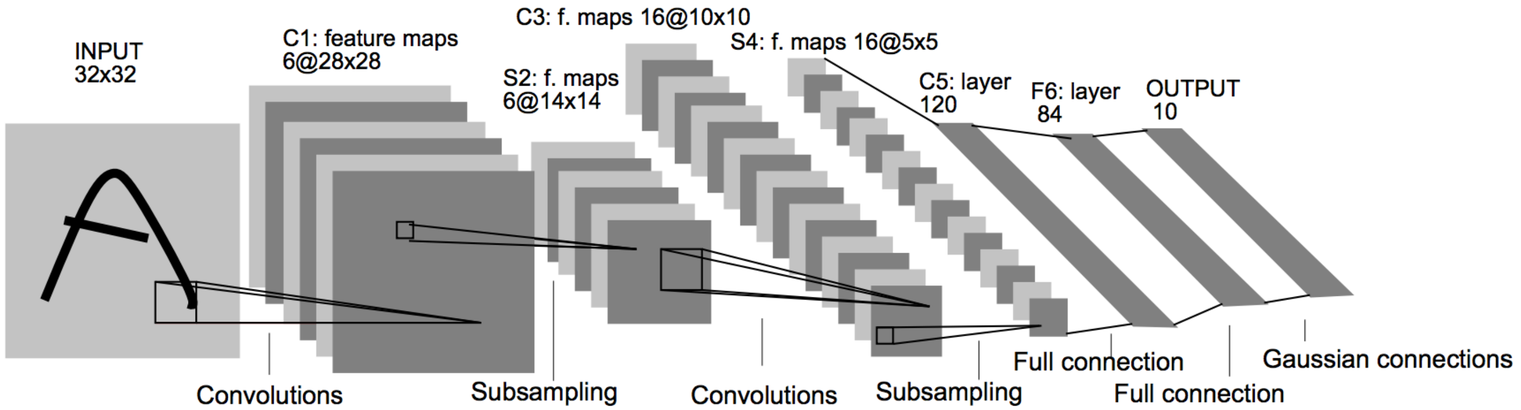
$$y_j = \frac{1}{1+e^{-x_j}} \quad (2)$$

† To whom correspondence should be addressed.

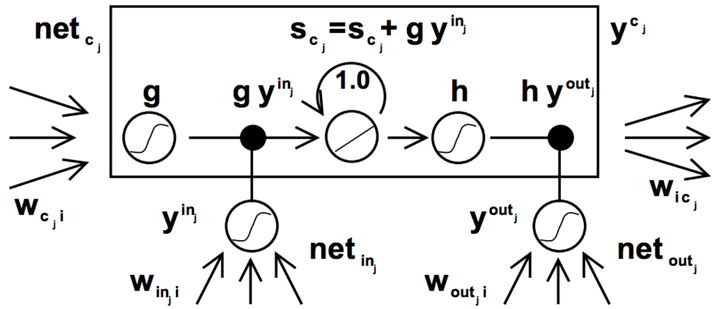
https://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/

History: NN “dark ages”

- ConvNets: applied to MNIST by LeCun in 1990s



- LSTMs: Hochreiter and Schmidhuber (1997)

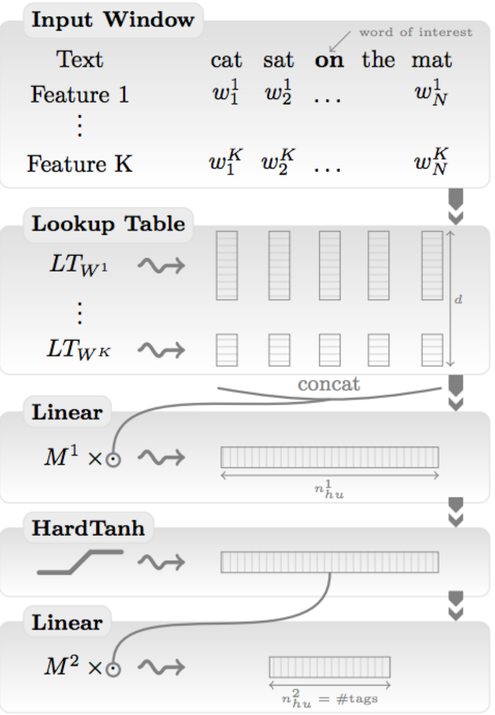


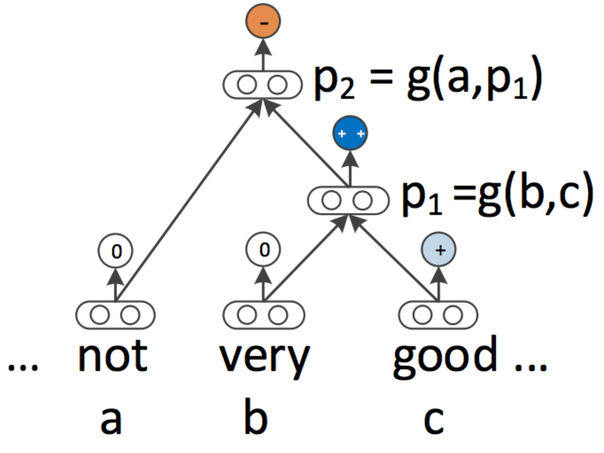
- Henderson (2003): neural shift-reduce parser, not SOTA

https://www.youtube.com/watch?v=FwFduRA_L6Q&feature=youtu.be

2008-2013: A glimmer of light…

- Collobert and Weston 2011: “NLP (almost) from scratch”
  - Feedforward neural nets induce features for sequential CRFs (“neural CRF”)
  - 2008 version was marred by bad experiments, claimed SOTA but wasn’t, 2011 version tied SOTA
- 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 (CNNs work for NLP?)
- Sutskever et al. (2014) + Bahdanau et al. (2015) : seq2seq + attention for neural MT (LSTMs work for NLP?)
- Chen and Manning (2014) transition-based dependency parser (even feedforward networks work well for NLP?)
- 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 (2012) is pretty helpful
  - **Computers not big enough**: can’t run for enough iterations
- **Inputs**: need word representations to have the right continuous semantics
- **Libraries**: TensorFlow (first released in Nov 2015), PyTorch (Sep 2016)

GPU server


Lambda
Lambda
Hypercluster
FRAGILE
KEEP
DRY
CAUTION
HEAVY RACK
2 MAN OPEN/CLOSE
LLB-APCAR3300

Neural Net Basics

Linear Transformation (math review)

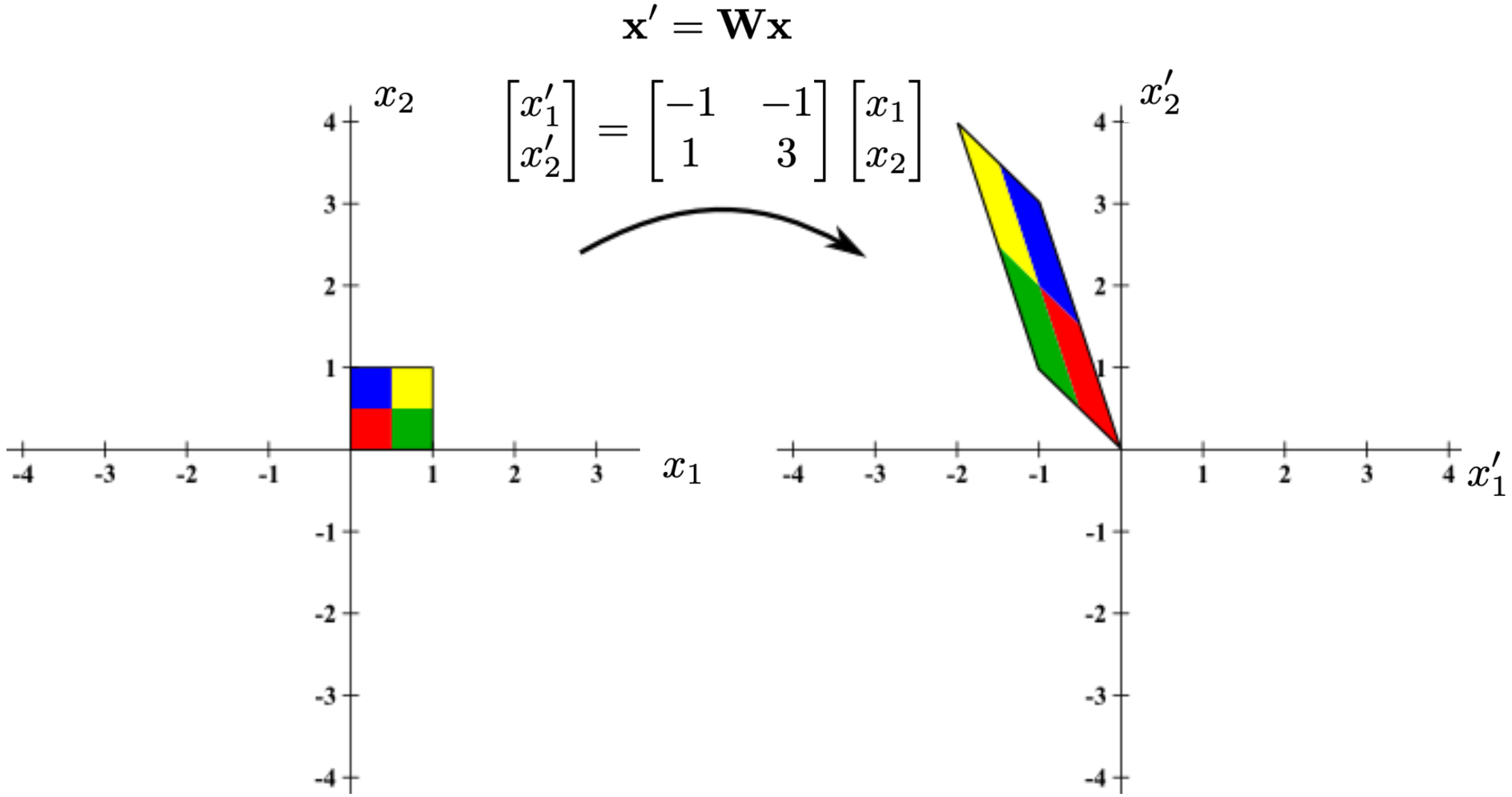


Image adopted from Duane Q. Nykamp

Neural Networks: motivation

- Linear classification: $\text{argmax}_y w^\top f(x, y)$
- How can we do nonlinear classification? Kernels are too slow…
- Want to learn intermediate conjunctive features of the input

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

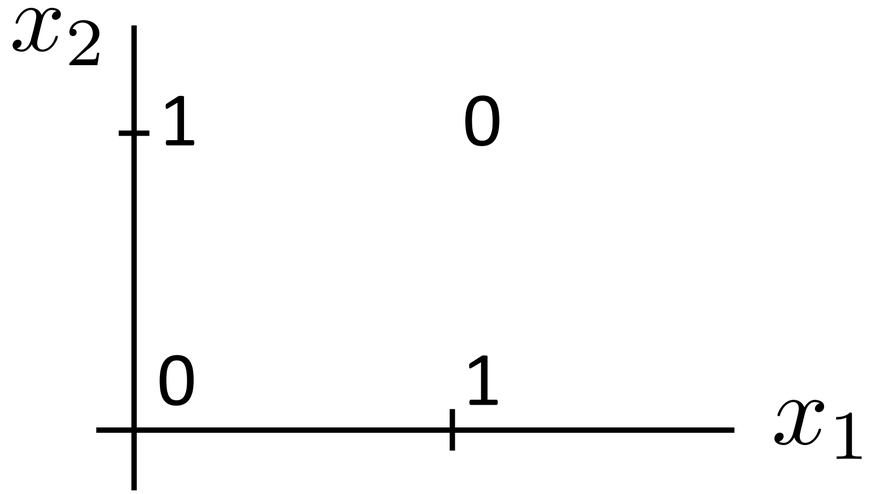
$\mathbb{I}$[contains *not* & contains *good*]

Neural Networks: XOR

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

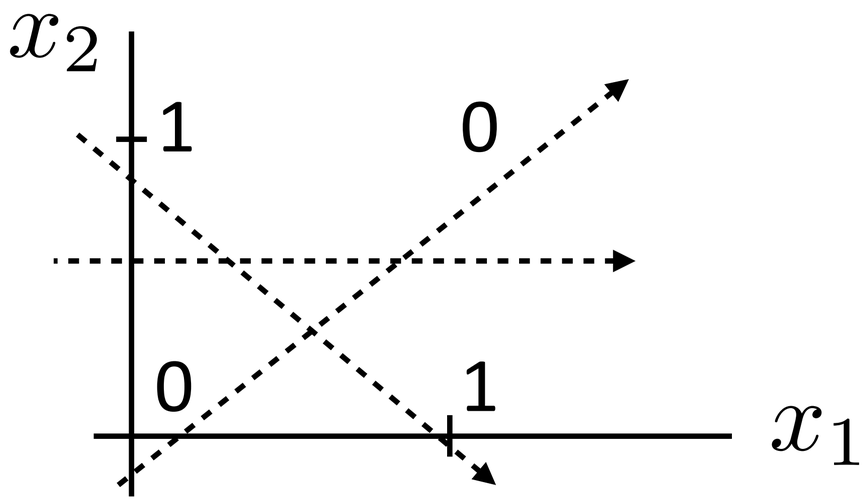
  $(\text{generally } \mathbf{x} = (x_1, \ldots, x_m))$
- Output y

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



x_1	x_2	$y = x_1$ XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



$$y = a_1 x_1 + a_2 x_2$$

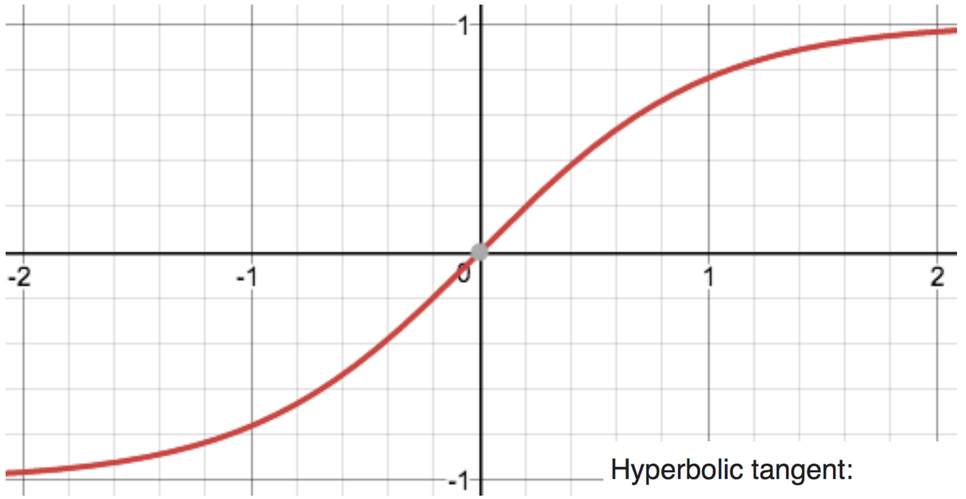
X

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

"or"

(looks like action potential in neuron)

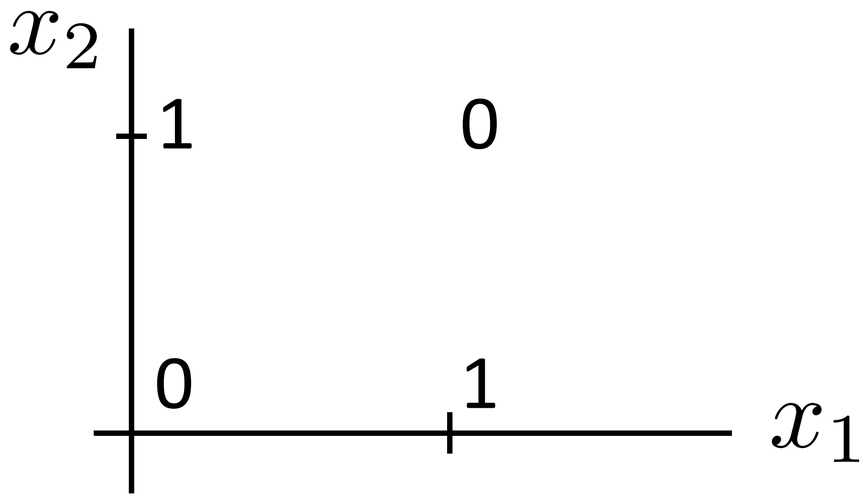
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0



Hyperbolic tangent:

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Neural Networks: XOR



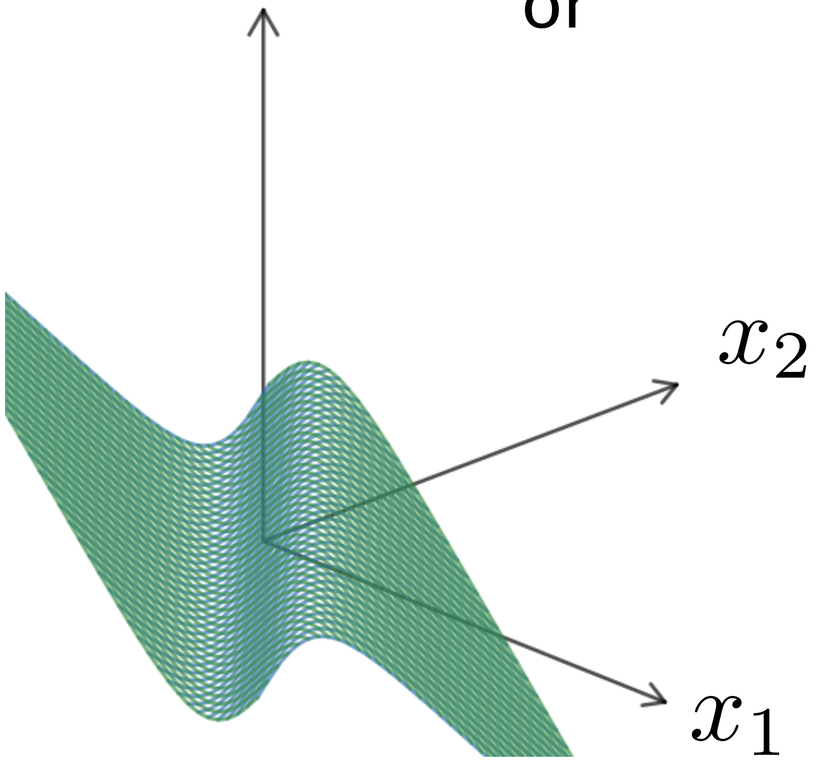
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

$$y = a_1 x_1 + a_2 x_2$$ X

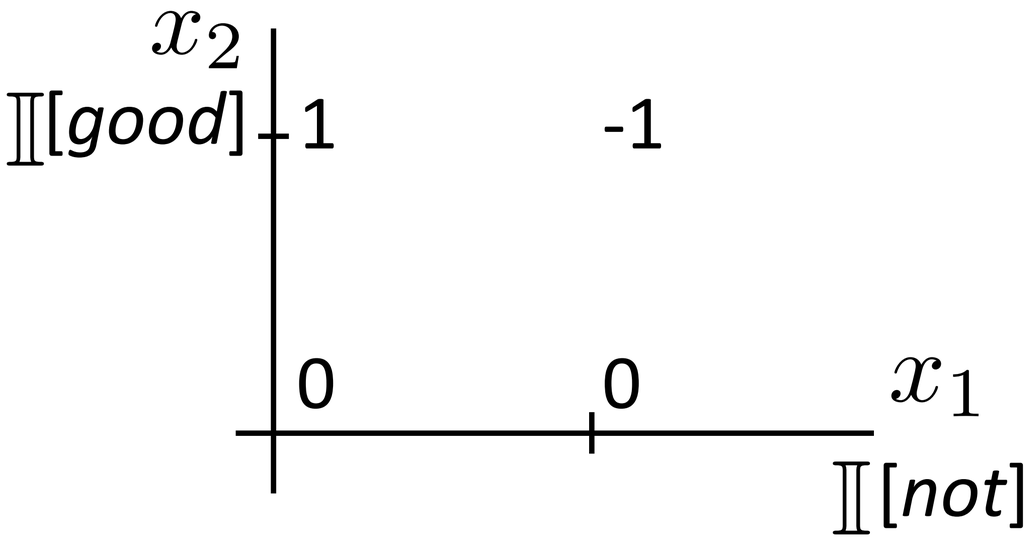
$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$ ✓

$$y = -x_1 - x_2 + 2 \tanh(x_1 + x_2)$$

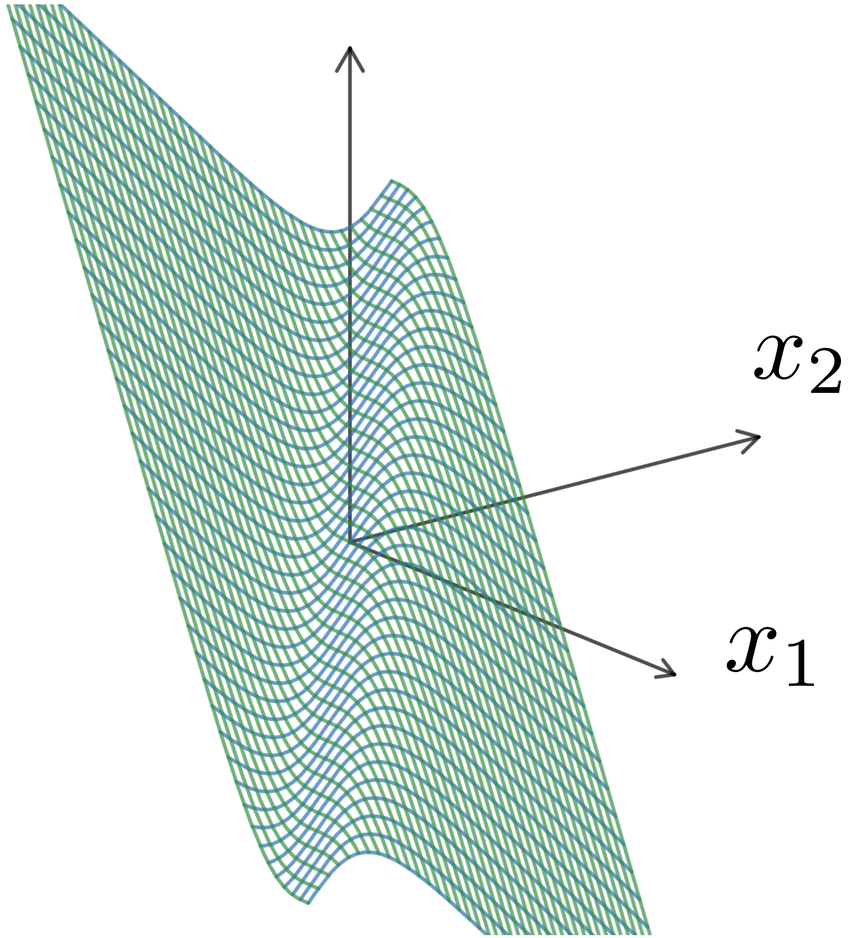
“or”



Neural Networks: XOR



$$y = -2x_1 - x_2 + 2\tanh(x_1 + x_2)$$



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

Neural Networks

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

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

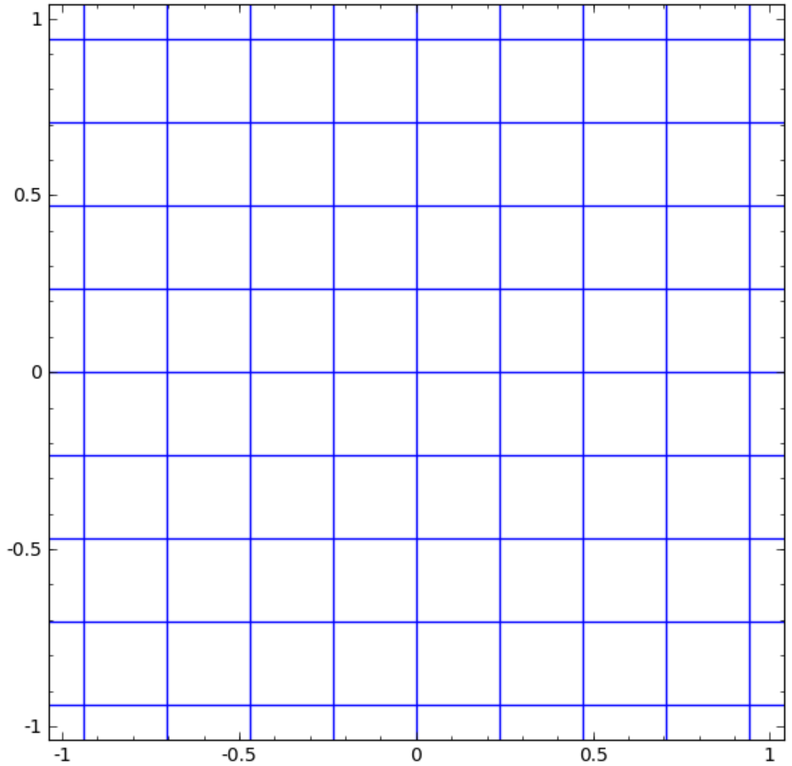
$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Nonlinear transformation

Warp space

Shift

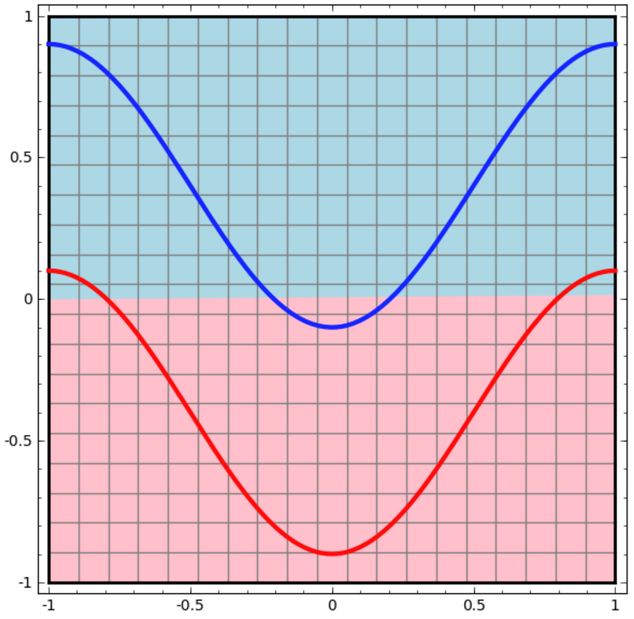
tanh



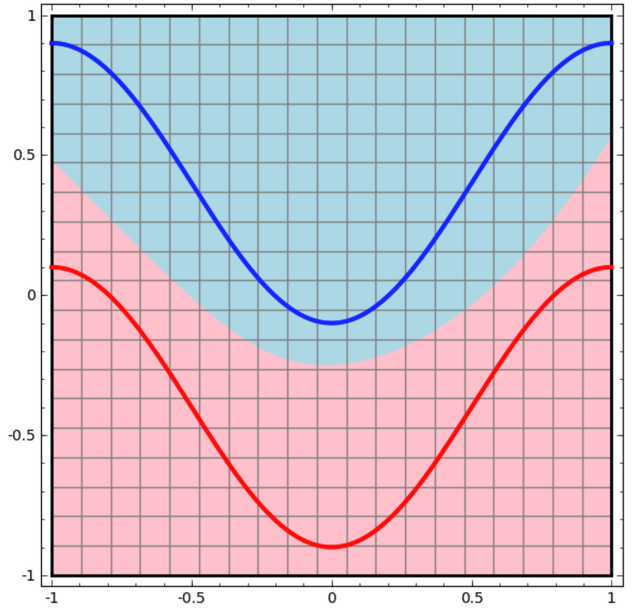
Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Neural Networks

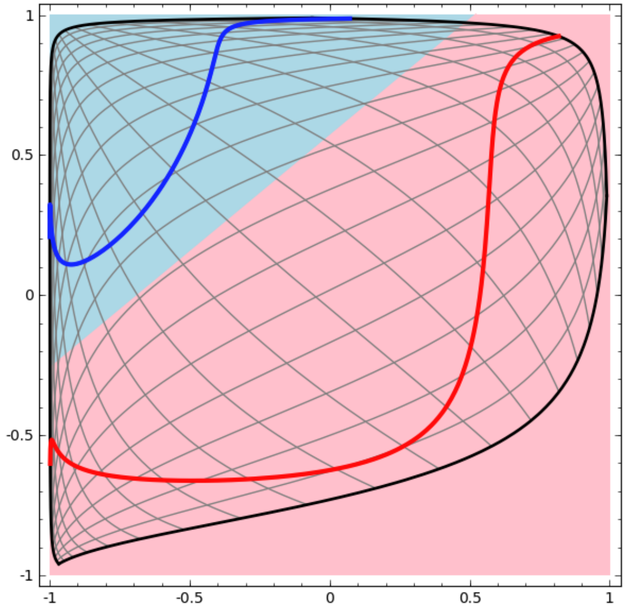
Linear classifier



Neural network

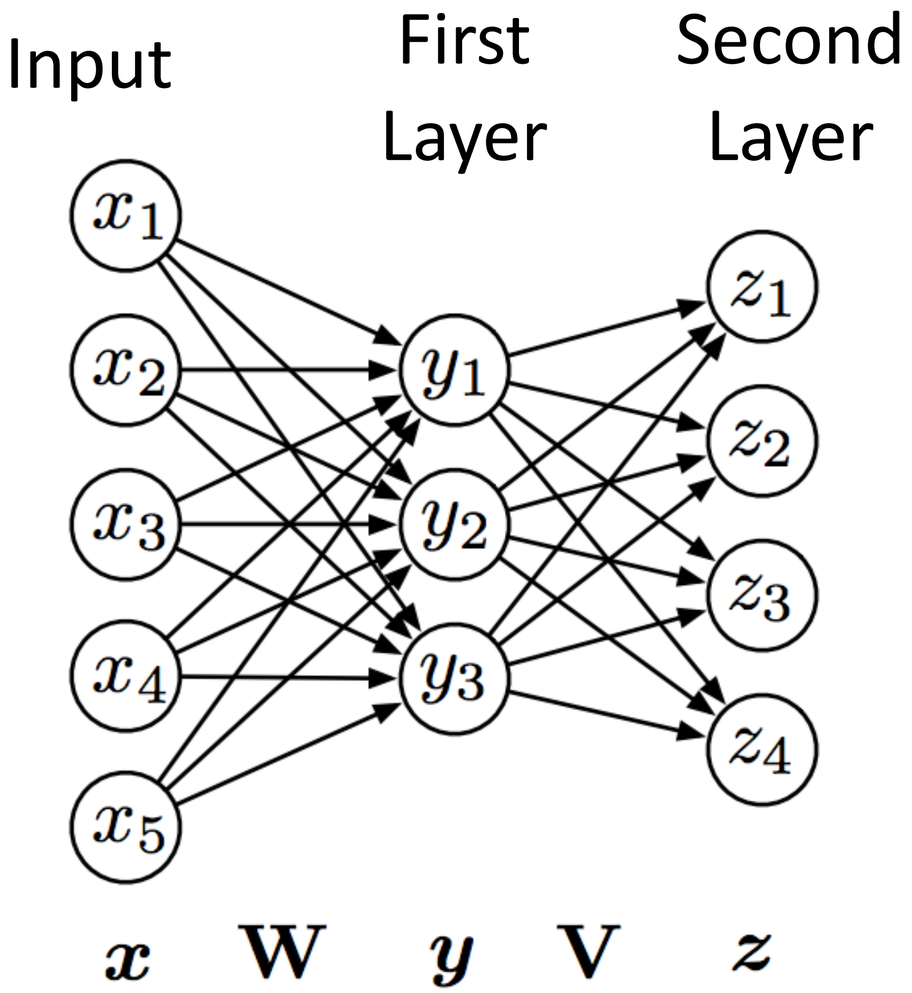


…possible because we transformed the space!



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

Deep Neural Networks



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

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

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

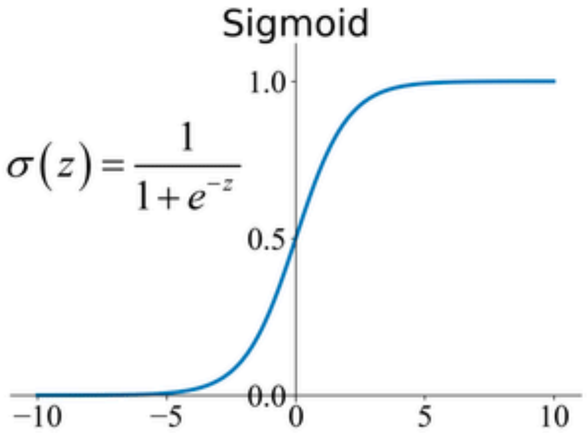
"Feedforward" computation (not recurrent)

Check: what happens if no nonlinearity?
More powerful than basic linear models?

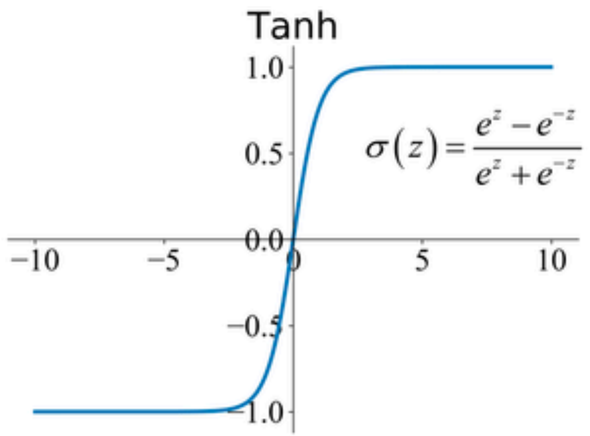
$$\mathbf{z} = \mathbf{V}(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

Adopted from Chris Dyer

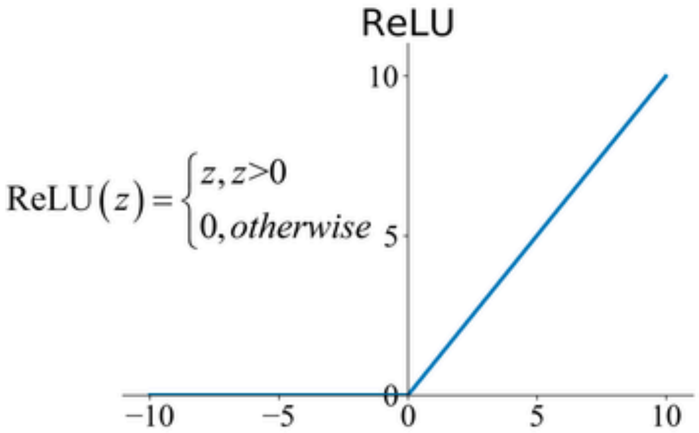
Activation Functions



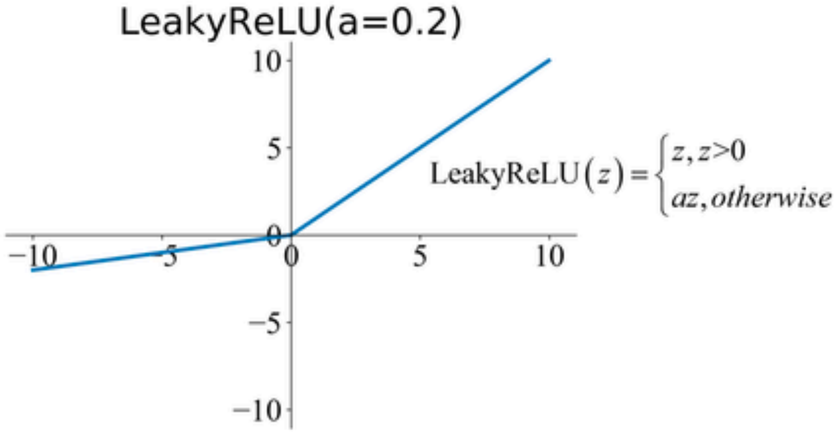
(a)



(b)



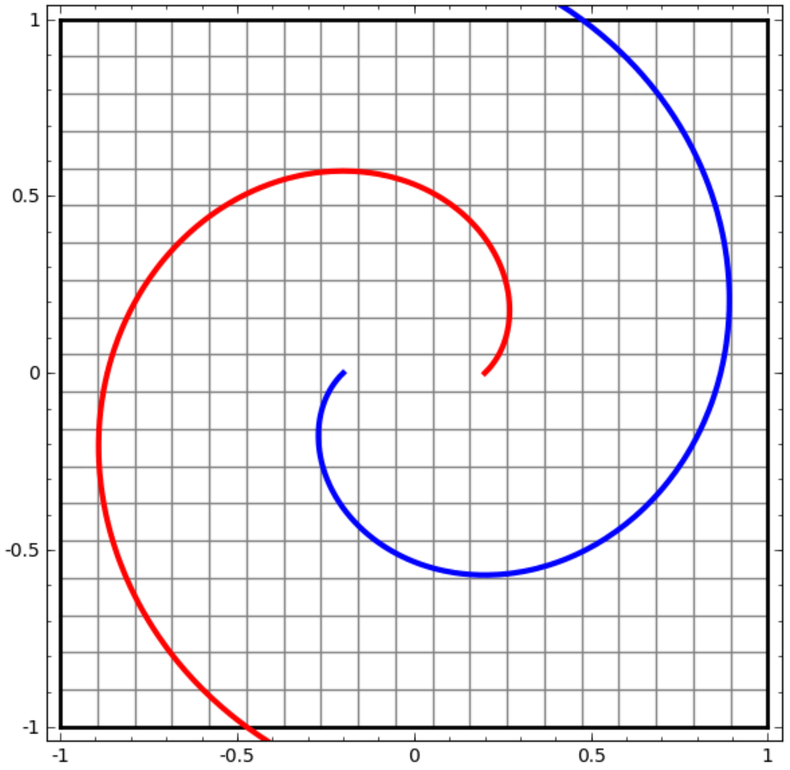
(c)



(d)

Image Credit: Junxi Feng

Deep Neural Networks



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

Feedforward Networks, Backpropagation

Recap: Multiclass Logistic Regression

$$P_w(y|x) = \frac{\exp\left(w^\top f(x,y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(w^\top f(x,y')\right)}$$

sum over output space to normalize

too many drug trials, too few patients

Health: +2.2

Sports: +3.1

Science: -0.6

$w^\top f(x,y)$

exp

probabilities must be >= 0

6.05

22.2

0.55

unnormalized probabilities

normalize

probabilities must sum to 1

0.21

0.77

0.02

probabilities

log(0.21) = - 1.56

compare

$\mathcal{L}(x_j, y_j^*) = \log P(y_j^*|x_j)$

1.00

0.00

0.00

correct (gold) probabilities

Logistic Regression with NNs

$$P(y|\mathbf{x}) = \frac{\exp(w^\top f(\mathbf{x}, y))}{\sum_{y'} \exp(w^\top f(\mathbf{x}, y'))}$$

- Single scalar probability

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}\left([w^\top f(\mathbf{x}, y)]_{y \in \mathcal{Y}}\right)$$

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

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

- softmax: exps and normalizes a given vector

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

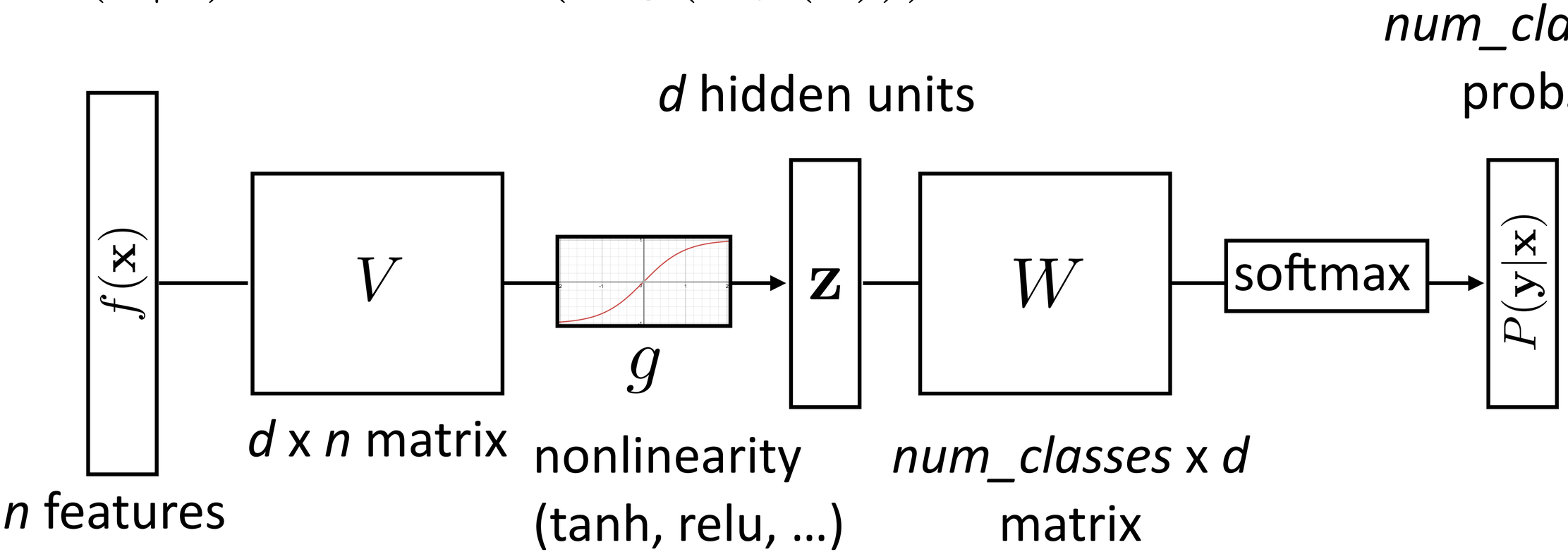
- Weight vector per class; W is [num classes x num feats]

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

- Now one hidden layer

Neural Networks for Classification

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



We can think of a neural network classifier with one hidden layer as building a vector z which is a hidden layer representation (i.e. latent features) of the input, and then running standard logistic regression on the features that the network develops in z.

Training Neural Networks

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

- 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

one-hot vector

Training Neural Networks

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

- 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

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

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*

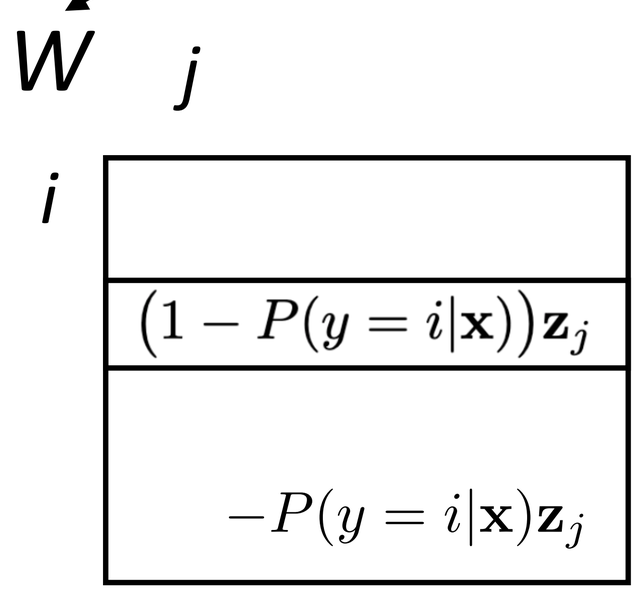
index of output space $\mathcal{Y}$

$$\frac{\partial}{\partial W_{ij}} \mathcal{L}(\mathbf{x}, i^*) = \begin{cases} \big(1 - P(y = i|\mathbf{x})\big)\mathbf{z}_j & \text{if } i = i^* \\ -P(y = i|\mathbf{x})\mathbf{z}_j & \text{otherwise} \end{cases}$$

index of gold label

index of vector **z**

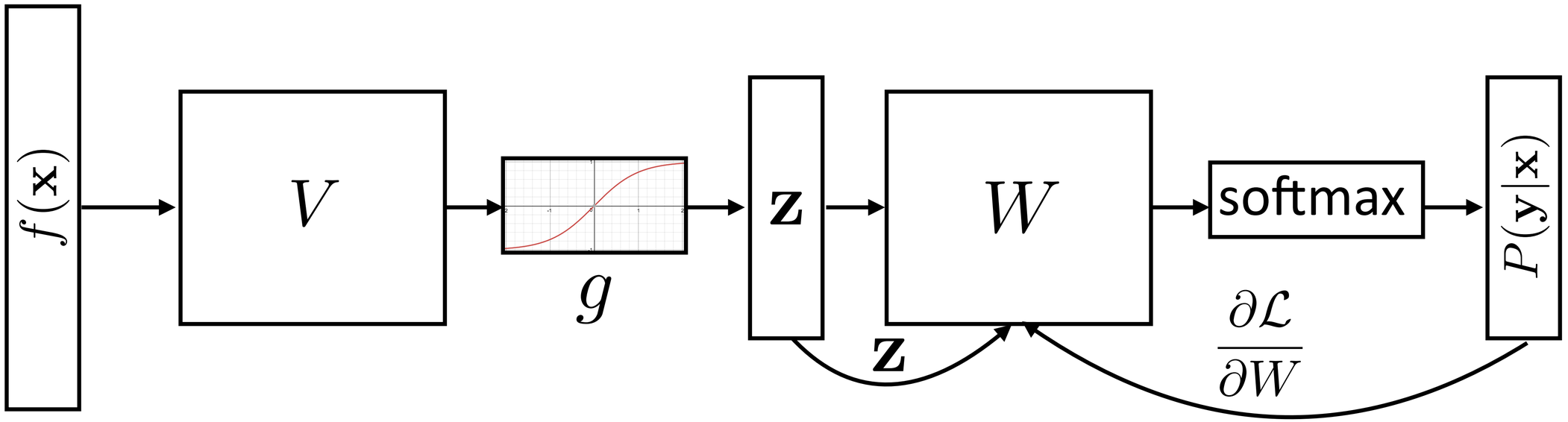
num_classes x *d* matrix



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

Neural Networks for Classification

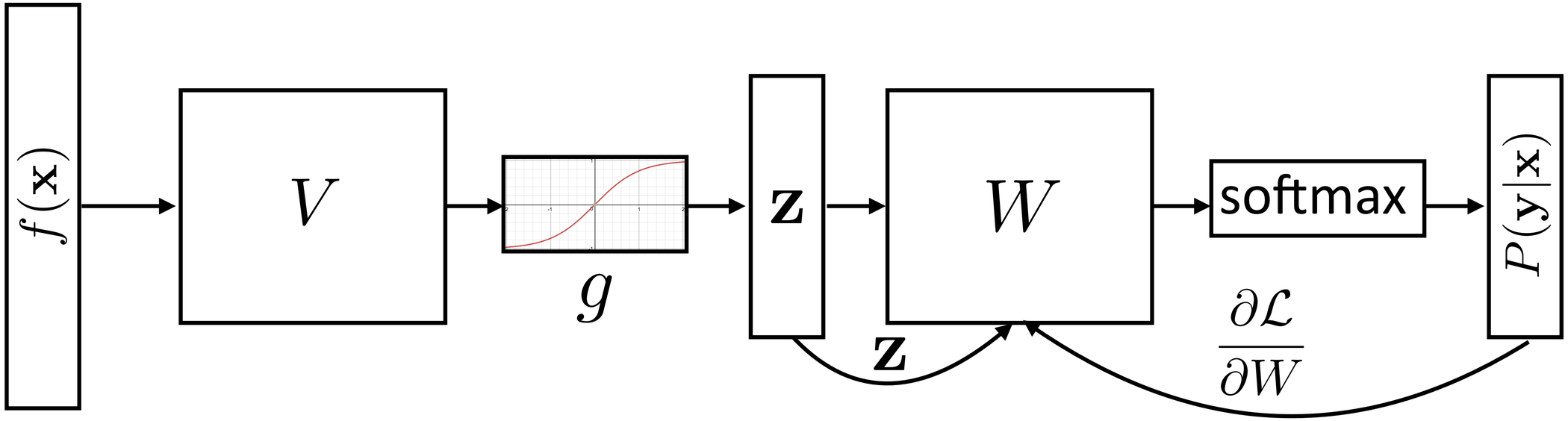
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W\, g(V\, f(\mathbf{x})))$$



- Gradient w.r.t. W: looks like logistic regression with ***z*** as the features!

Neural Networks for Classification

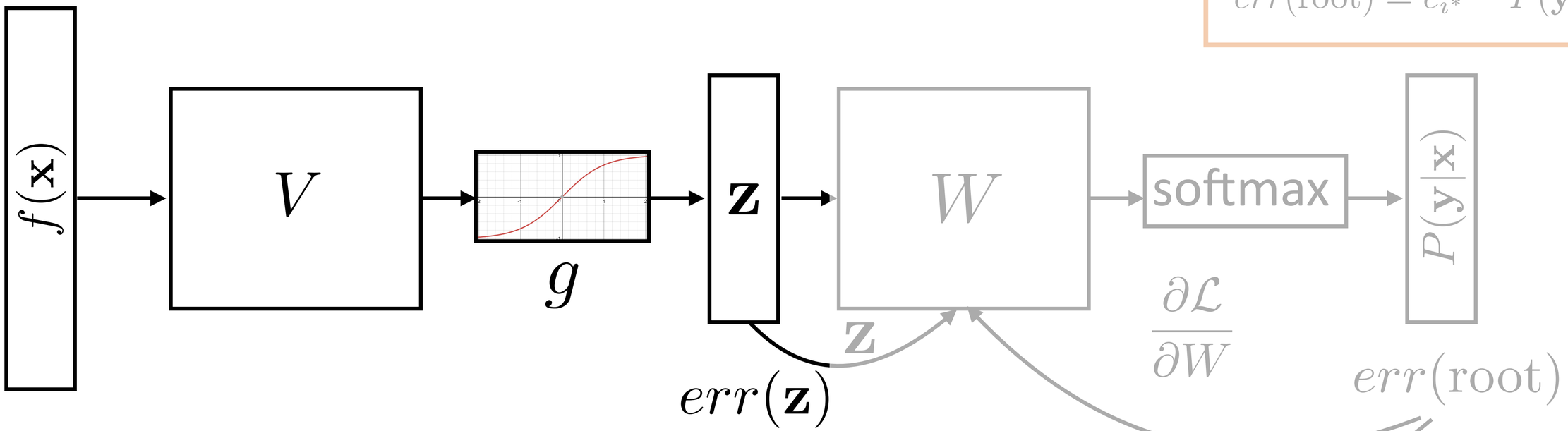
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W\, g(V\, f(\mathbf{x})))$$



$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{W}} = \mathbf{z}(e_{i^*} - P(\mathbf{y}|\mathbf{x})) = \mathbf{z} \cdot err(\text{root})$$

Backpropagation: Picture

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



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

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

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

$\mathbf{z} = g(V f(\mathbf{x}))$
Activations at hidden layer

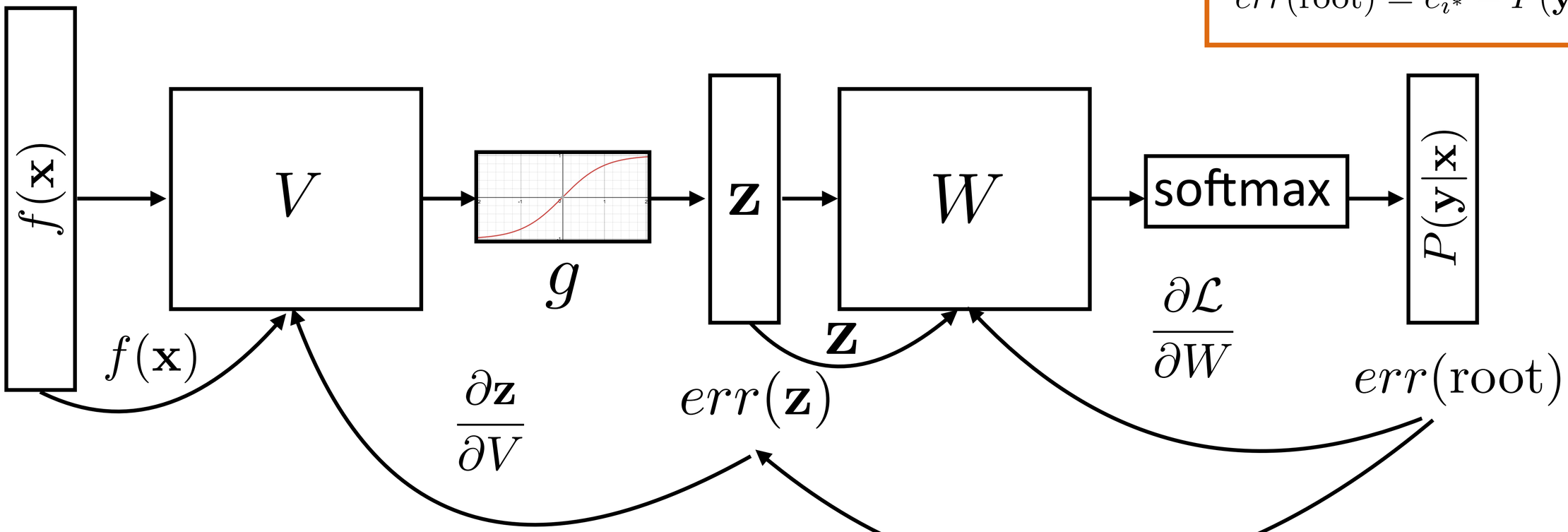
- Gradient with respect to *V*: apply the chain rule

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

- First term: gradient of nonlinear activation function at ***a*** (depends on current value)
- Second term: gradient of linear function
- Straightforward computation once we have *err*(**z**)

Backpropagation: Picture

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



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

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

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

Backpropagation

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

- 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 there are more hidden layers …)

Backpropagation: Takeaways

- Gradients of output weights W are easy to compute — looks like logistic regression with hidden layer $\boldsymbol{z}$ as feature vector
- Can compute derivative of loss with respect to $\boldsymbol{z}$ 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

Applications

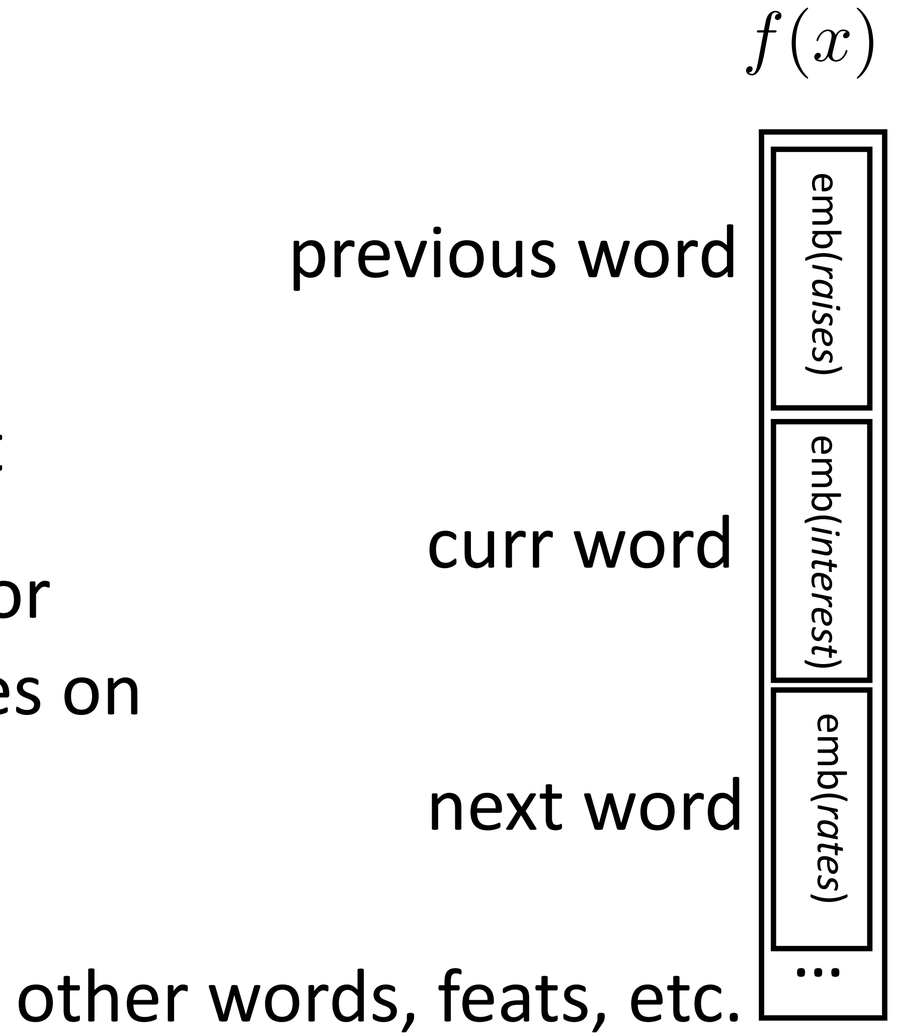
NLP with Feedforward Networks

- Part-of-speech tagging with FFNNs

??

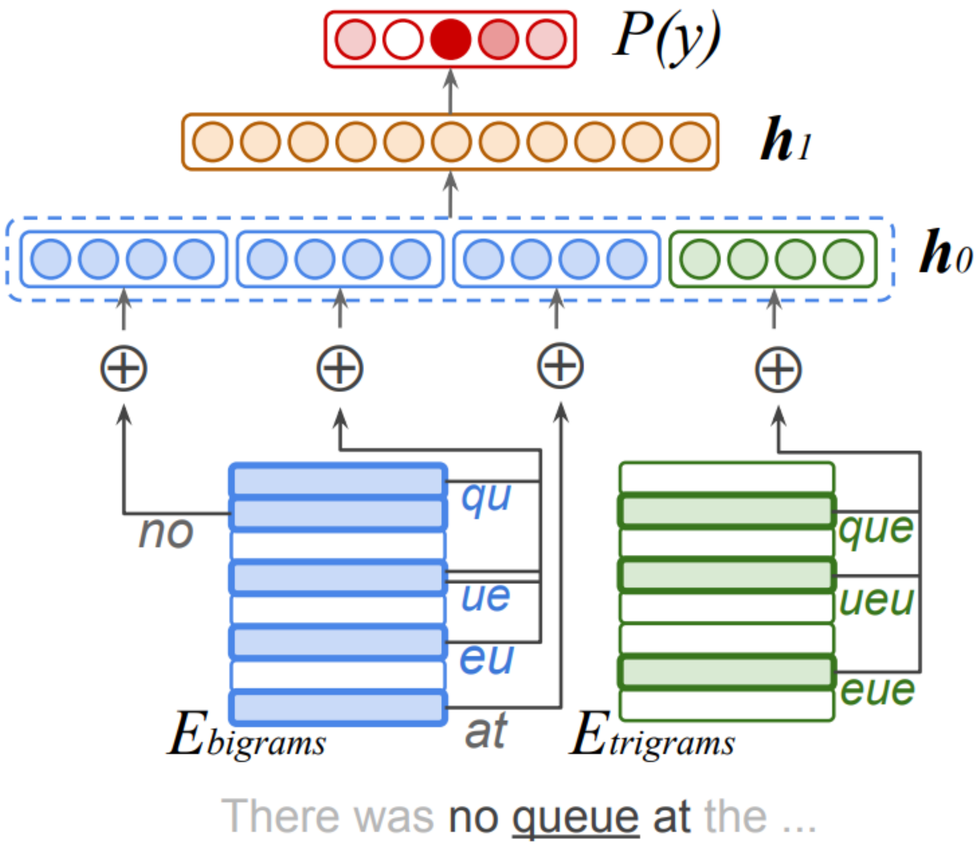
*Fed raises **interest** rates in order to …*

- Word embeddings for each word form input
- ~1000 features here — smaller feature vector than in sparse models, but every feature fires on every example
- Weight matrix learns position-dependent processing of the words



Botha et al. (2017)

NLP with Feedforward Networks



- Hidden layer mixes these different signals and learns feature conjunctions

Botha et al. (2017)

NLP with Feedforward Networks

- Multilingual POS tagging results:

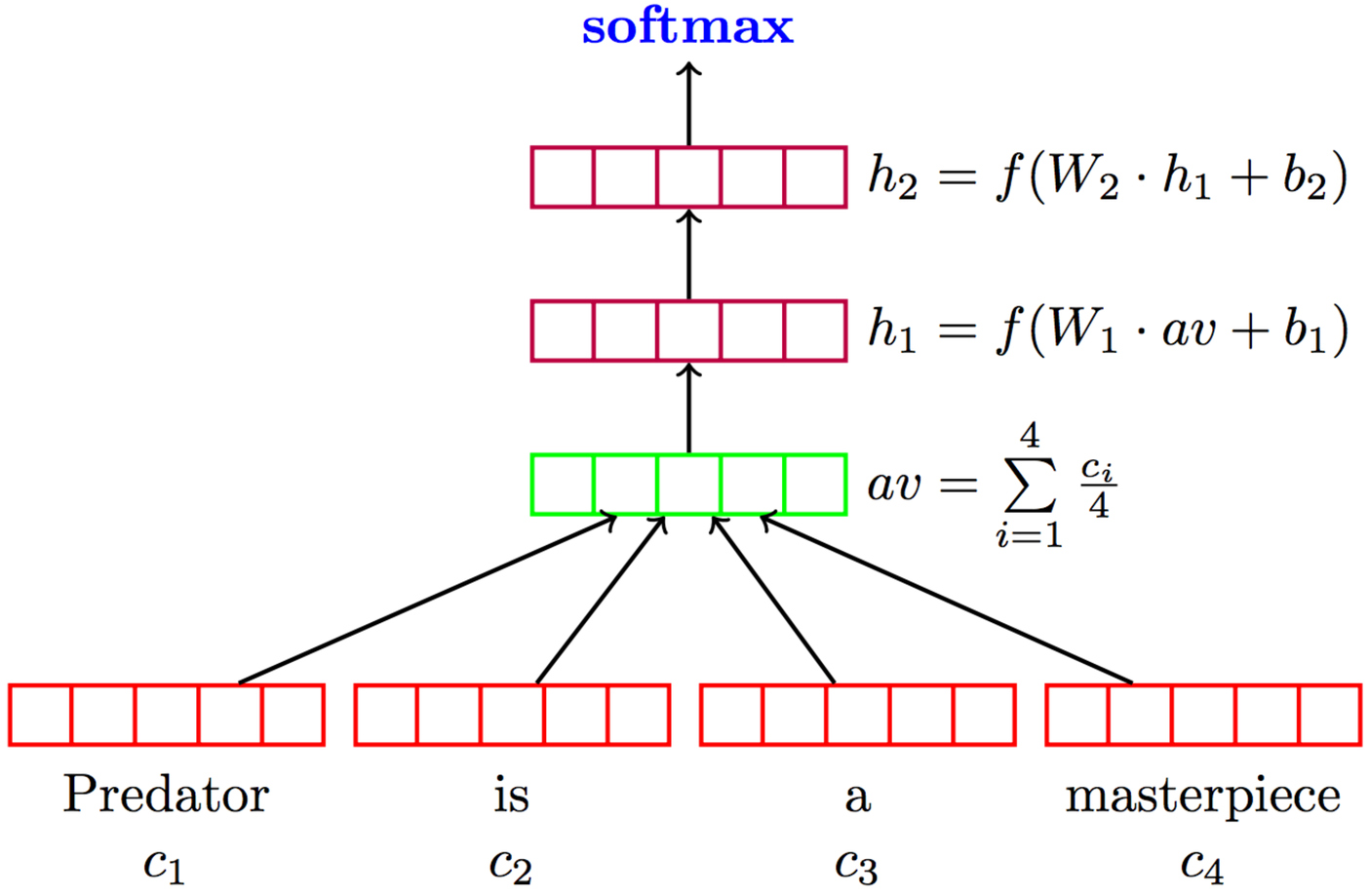
Model	Acc.	Wts.	MB	Ops.
Gillick et al. (2016)	95.06	900k	-	6.63m
Small FF	94.76	241k	0.6	0.27m
+Clusters	95.56	261k	1.0	0.31m
$\frac{1}{2}$ Dim.	95.39	143k	0.7	0.18m

- Gillick used LSTMs; this is smaller, faster, and better

Botha et al. (2017)

Sentiment Analysis

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



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	—

Iyyer et al. (2015)

Bag-of-words

Wang and Manning (2012)

Tree RNNs / CNNS / LSTMS

Kim (2014)

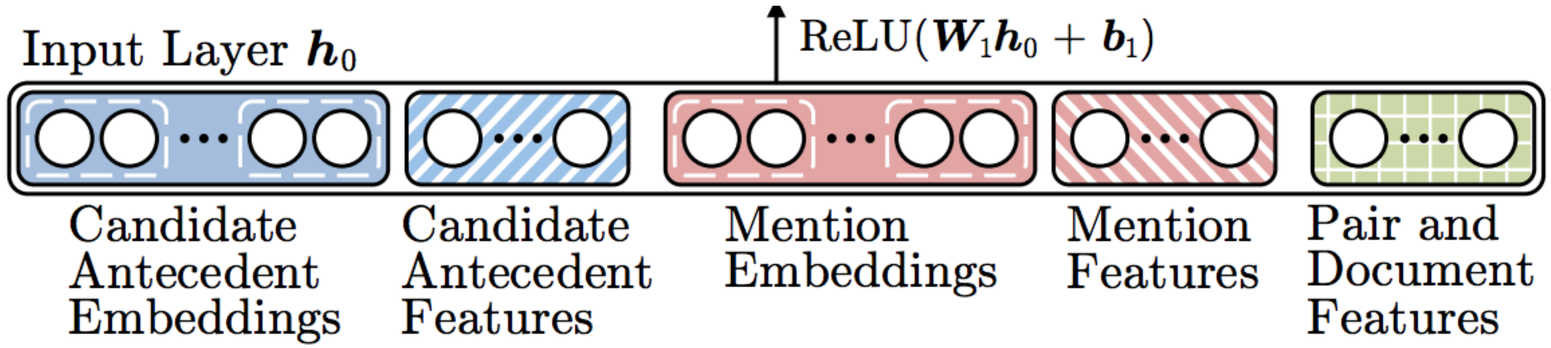
Coreference Resolution

- Feedforward networks identify coreference arcs

President Obama signed…

?

He later gave a speech…



- Mention features include: type of mention (pronoun, nominal, proper), the mention's position in the article, length of the mention in words …

Clark and Manning (2015, 2016), Wiseman et al. (2015)

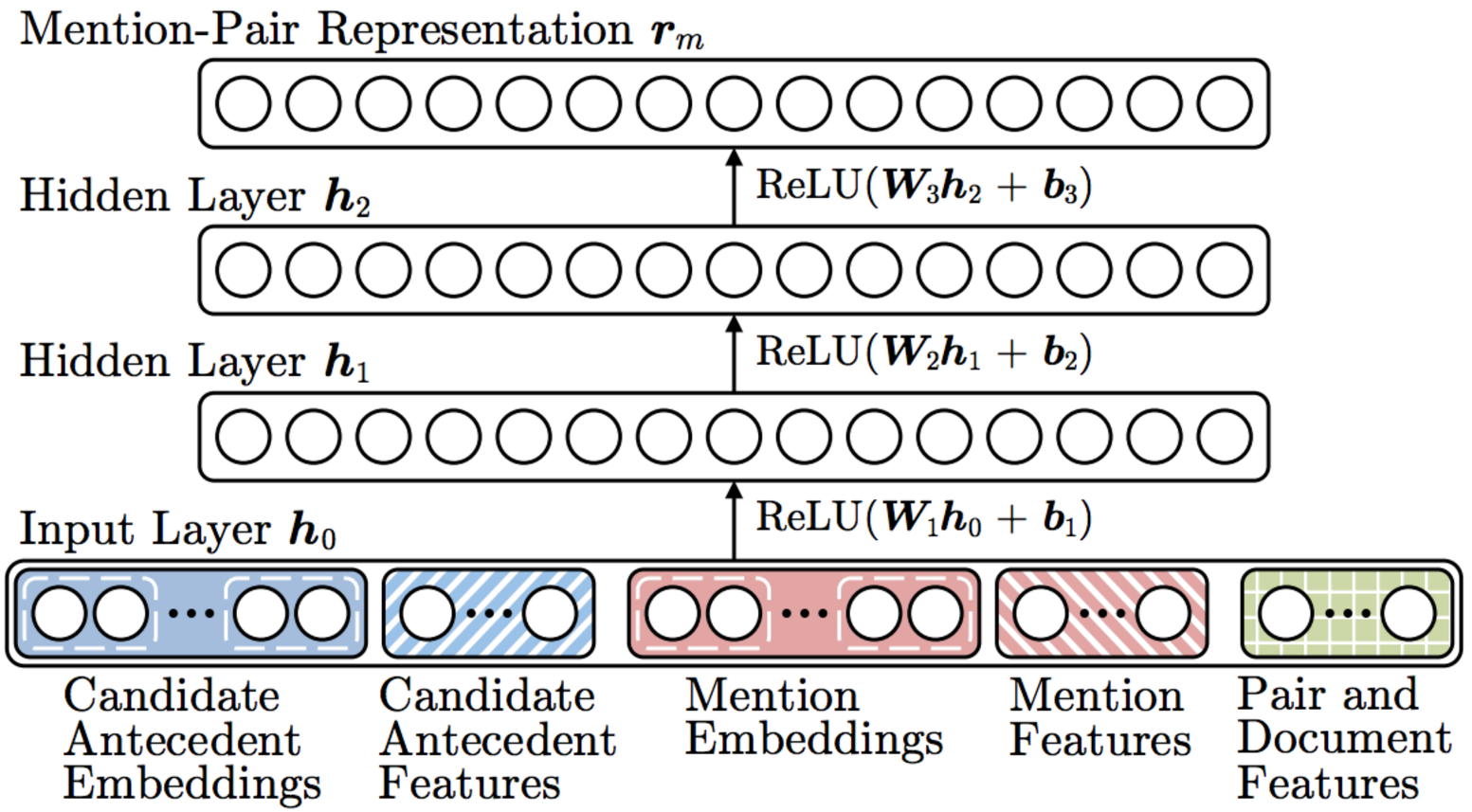
Coreference Resolution

- Feedforward networks identify coreference arcs

President Obama *signed…*

?

He later gave a speech…



Clark and Manning (2015, 2016), Wiseman et al. (2015)

Coreference Resolution

Input Layer. For each mention, the model extracts various words and groups of words that are fed into the neural network. Each word is represented by a vector $\boldsymbol{w}_i \in \mathbb{R}^{d_w}$. Each group of words is represented by the average of the vectors of each word in the group. For each mention and pair of mentions, a small number of binary features and distance features are also extracted. Distances and mention lengths are binned into one of the buckets $[0, 1, 2, 3, 4, 5\text{-}7, 8\text{-}15, 16\text{-}31, 32\text{-}63, 64+]$ and then encoded in a one-hot vector in addition to being included as continuous features. The full set of features is as follows:

Embedding Features: Word embeddings of the head word, dependency parent, first word, last word, two preceding words, and two following words of the mention. Averaged word embeddings of the five preceding words, five following words, all words in the mention, all words in the mention's sentence, and all words in the mention's document.

Additional Mention Features: The type of the mention (pronoun, nominal, proper, or list), the mention's position (index of the mention divided by the number of mentions in the document), whether the mentions is contained in another mention, and the length of the mention in words.

Document Genre: The genre of the mention's document (broadcast news, newswire, web data, etc.).

Distance Features: The distance between the mentions in sentences, the distance between the mentions in intervening mentions, and whether the mentions overlap.

Speaker Features: Whether the mentions have the same speaker and whether one mention is the other mention's speaker as determined by string matching rules from Raghunathan et al. (2010).

String Matching Features: Head match, exact string match, and partial string match.

The vectors for all of these features are concatenated to produce an I-dimensional vector $\boldsymbol{h}_0$, the input to the neural network. If $a = \text{NA}$, the fea-

Clark and Manning (2015, 2016), Wiseman et al. (2015)

Training Tips

Training Basics

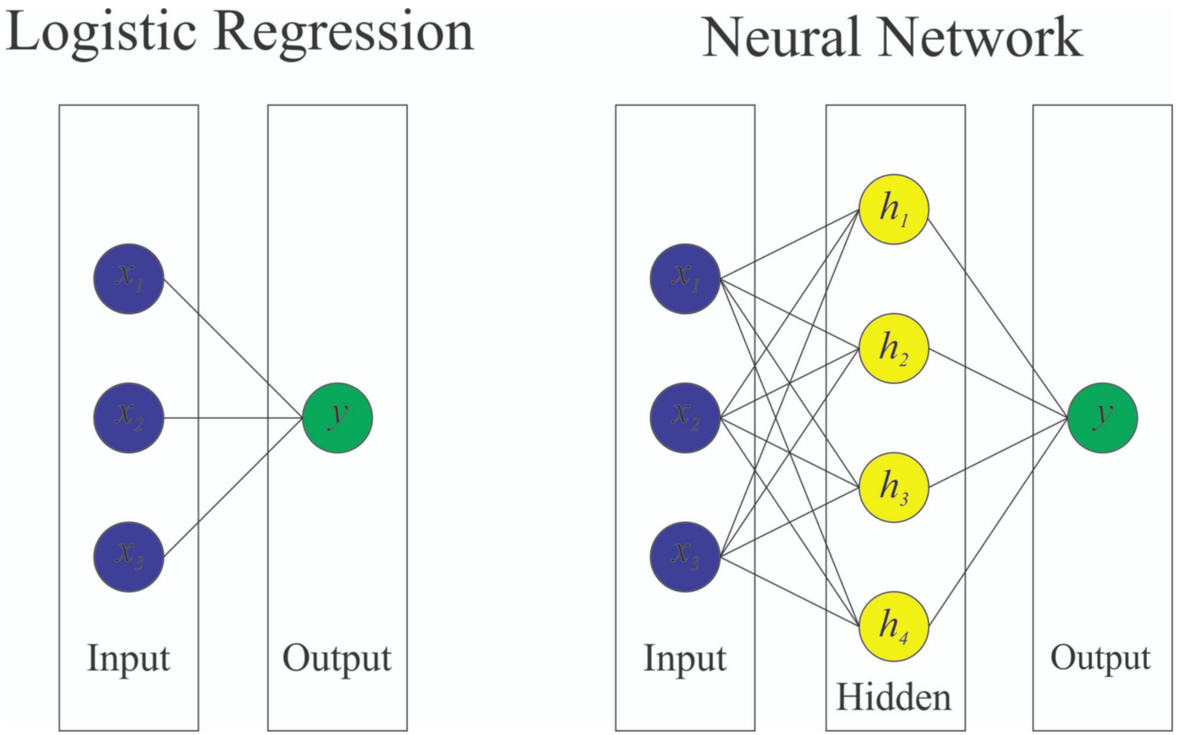
- Basic formula: compute gradients on batch, use first-order optimization method (SGD, Adagrad, etc.)
- How to initialize? How to regularize? What optimizer to use?
- This lecture: some practical tricks. Take deep learning or optimization courses to understand this further

Trivia Time

Q: how should parameters be initialized in a logistic regression model?

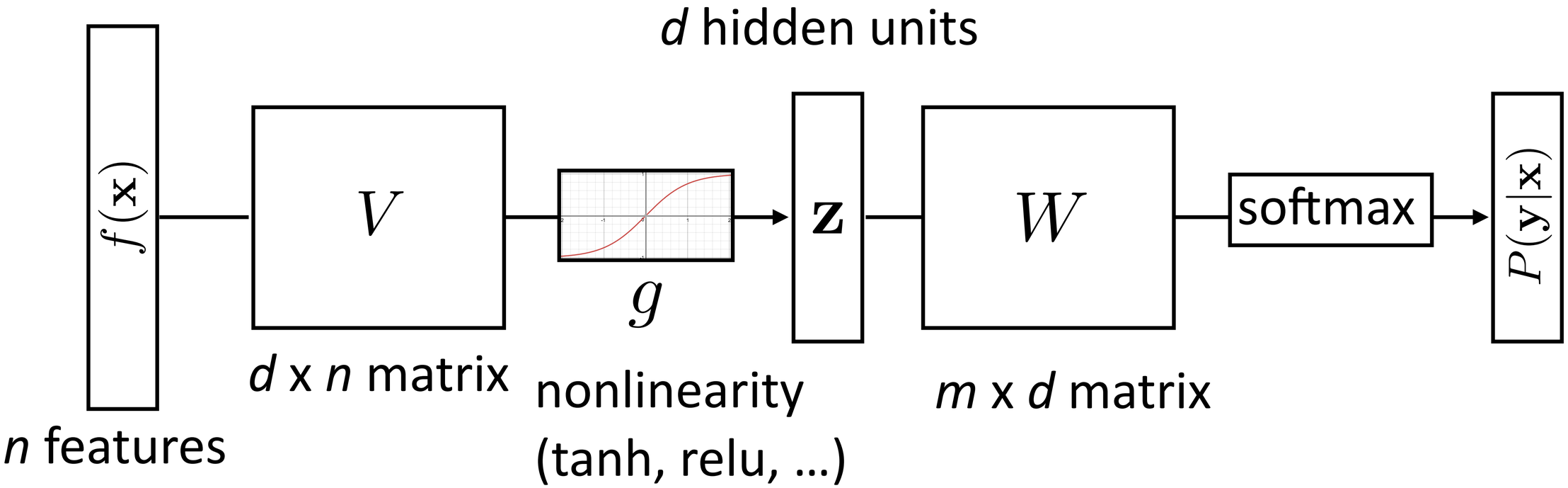
Trivia Time

Q: how should parameters be initialized in a neural network model?



How does initialization affect learning?

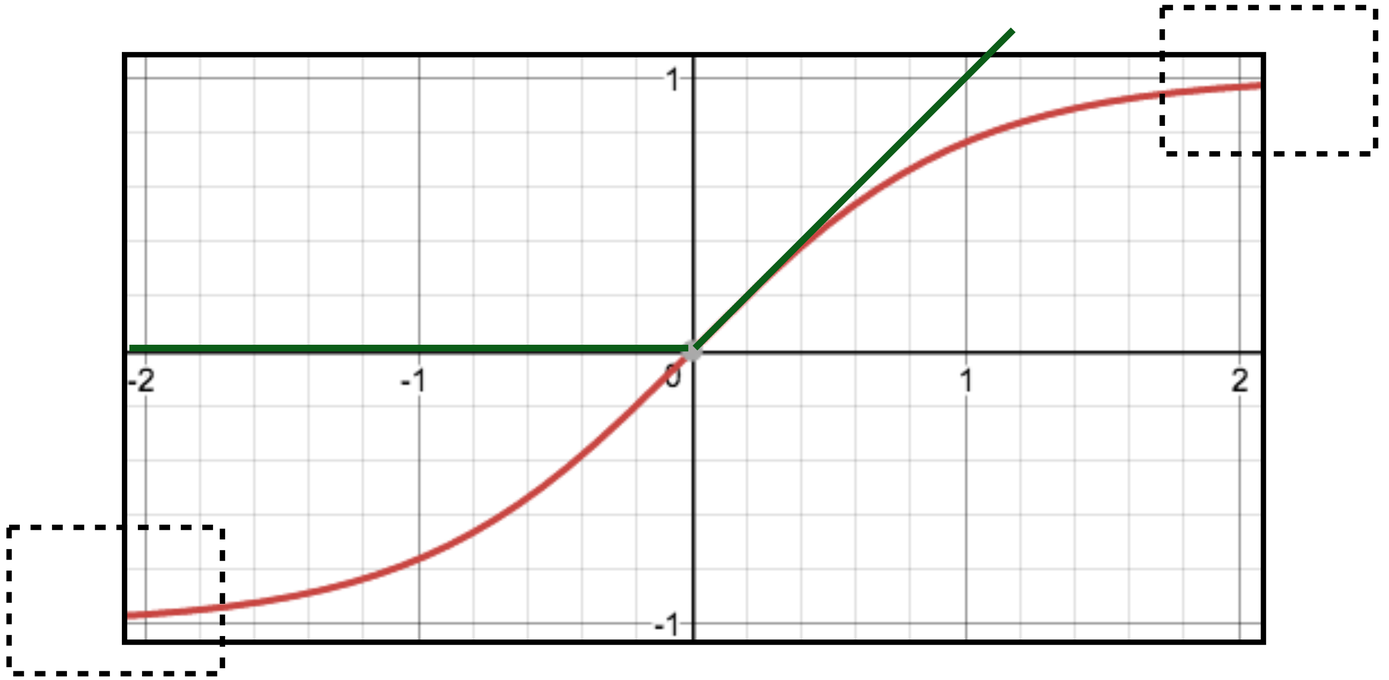
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W g(V f(\mathbf{x})))$$



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

How does initialization affect learning?

- Nonlinear model…how does this affect things?



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

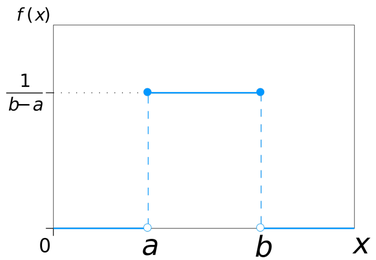
Krizhevsky et al. (2012)

Initialization

1) Can't use zeroes for parameters to produce hidden layers: all values in that hidden layer are always the same (0 if tanh) and have same gradients (0 if tanh), and can't break symmetry (or never change)

2) Initialize too large and cells are saturated

- Can do random uniform / normal initialization with appropriate scale
- Xavier 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



Mean & Standard Deviation

$$\mu = \frac{a+b}{2} \text{ and } \sigma = \frac{b-a}{\sqrt{12}}$$

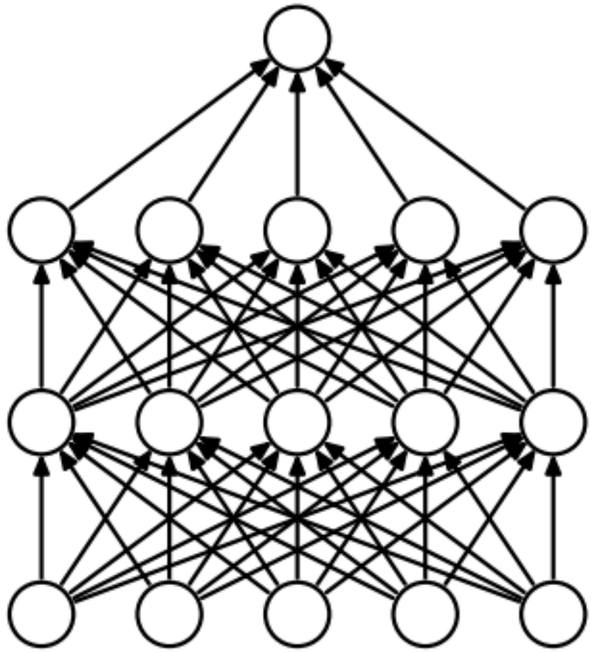
Glorot & Bengio (2010)

https://mmuratarat.github.io/2019-02-25/xavier-glorot-he-weight-init

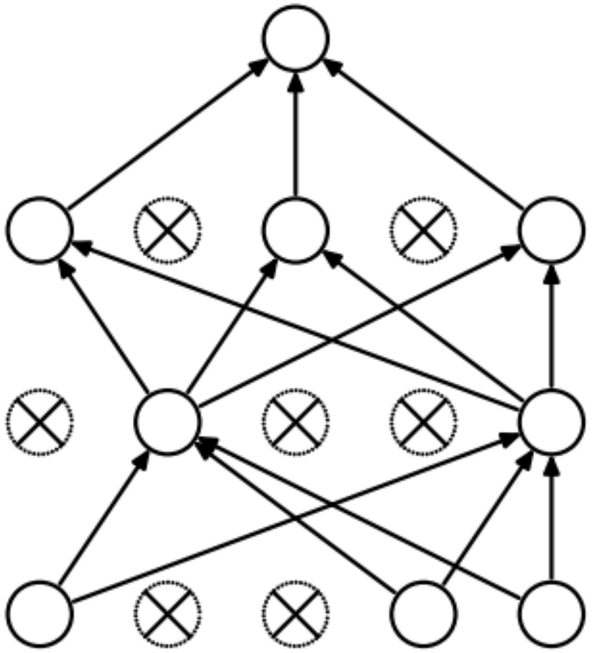
Maverick Meerkat's answer - https://stats.stackexchange.com/questions/27112/danger-of-setting-all-initial-weights-to-zero-in-backpropagation

Regularization: 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)

Optimization

- Gradient descent
  - **Batch update** for logistic regression
  - Each update is based on a computation over the entire dataset

Multiclass Logistic Regression

$$P_w(y|x) = \frac{\exp\left(w^\top f(x,y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(w^\top f(x,y')\right)}$$

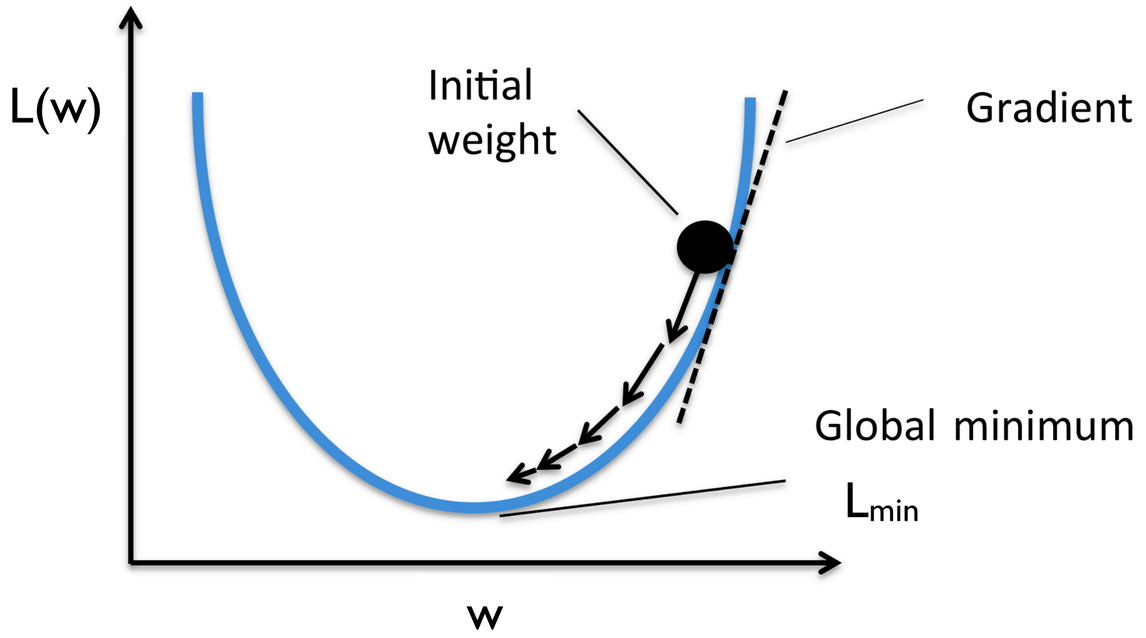
sum over output space to normalize

i.e. minimize negative log likelihood or cross-entropy loss

- Training: maximize $\mathcal{L}(x,y) = \sum_{j=1}^{m} \log P(y_j^*|x_j)$

index of data points (*j*)

$$= \sum_{j=1}^{m} \left(w^\top f(x_j, y_j^*) - \log \sum_y \exp(w^\top f(x_j, y)) \right)$$

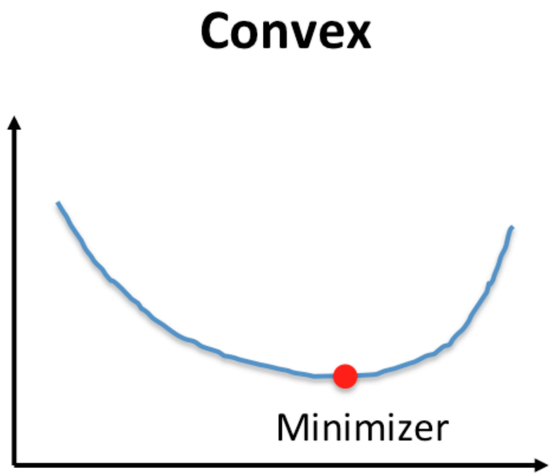


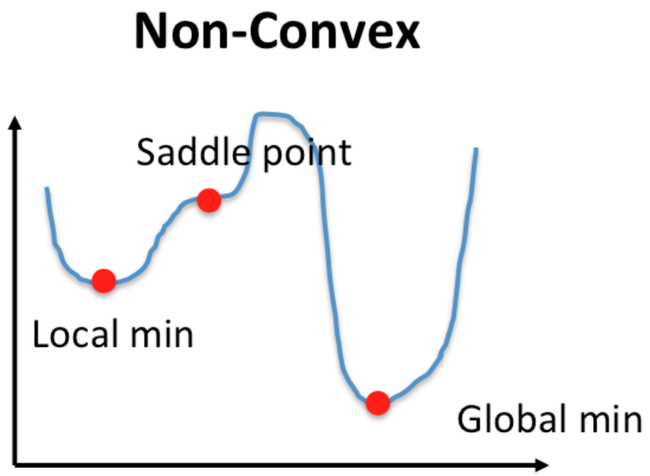
Optimization

- **Stochastic** gradient descent

$$w \leftarrow w - \alpha g, \quad g = \frac{\partial}{\partial w}\mathcal{L}$$

  - Approx. gradient is computed on a single instance
  - What if the loss function has a local minima or saddle point?





Dauphin et al. (2014)

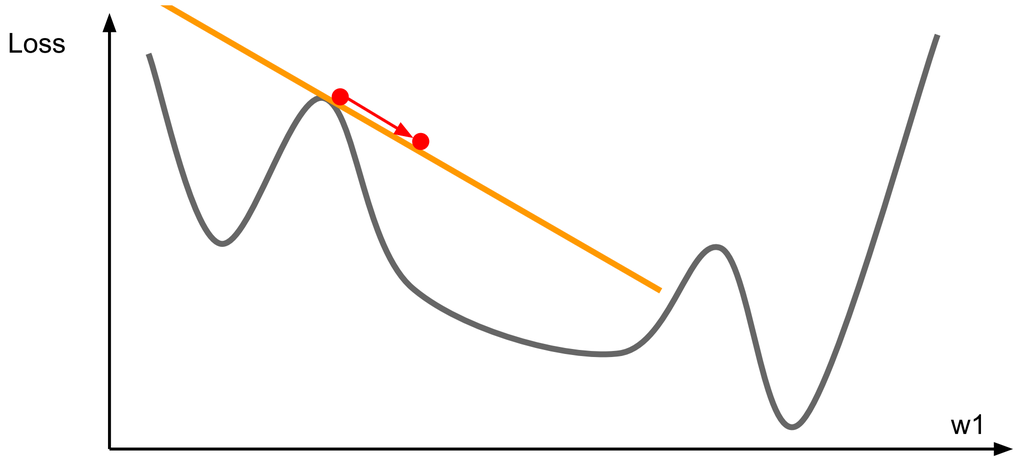
Image credit: Paweł Cislo

Optimization

- **Stochastic** gradient descent

$$w \leftarrow w - \alpha g, \quad g = \frac{\partial}{\partial w}\mathcal{L}$$

  - Approx. gradient is computed on a single instance
  - “First-order” technique: only relies on having gradient



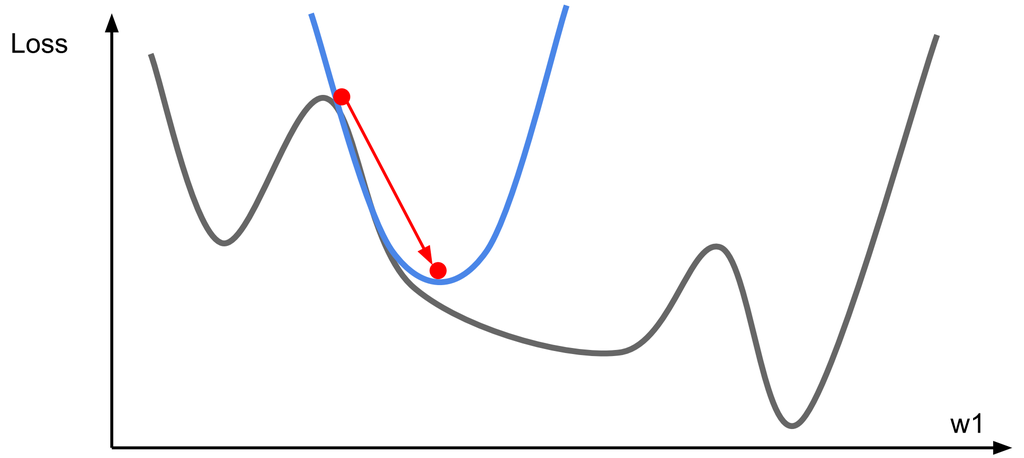


Image credit: Stanford CS231N

Momentum

- Gradients come from a single instance or a mini-batch can be noisy
- Use “velocity” to accumulates the gradients from the past steps

Standard SGD

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD with Momentum

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

Polyak (1964), Sutskever et al. (2013)

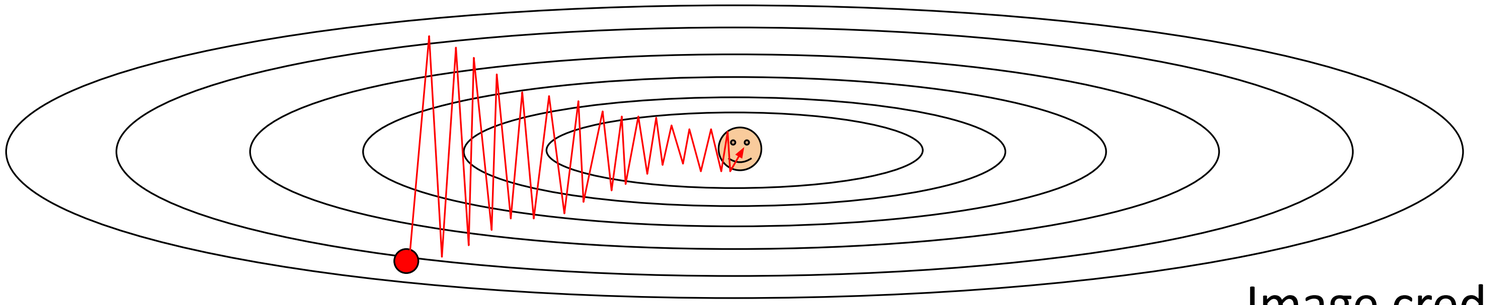
Image credit: Stanford CS231N

AdaGrad (Adaptive Gradient)

- Optimized for problems with sparse features

- Per-parameter learning rate: smaller updates are made to parameters that get updated frequently

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Duchi et al. (2011)

Image credit: Stanford CS231N

AdaGrad (Adaptive Gradient)

- Optimized for problems with sparse features
- Per-parameter learning rate: smaller updates are made to parameters that get updated frequently

$$w_i \leftarrow w_i + \alpha \frac{1}{\sqrt{\epsilon + \sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t_i}$$

(smoothed) sum of squared gradients from all updates

- Generally more robust than SGD, requires less tuning of learning rate

https://www.ruder.io/optimizing-gradient-descent/ Duchi et al. (2011)

Adam (Adaptive Moment Estimation)

- Another method that computes adaptive learning rates for each parameter

- Keep both:
  - an exponentially decaying average of past squared gradients (v)
  - an exponentially decaying average of past gradients (m)

Algorithm 1: The Basic Adam Optimizer

Require:
1: Initialized parameter θ_0, step size η, batch size N_B
2: Exponential decay rates $\beta_1, \beta_2, \varepsilon$ dataset $\{(x_i, y_i)\}_{i=1}^{N}$
Initialize: $m_0 = 0$, $v_0 = 0$
3: **For** all $t = 1, \ldots \ldots, T$ ***do***
4: Draw random batch $\{(x_{ik}, y_{ik})\}_{k=1}^{NB}$ from dataset
5: $g_t \leftarrow \sum_{k=1}^{N} \nabla\lfloor \{(x_{ik}, y_{ik}, \theta_{t-1})\}$ // $f'(\theta_{t-1})$
6: $m_t \leftarrow \beta_1 . m_{t-1} + (1-\beta_1). g_t$ // moving Average
7: $v_t \leftarrow \beta_2 . m_{t-1} + (1-\beta_2). g_t^2$
8: $\hat{m_t} \leftarrow \frac{m_t}{1-\beta_1^t}$, $\hat{v_t} \leftarrow \frac{v_t}{1-\beta_2^t}$ // correction bias
9: $\theta_t \leftarrow \theta_{t-1} - \eta . \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \varepsilon}$
10: **end for**
11: **return** final parameter θ_T

Kingma & Ba (2015)

Adam (Adaptive Moment Estimation)

Published as a conference paper at ICLR 2015

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

ABSTRACT

We introduce *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which *Adam* was inspired, are discussed. We also analyze the theoretical convergence properties of the algorithm and provide a regret bound on the convergence rate that is comparable to the best known results under the online convex optimization framework. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss *AdaMax*, a variant of *Adam* based on the infinity norm.

1 INTRODUCTION

Stochastic gradient-based optimization is of core practical importance in many fields of science and engineering. Many problems in these fields can be cast as the optimization of some scalar parameterized objective function requiring maximization or minimization with respect to its parameters. If the function is differentiable w.r.t. its parameters, gradient descent is a relatively efficient optimization method, since the computation of first-order partial derivatives w.r.t. all the parameters is of the same computational complexity as just evaluating the function. Often, objective functions are stochastic. For example, many objective functions are composed of a sum of subfunctions evaluated at different subsamples of data; in this case optimization can be made more efficient by taking gradient steps w.r.t. individual subfunctions, i.e. stochastic gradient descent (SGD) or ascent. SGD proved itself as an efficient and effective optimization method that was central in many machine learning success stories, such as recent advances in deep learning (Deng et al., 2013; Krizhevsky et al., 2012; Hinton & Salakhutdinov, 2006; Hinton et al., 2012a; Graves et al., 2013). Objectives may also have other sources of noise than data subsampling, such as dropout (Hinton et al., 2012b) regularization. For all such noisy objectives, efficient stochastic optimization techniques are required. The focus of this paper is on the optimization of stochastic objectives with high-dimensional parameters spaces. In these cases, higher-order optimization methods are ill-suited, and discussion in this paper will be restricted to first-order methods.

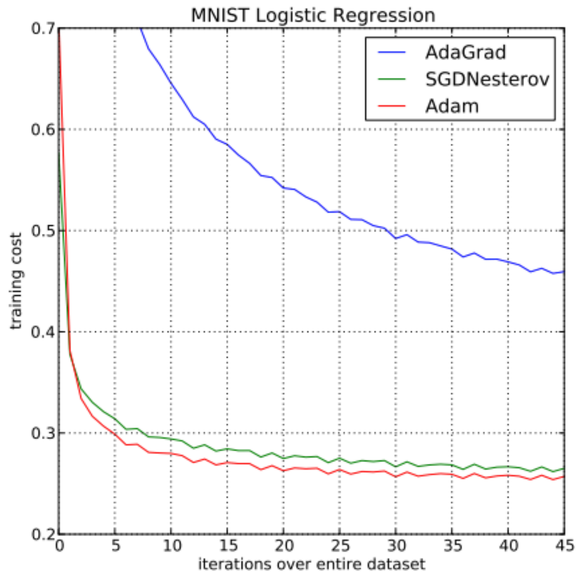
We propose *Adam*, a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name *Adam* is derived from adaptive moment estimation. Our method is designed to combine the advantages of two recently popular methods: AdaGrad (Duchi et al., 2011), which works well with sparse gradients, and RMSProp (Tieleman & Hinton, 2012), which works well in on-line and non-stationary settings; important connections to these and other stochastic optimization methods are clarified in section 5. Some of Adam's advantages are that the magnitudes of parameter updates are invariant to rescaling of the gradient, its stepsizes are approximately bounded by the stepsize hyperparameter, it does not require a stationary objective, it works with sparse gradients, and it naturally performs a form of step size annealing.

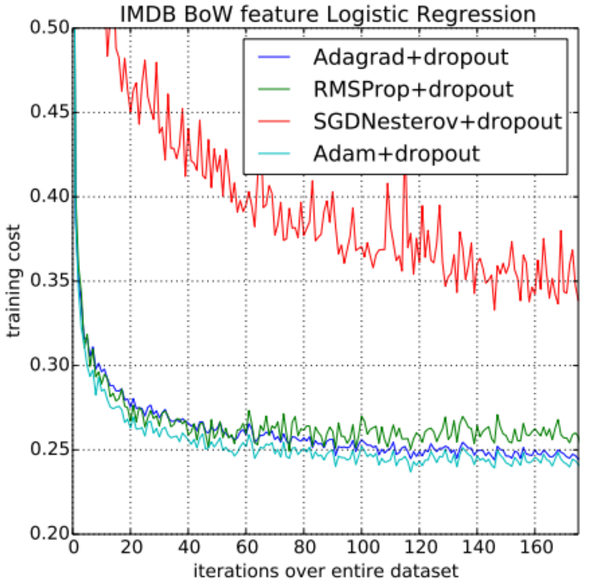
*Equal contribution. Author ordering determined by coin flip over a Google Hangout.

Kingma & Ba (2015)

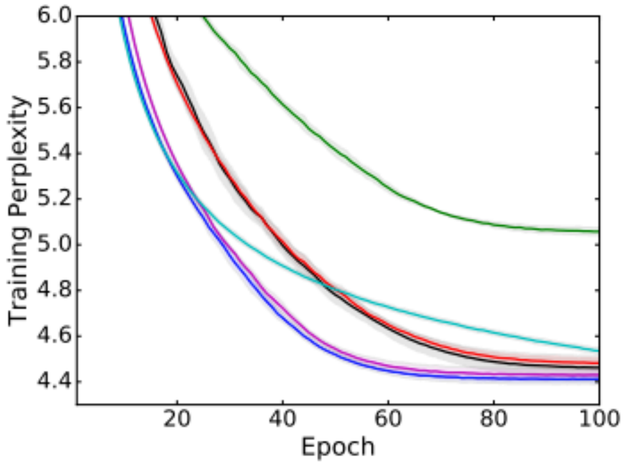
Optimizer

- Adam (Kingma and Ba, ICLR 2015): very widely used. Adaptive step size + momentum

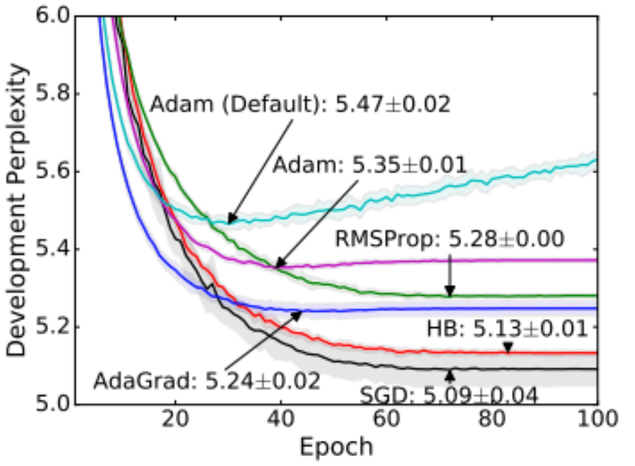




- Wilson et al. NIPS 2017: adaptive methods can actually perform badly at test time (Adam is in pink, SGD in black)



(e) Generative Parsing (Training Set)

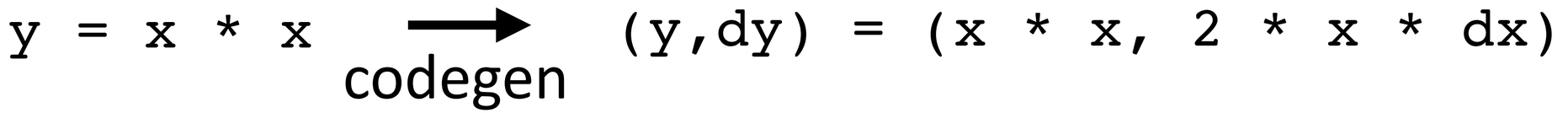


(f) Generative Parsing (Development Set)

- One more trick: **gradient clipping** (set a max value for your gradients)

Computation Graphs

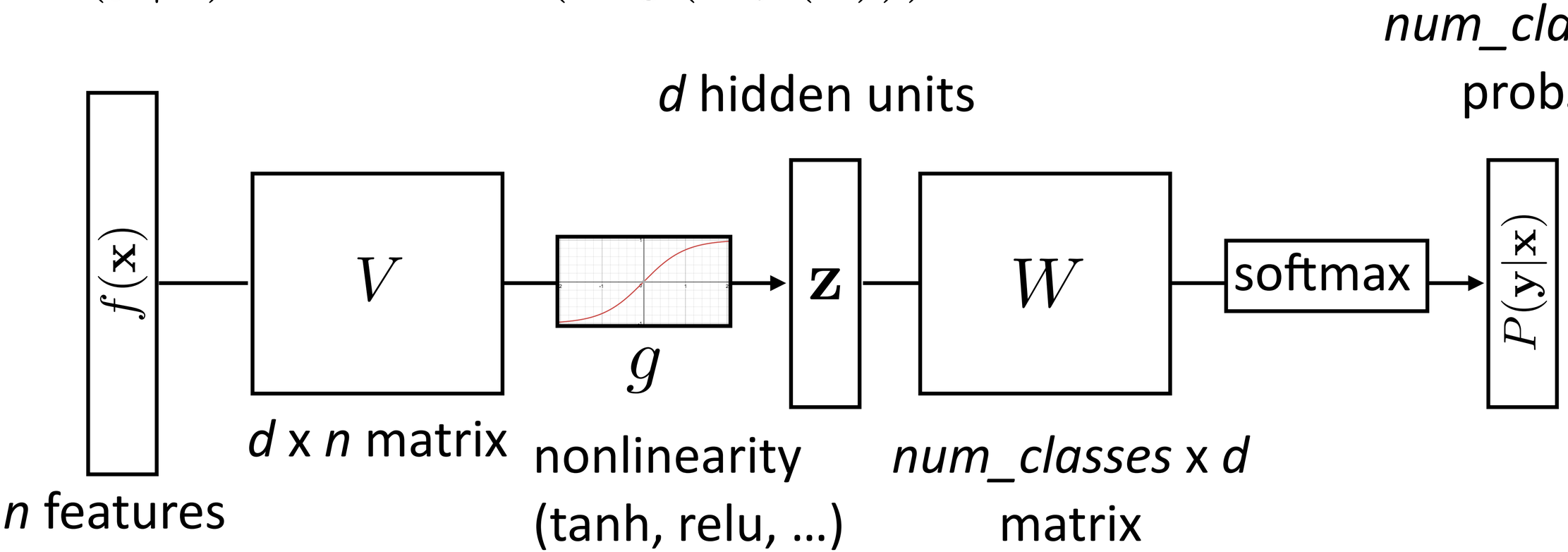
- Computing gradients is hard!
- Automatic differentiation: instrument code to keep track of derivatives



- Computation is now something we need to reason about symbolically
- Use a library like PyTorch or TensorFlow. This class: PyTorch

Neural Networks for Classification

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



We can think of a neural network classifier with one hidden layer as building a vector z which is a hidden layer representation (i.e. latent features) of the input, and then running standard logistic regression on the features that the network develops in z.

Computation Graphs in Pytorch

- Define forward pass for $P(\mathbf{y}|\mathbf{x}) = \text{softmax}(W\,g(V\,f(\mathbf{x})))$

```
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))))
```

Training Neural Networks

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

- 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

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

Computation Graphs in Pytorch

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

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

```
ffnn = FFNN(in_d, hi_d, out_d)
optimizer = optim.Adam(ffnn.parameters(), lr=0.01)
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()
```

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

Training a Model

Define a computation graph

For each epoch:

For each batch of data:

Compute loss on batch

Autograd to compute gradients and take step

Check performance on dev set periodically to identify overfitting

Batching (aka, mini-batch)

- Batching data gives speedups due to more efficient matrix operations

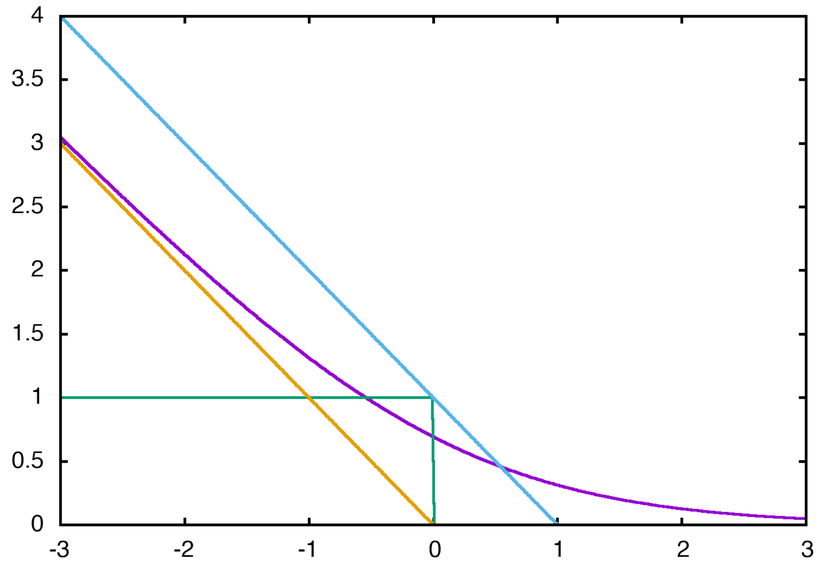
- Need to make the computation graph process a batch at the same time

```
# 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

Four Elements of NNs

- Model: feedforward, RNNs, CNNs can be defined in a uniform framework
- Objective: many loss functions look similar, just changes the last layer of the neural network
- Inference: define the network, your library of choice takes care of it (mostly…)



- Training: lots of choices for optimization/hyperparameters

Next Up

- Word representations
- word2vec/GloVe