Introduction to NLP

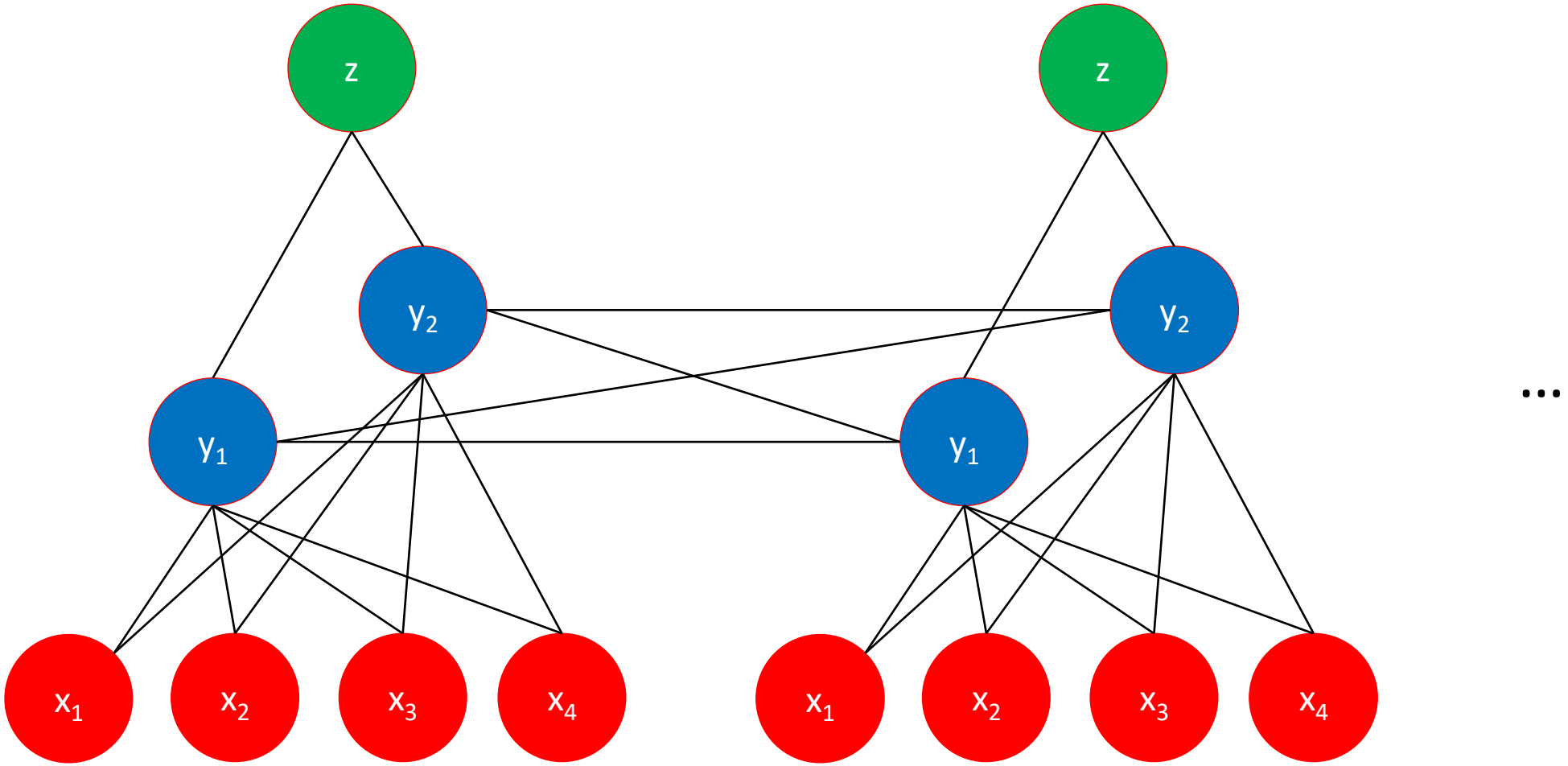
712.

Deep Learning for NLP

Deep Learning for NLP

- Input:
  - words, sentences, documents
- Neural networks expect continuous vectors:
  - word embeddings (next lecture), sentence embeddings, document embeddings
- For sequence of variable length
  - Use recurrent neural nets (RNN), e.g., LSTM

Recurrent Neural Networks (RNN)



Compositional Examples

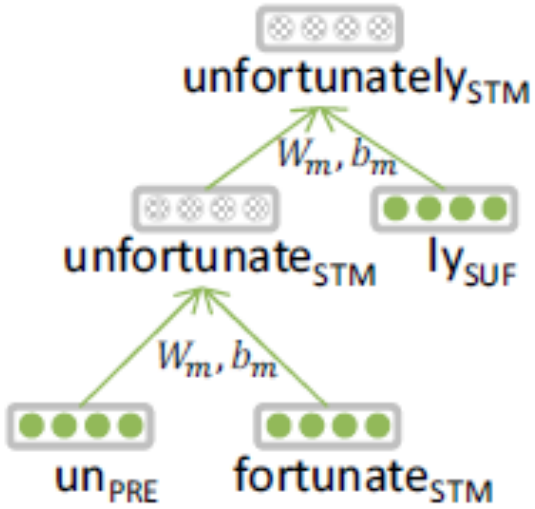


Figure 1: **Morphological Recursive Neural Network.** A vector representation for the word "unfortunately" is constructed from morphemic vectors: $un_{\texttt{pre}}$, $fortunate_{\texttt{stm}}$, $ly_{\texttt{suf}}$. Dotted nodes are computed on-the-fly and not in the lexicon.

[Thang et al. 2013]

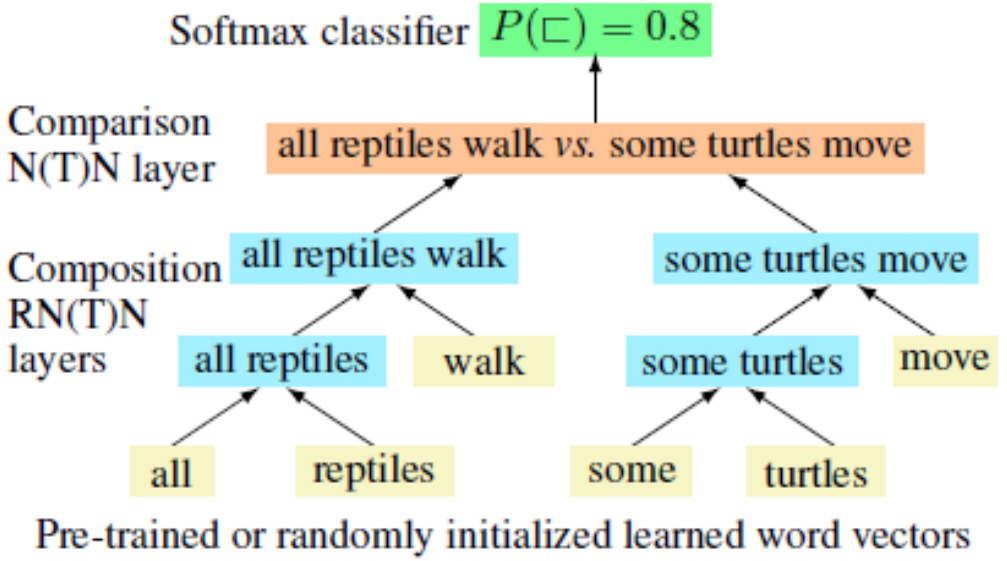


Figure 1: In our model, two separate tree-structured networks build up vector representations for each of two sentences using either NN or NTN layer functions. A comparison layer then uses the resulting vectors to produce features for a classifier.

[Bowman et al. 2014]

Sentiment Analysis

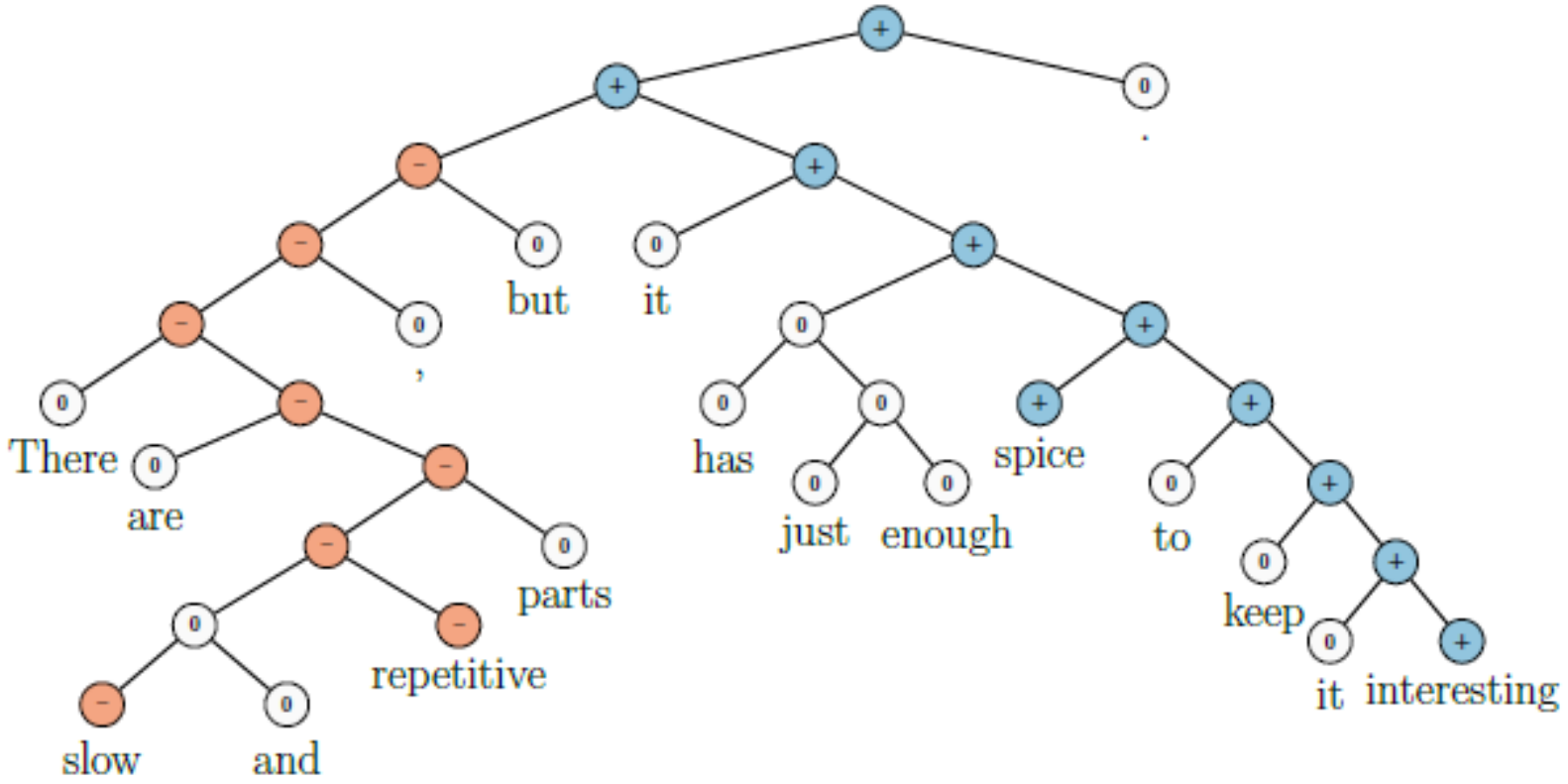


Figure 7: Example of correct prediction for contrastive conjunction *X but Y*.

[Socher et al. 2013]

Deep Averaging Networks for Sentiment Analysis

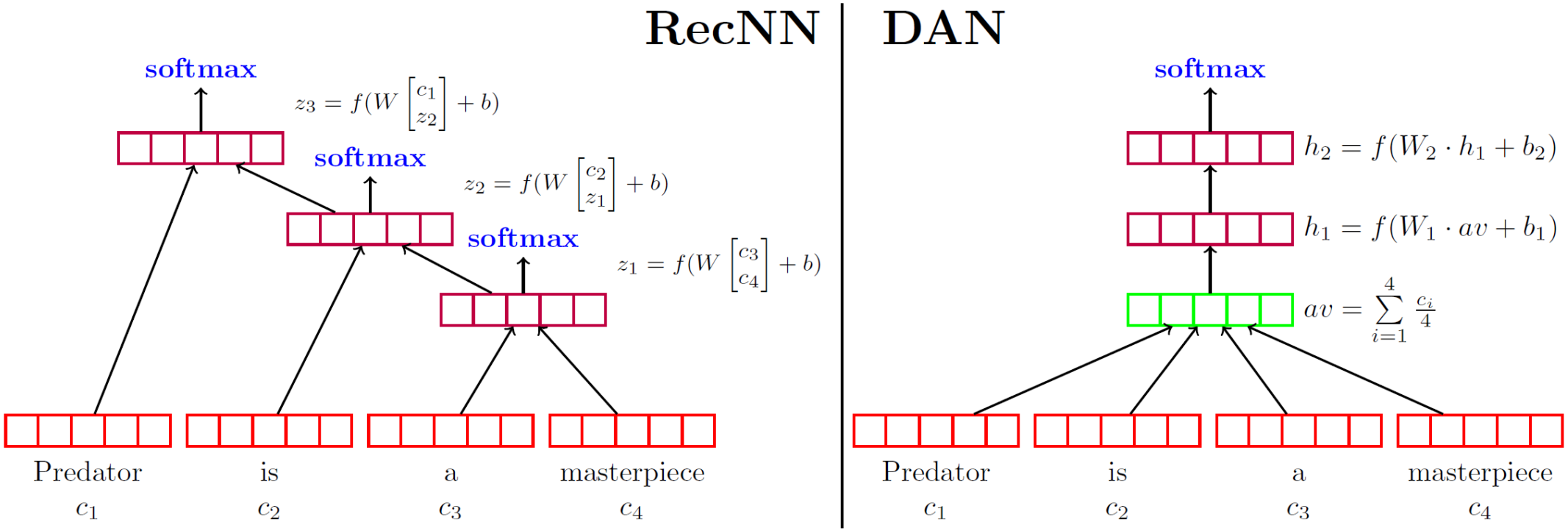


Figure 1: On the left, a **RecNN** is given an input sentence for sentiment classification. Softmax layers are placed above every internal node to avoid vanishing gradient issues. On the right is a two-layer **DAN** taking the same input. While the **RecNN** has to compute a nonlinear representation (purple vectors) for every node in the parse tree of its input, this **DAN** only computes two nonlinear layers for every possible input.

[Iyyer et al. 2015]

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	—

Table 1: **DAN**s achieve comparable sentiment accuracies to syntactic functions (bottom third of table) but require much less training time (measured as time of a single epoch on the SST fine-grained task). Asterisked models are initialized either with different pretrained embeddings or randomly.

Machine Translation

- Sequence-to-sequence

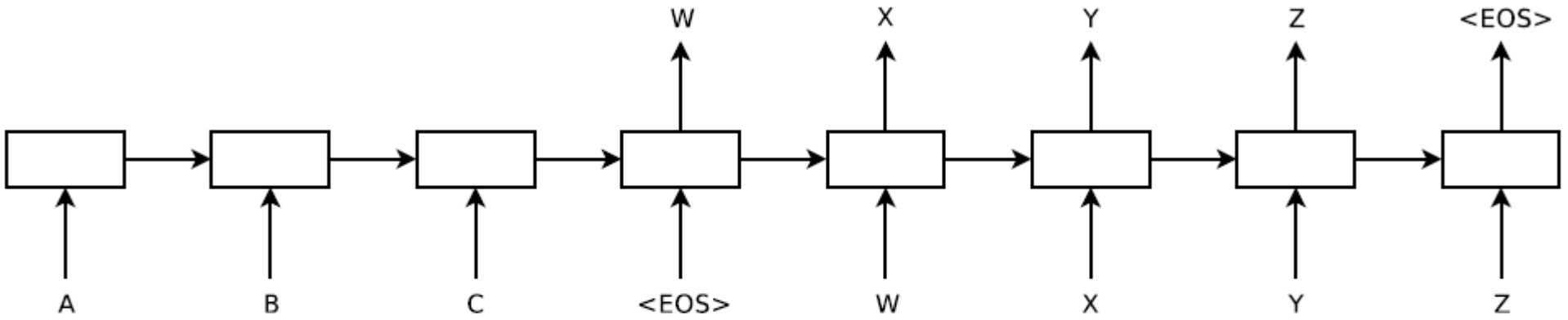


Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

[Sutskever et al. 2014]

Coreference Resolution

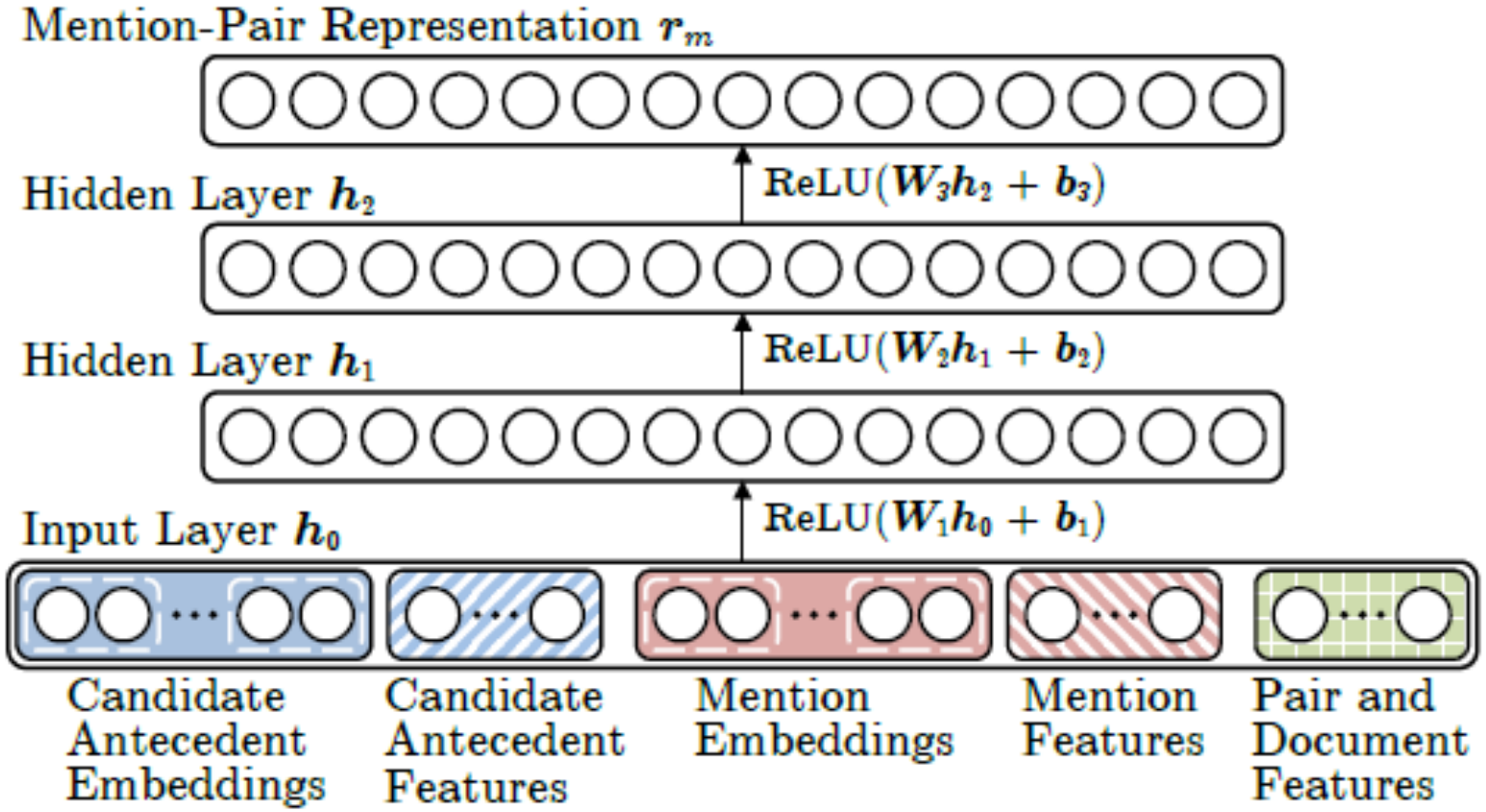
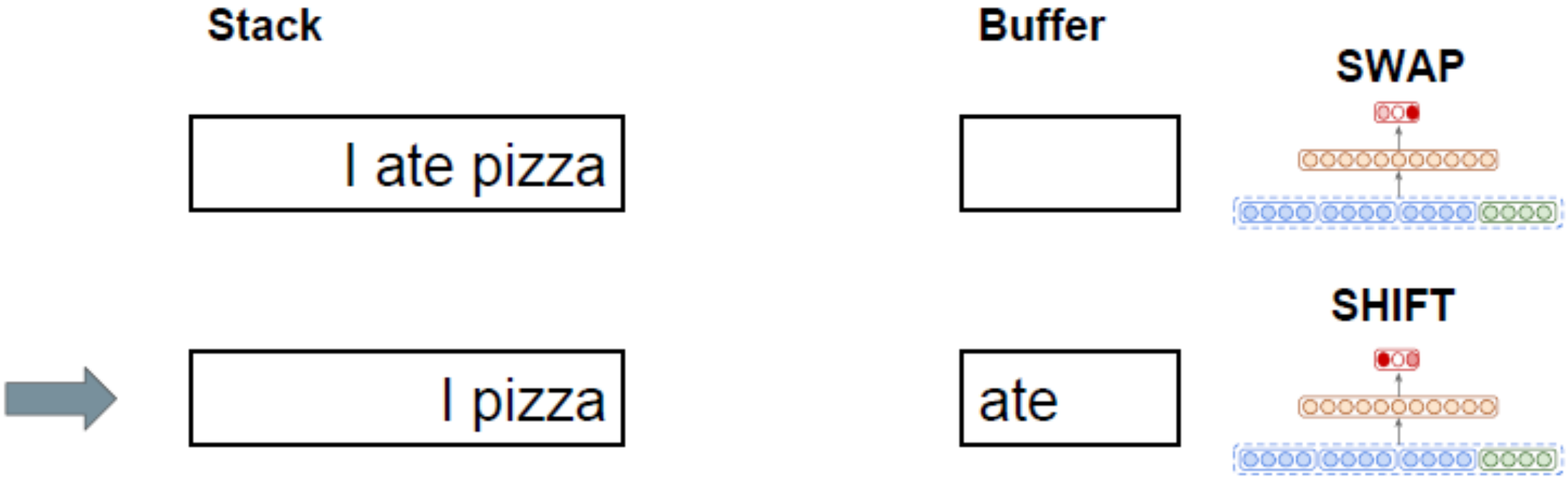


Figure 2: Mention-pair encoder.

[Clark and Manning 2016]

Input Reordering

English → Japanese word ordering: I ate pizza → I pizza ate



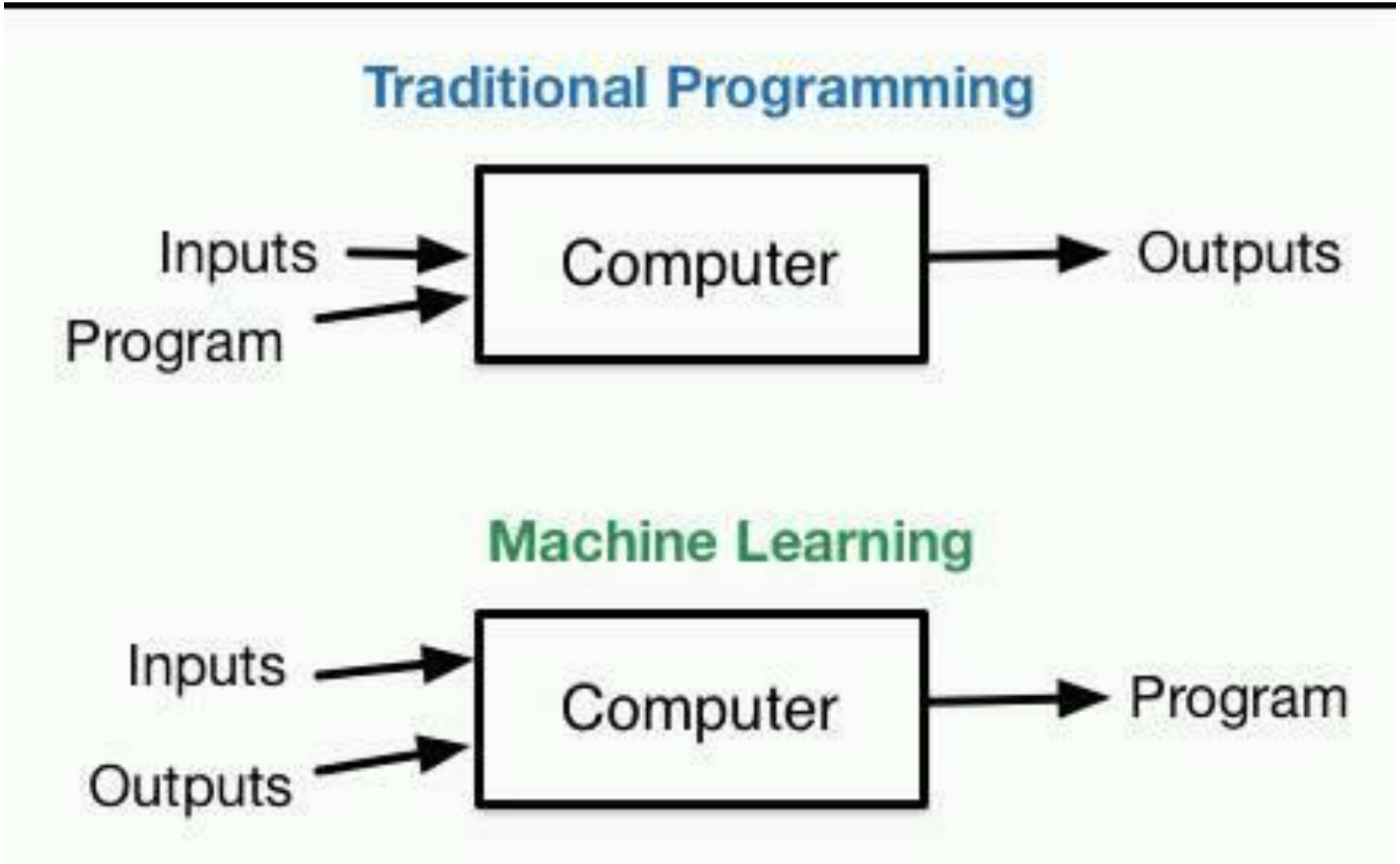
(Botha et al. 2017)

Introduction to NLP

711.

Introduction to Deep Learning

Modern AI Paradigm



Neural Networks

- Remember Logistic Regression?
  - Input: vector of features
  - Output: classification decision
- One-layer LR (maxent model) is a simple Neural Network
  - Neuron (perceptron): output is a non-linear combination of the inputs
  - Non-linear because of the activation function
  - Softmax

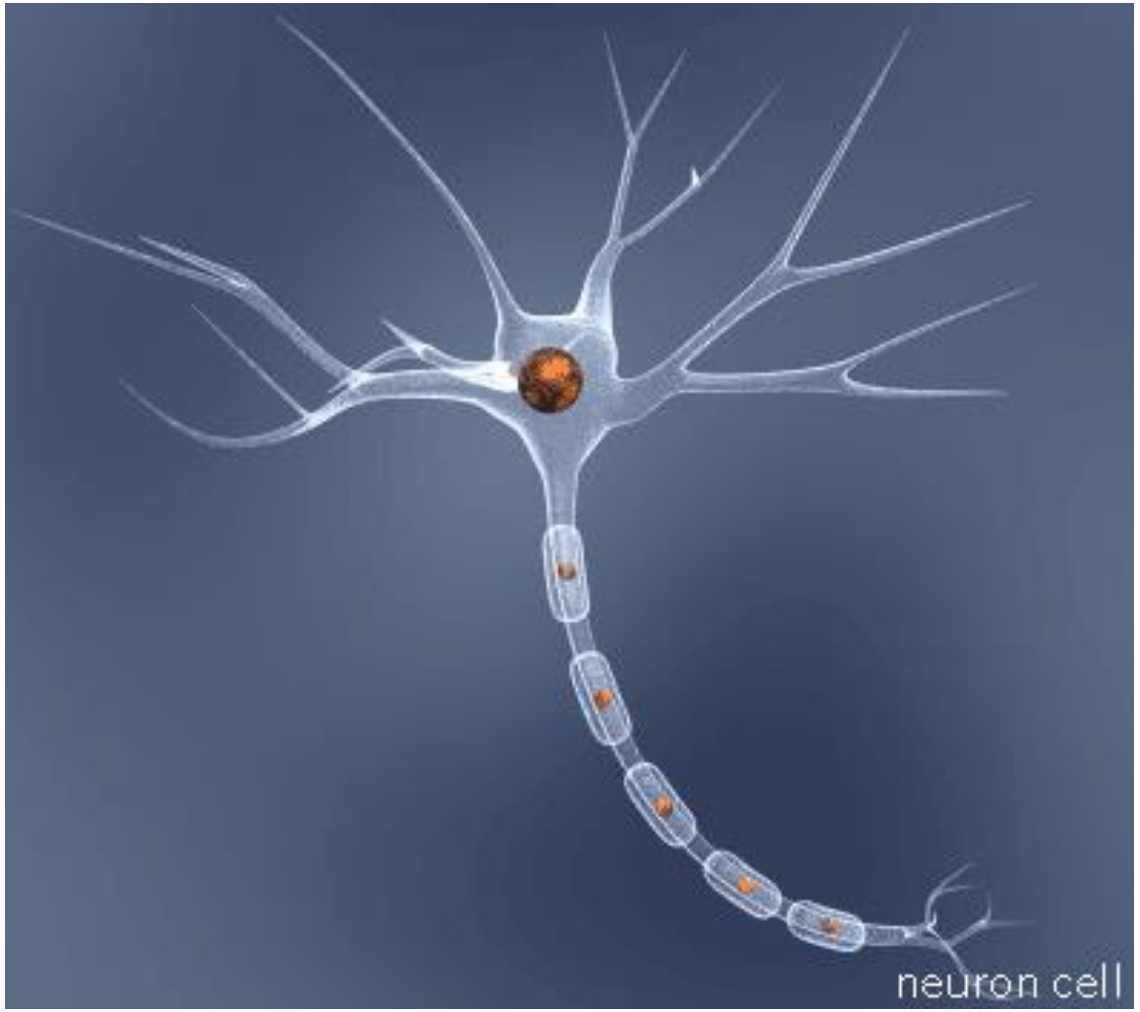
Introduction to NLP

515.

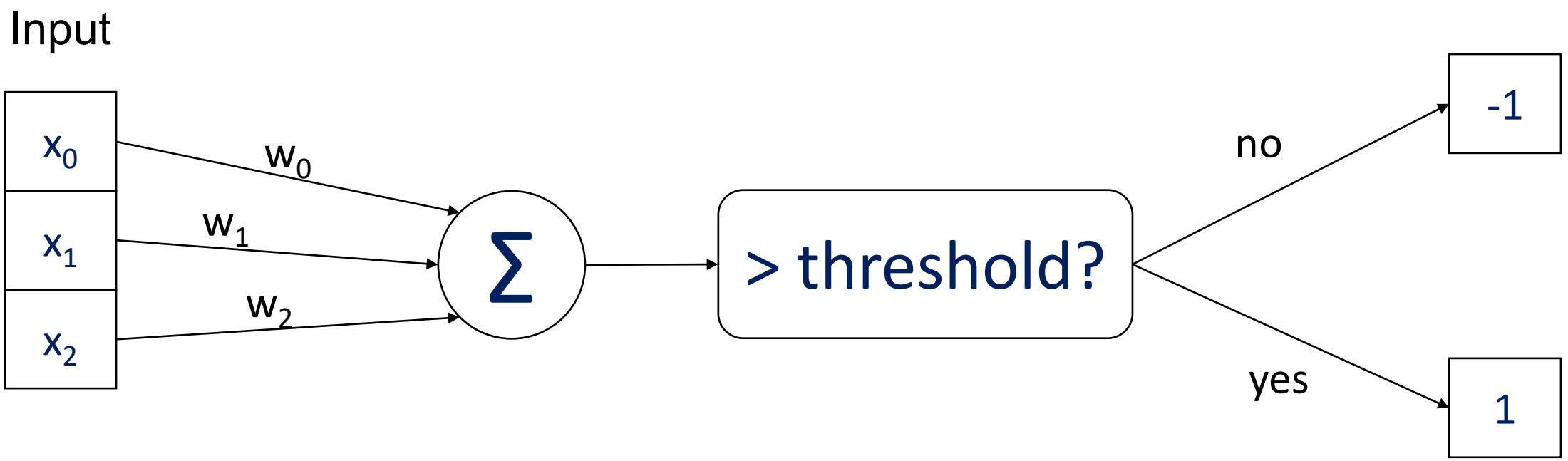
Perceptrons

The Perceptron

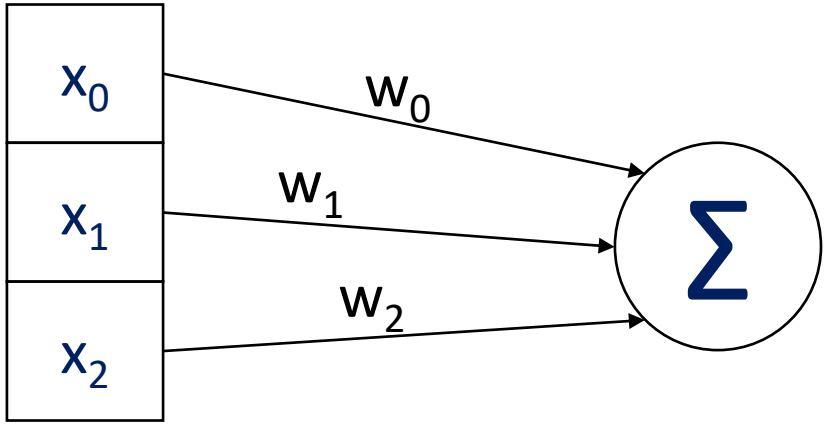
- A simple but very important (discriminative) classifier
- Model of a neuron
  - Input excitations
  - If excitation > inhibition, send an electrical signal out the axon
- Earliest neural network
  - invented in 1957 by Frank Rosenblatt at Cornell



Perceptron Idea



So we can rewrite

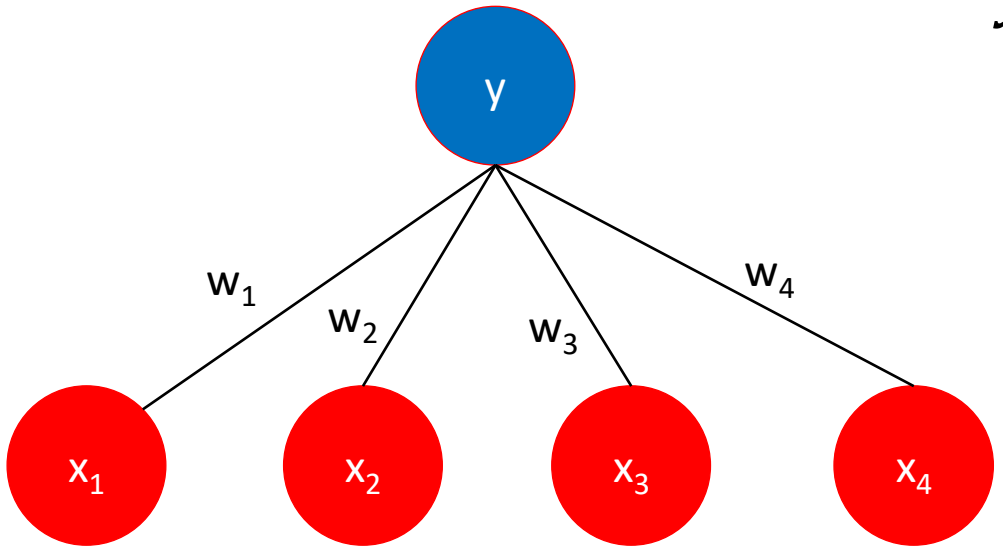


- as a dot product of two vectors

$$\vec{w} \cdot \vec{x}$$

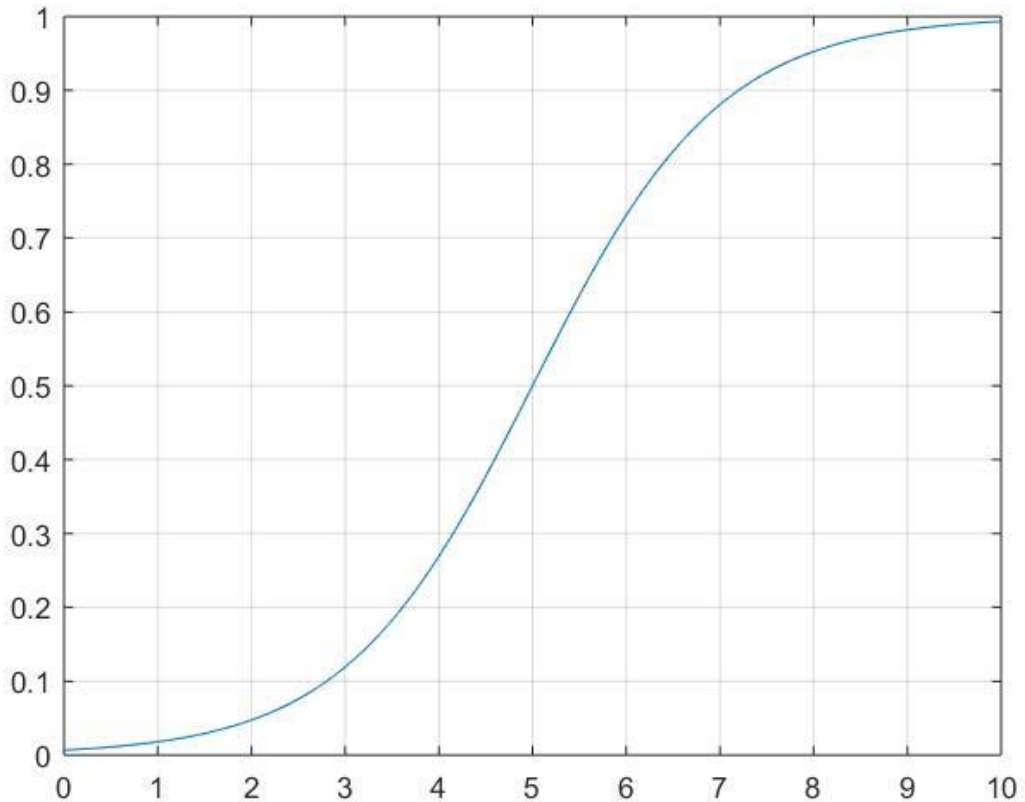
Perceptron

$$\hat{y} = 1 \ \textit{iff} \ \sum x_i w_i \geq b$$

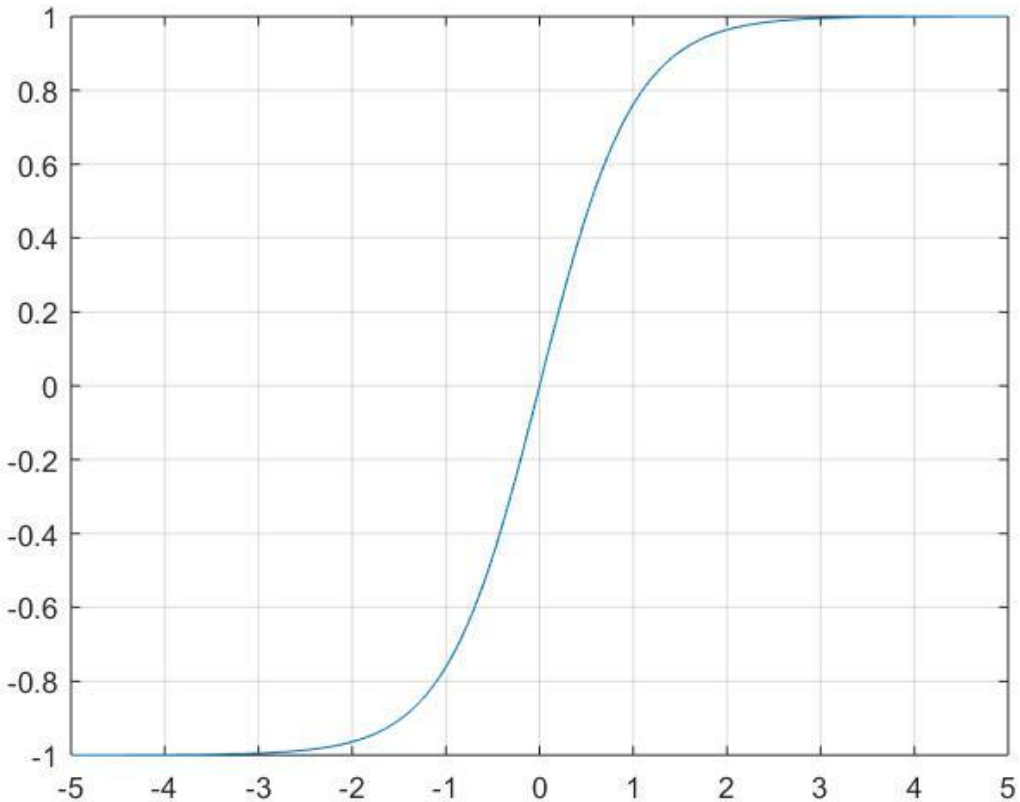


Non-linearities: sigmoid, tanh

$$\sigma(z) = 1/(1 + \exp(-z))$$

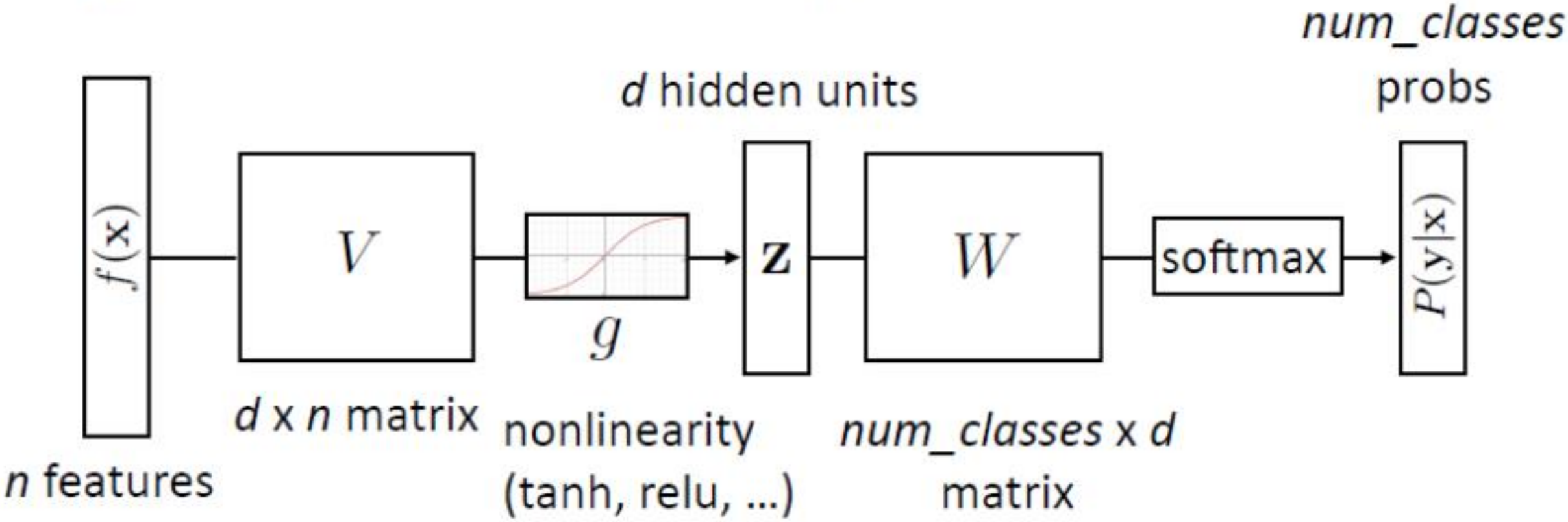


$$tanh(z) = (e^z - e^{-z})/(e^z + e^{-z})$$



Simple Neural Network

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



[Example from Greg Durrett]

Computing Functions using a Perceptron

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

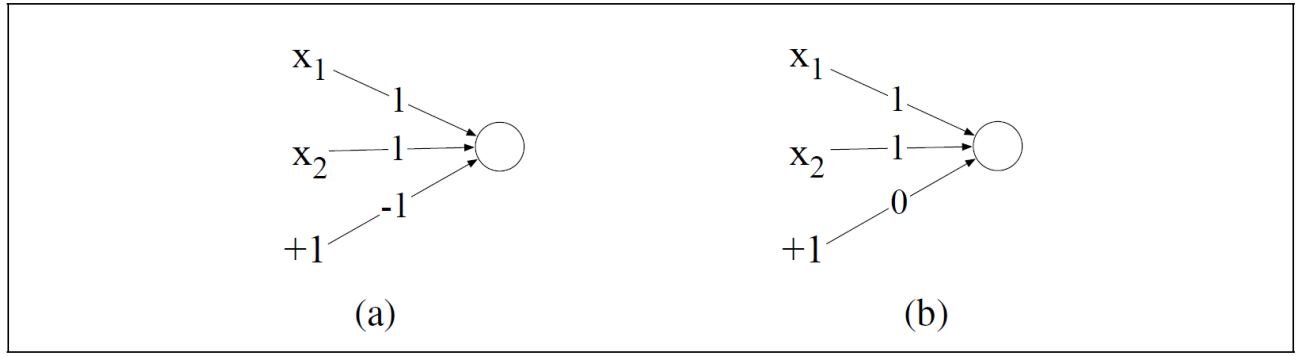
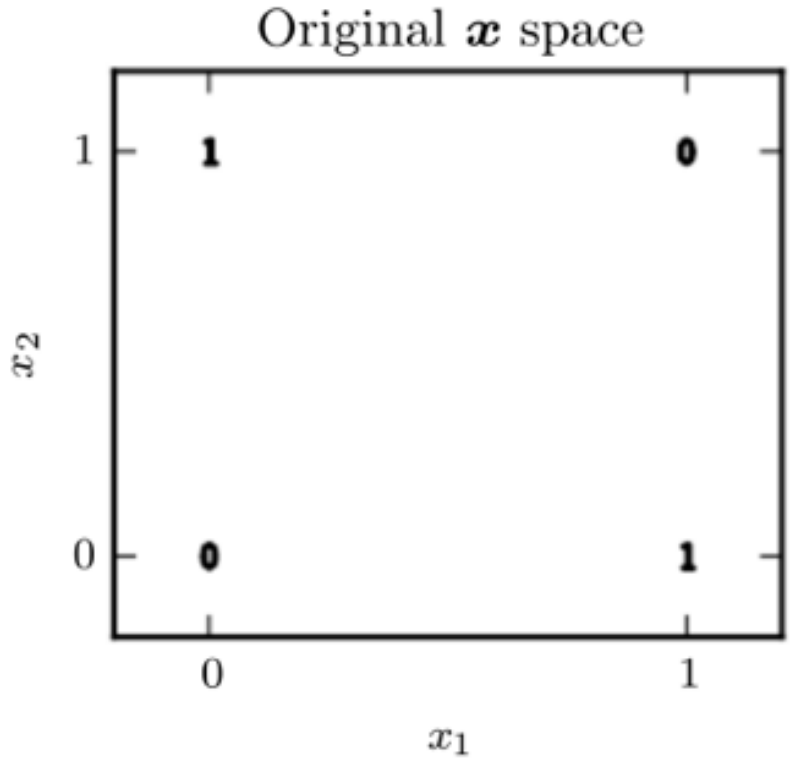


Figure 7.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$. These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

The XOR function



[[0,0], [0]]
[[0,1], [1]]
[[1,0], [1]]
[[1,1], [0]]

The XOR function

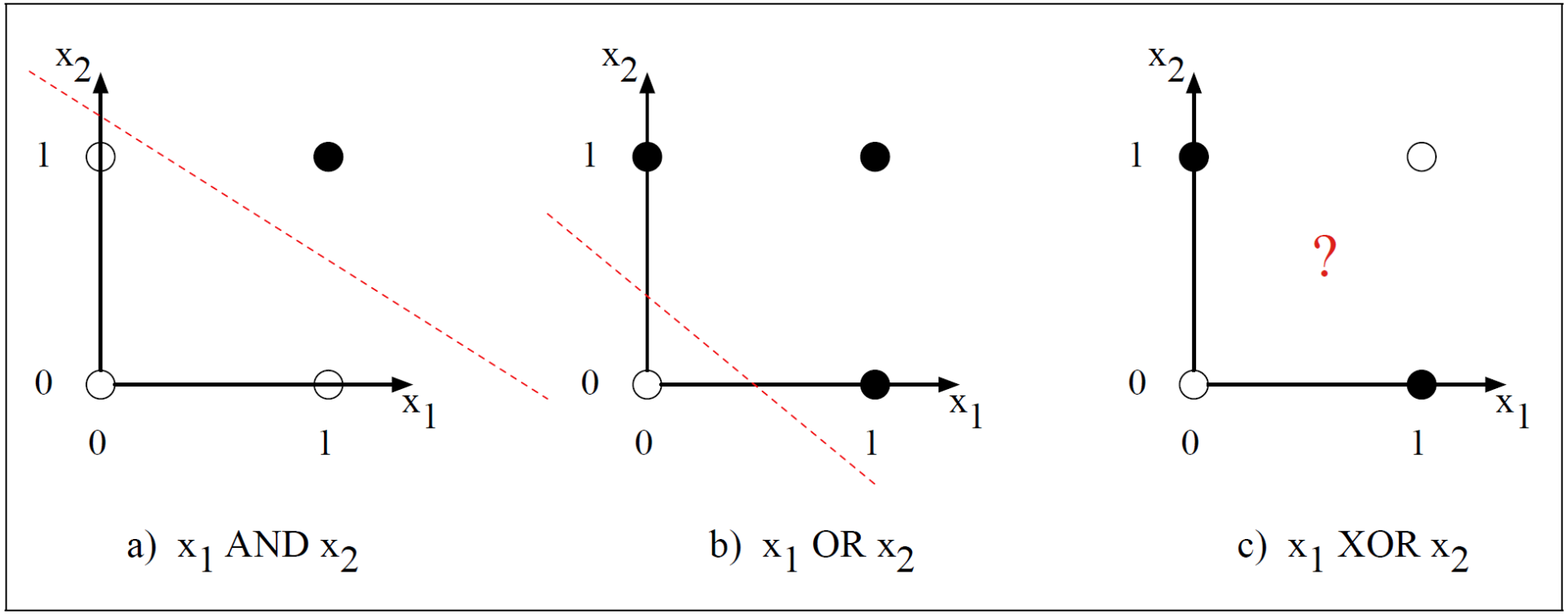


Figure 7.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

One possible solution

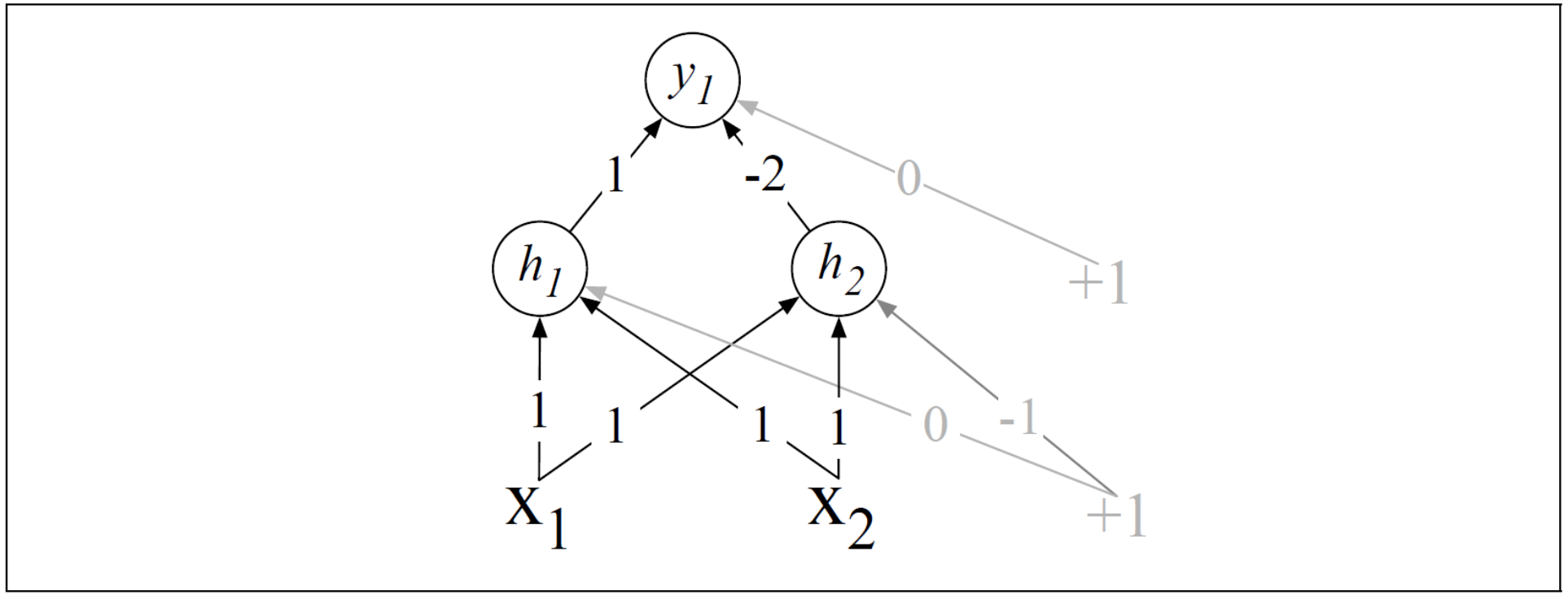


Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

Using a Hidden Layer

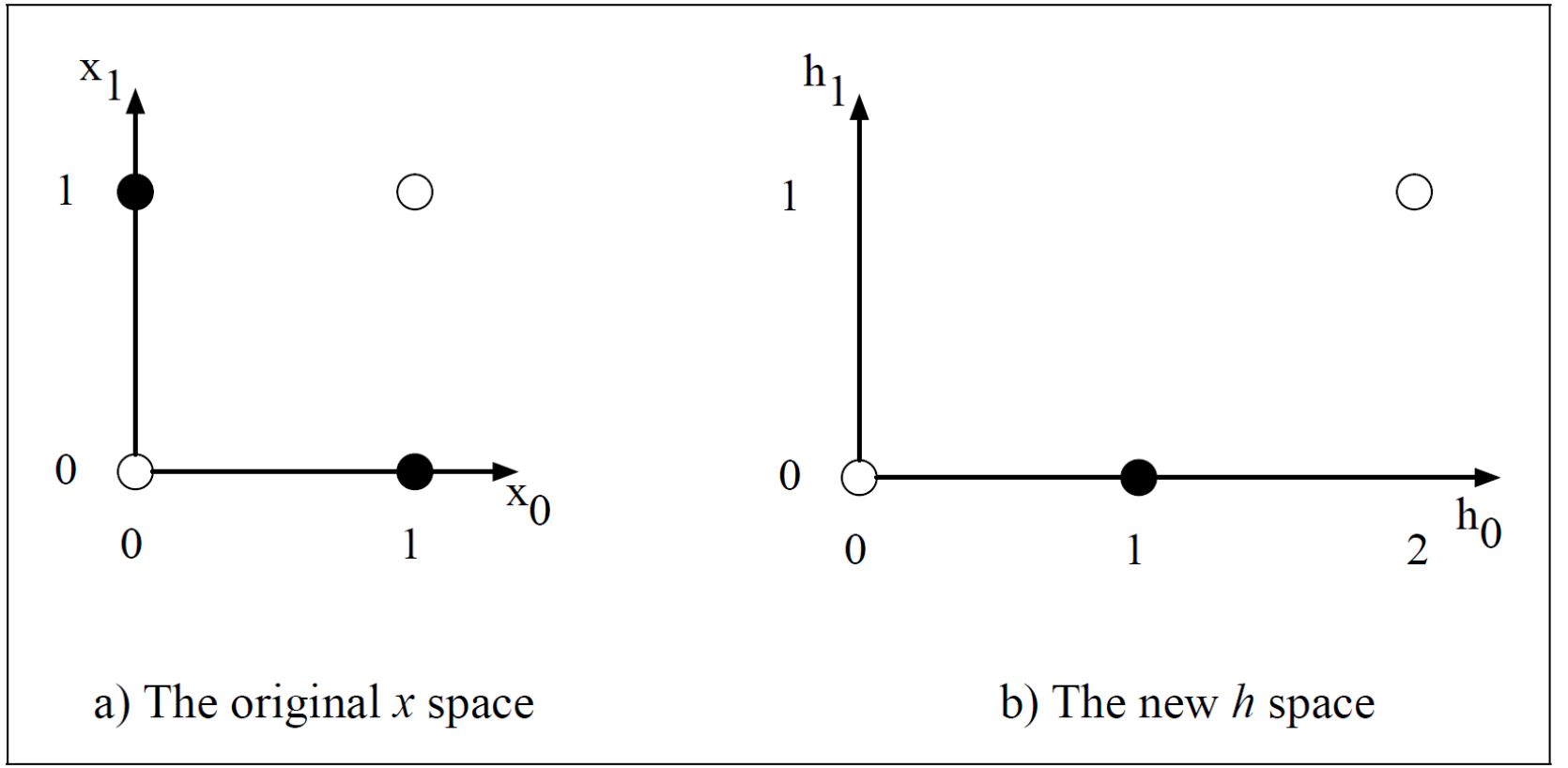


Figure 7.7 The hidden layer forming a new representation of the input. Here is the representation of the hidden layer, h, compared to the original input representation x. Notice that the input point [0 1] has been collapsed with the input point [1 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

Network with a Hidden Layer

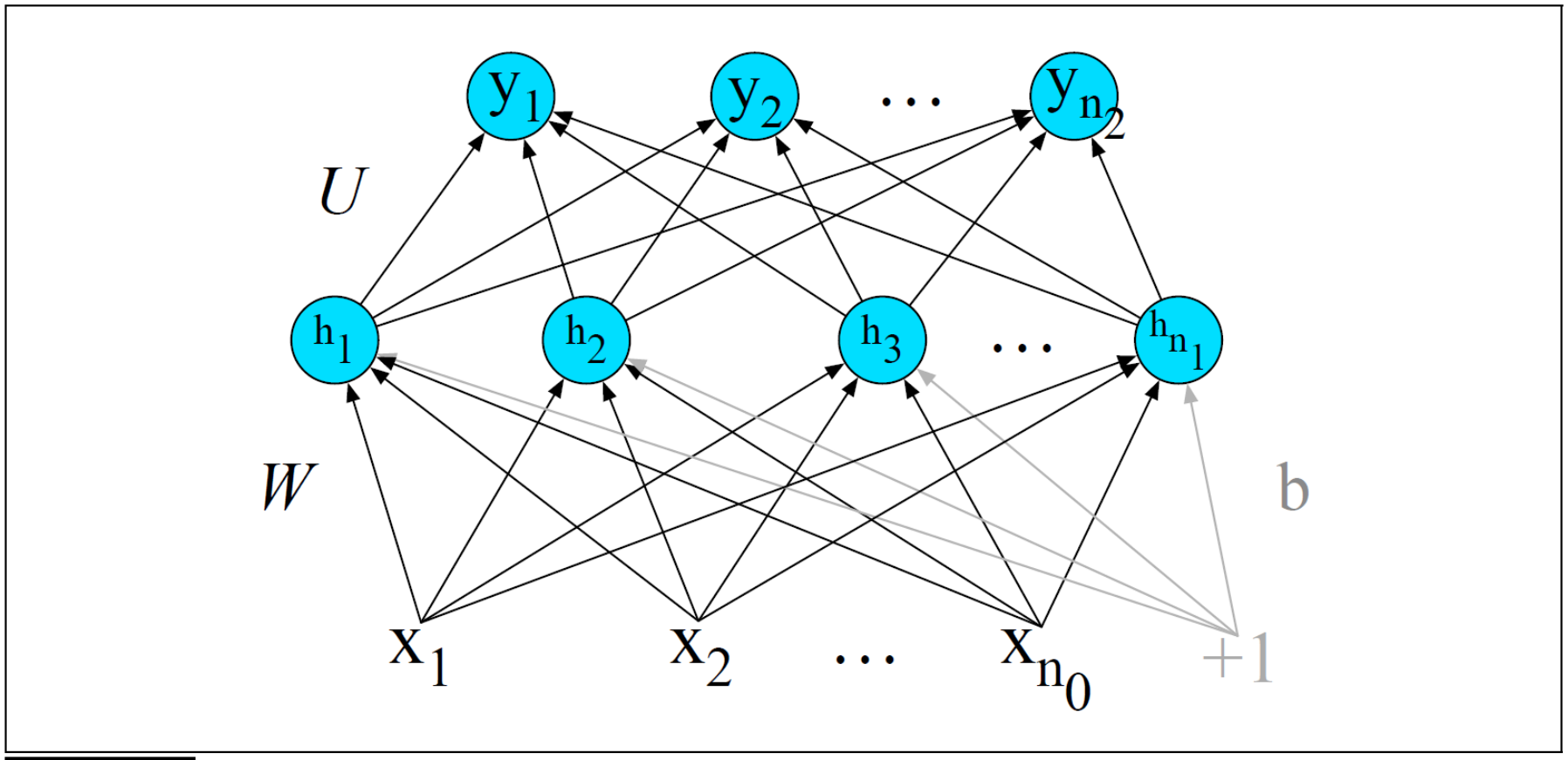


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

$$
\begin{aligned}
h &= \sigma(Wx+b) \\
z &= Uh \\
y &= \text{softmax}(z)
\end{aligned}
$$

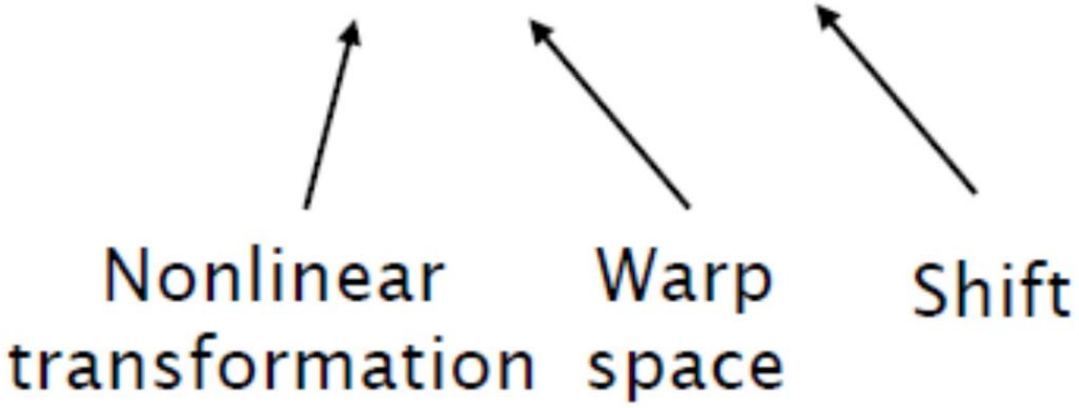
$$
\begin{aligned}
z^{[1]} &= W^{[1]}a^{[0]}+b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]}+b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

A non-linear layer

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

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

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

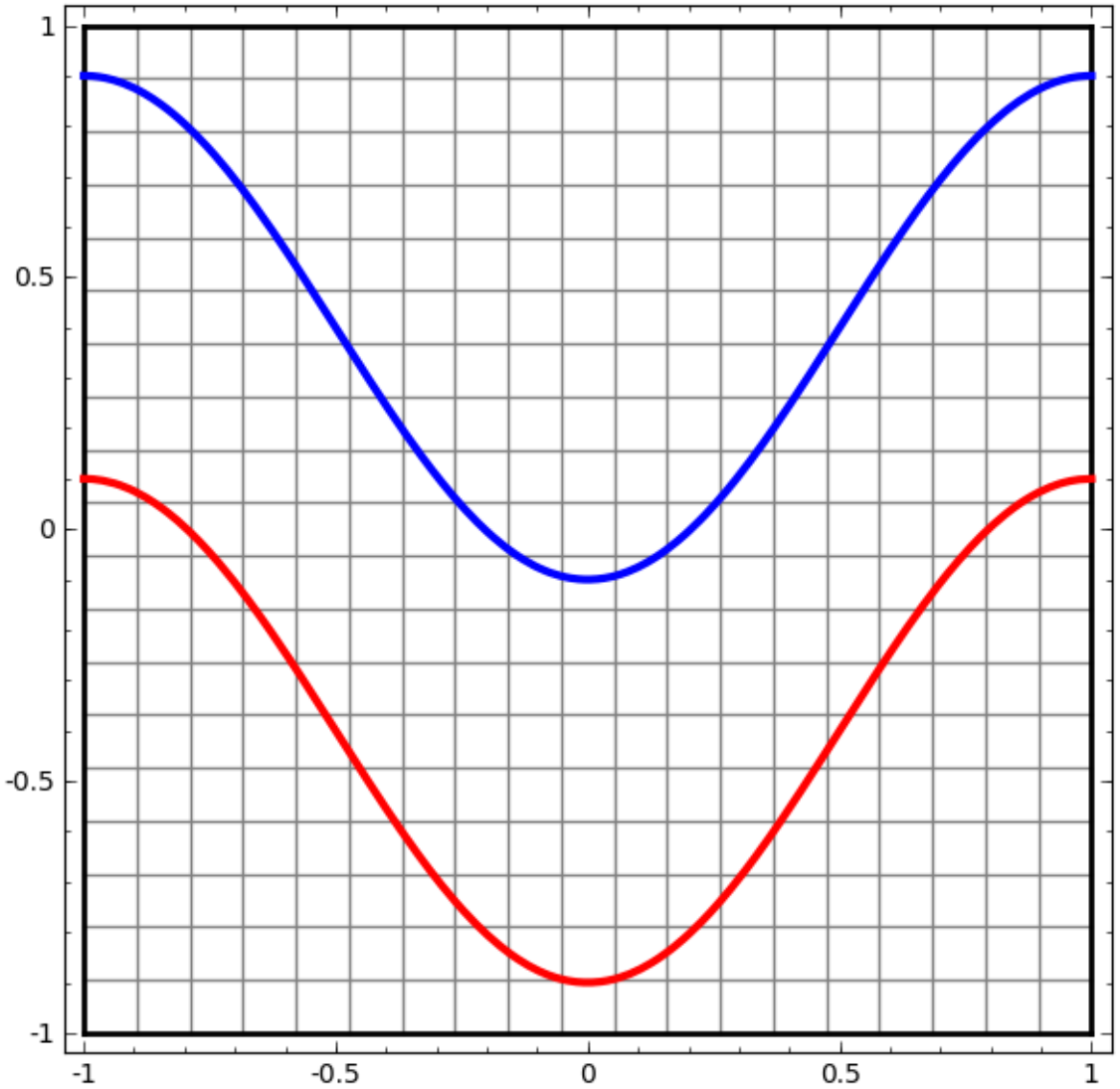


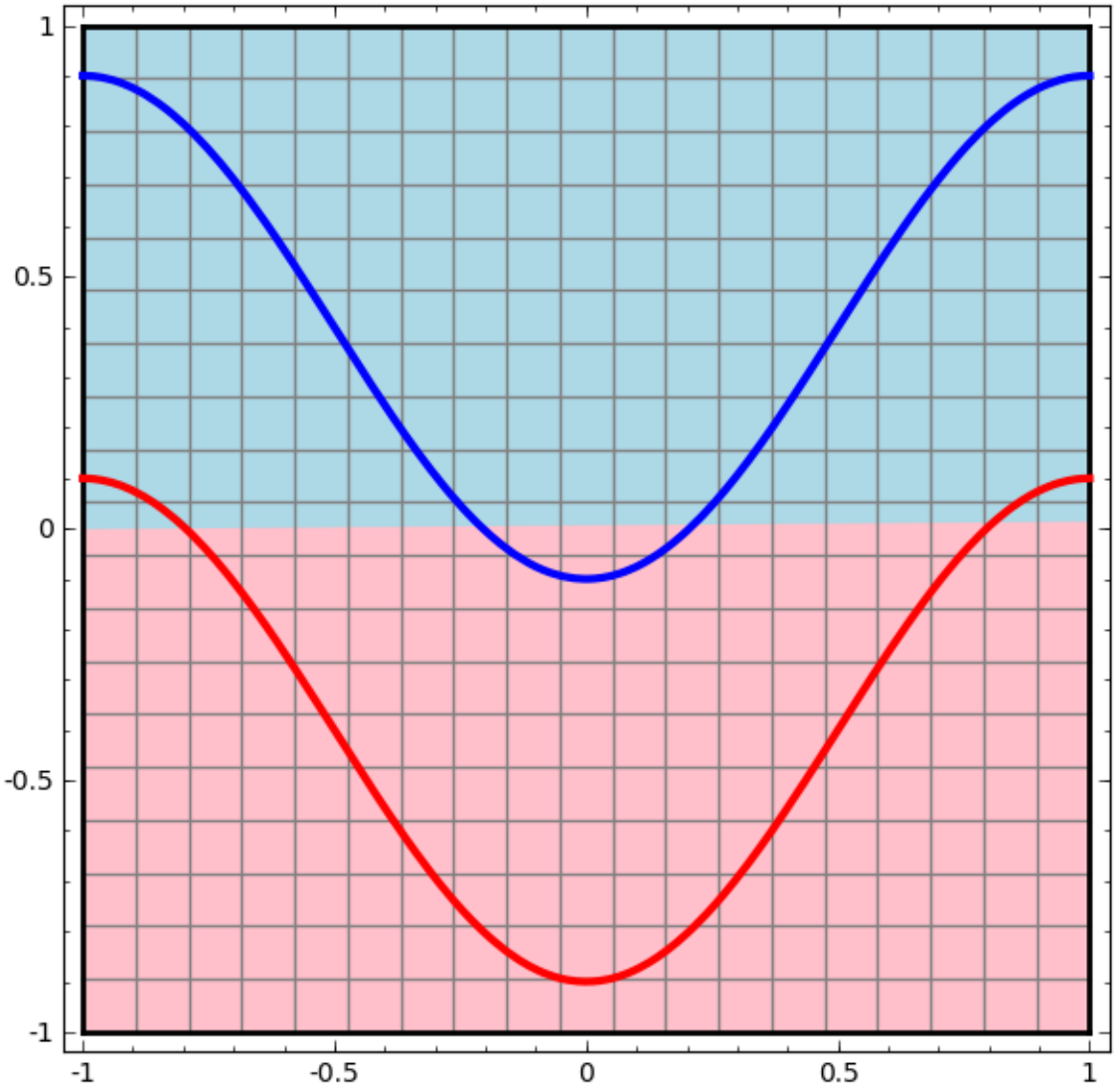
[Greg Durrett]

A tanh layer, **tanh(wx+b)**, consists of:

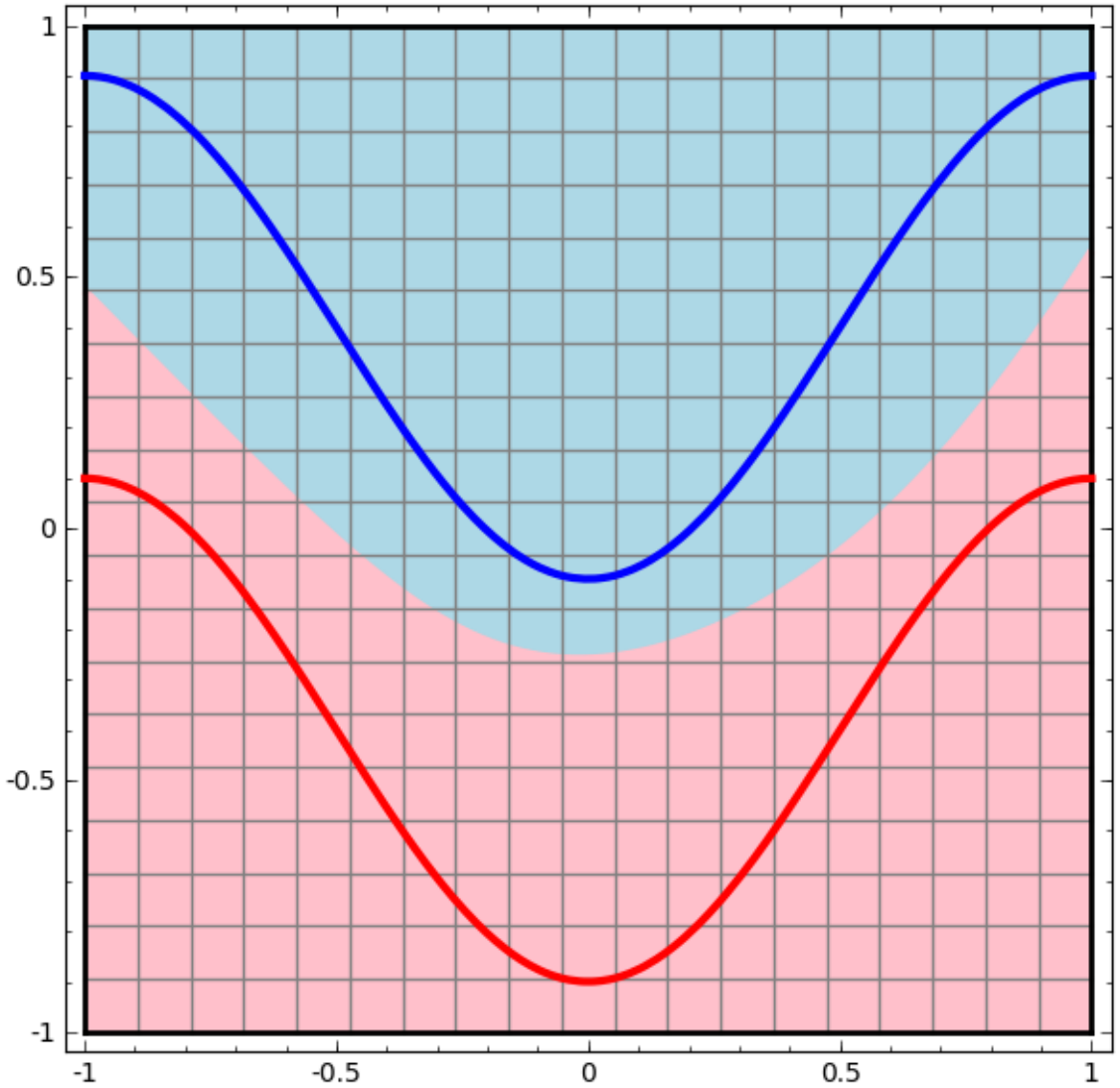
1. A linear transformation by the "weight" matrix WW
2. A translation by the vector bb
3. Point-wise application of tanh.

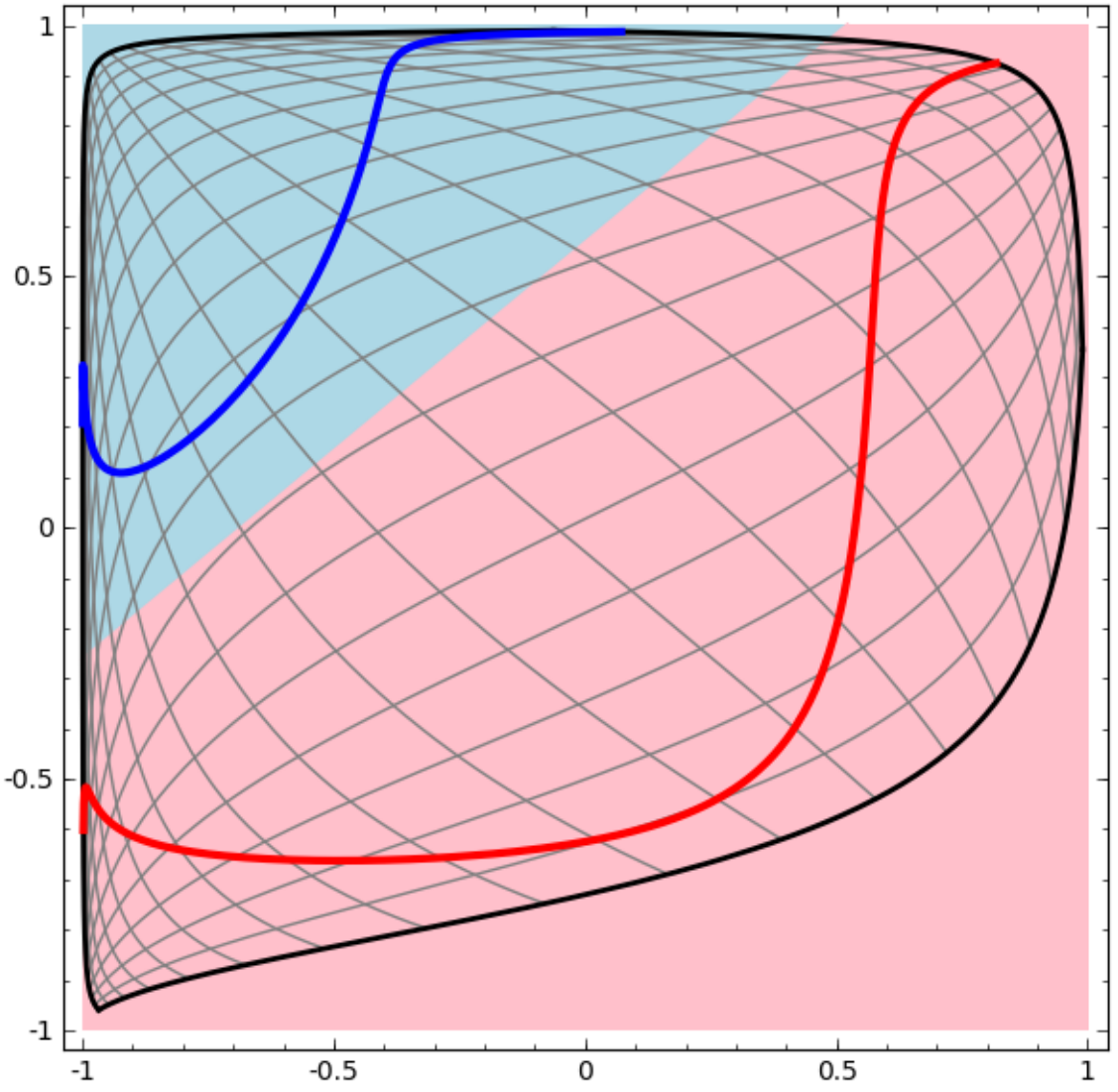
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/



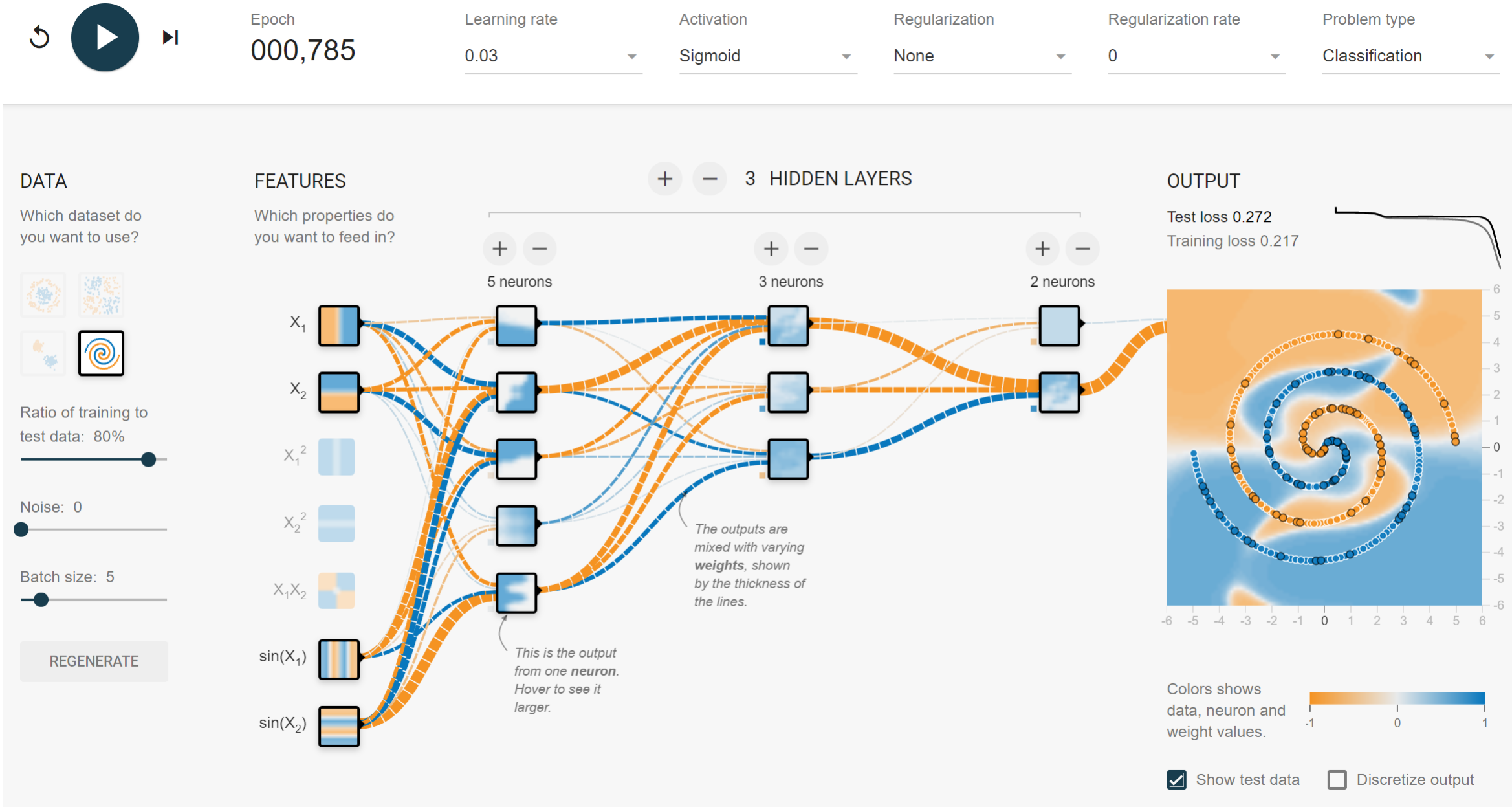


http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

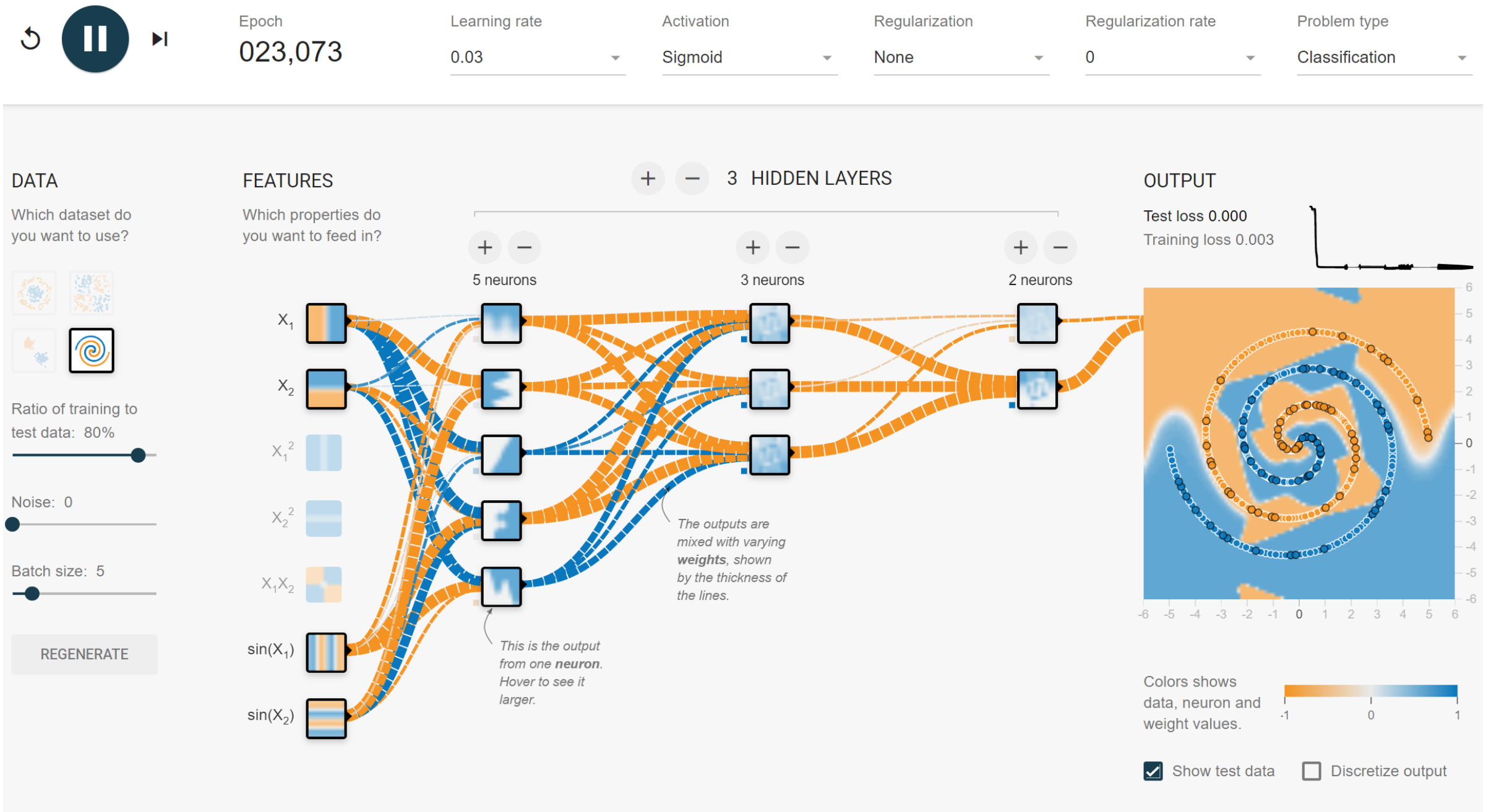




https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/



https://playground.tensorflow.org/



https://playground.tensorflow.org/

Demos

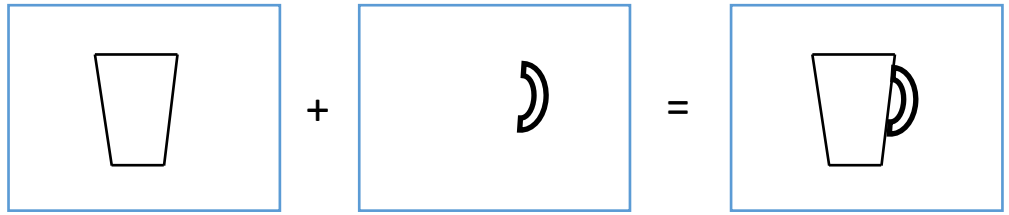
- http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/
- http://www.youtube.com/watch?v=aVId8KMsdUU&feature=BFa&list=LLldMCkmXl4j9_v0HeKdNcRA
- https://cs.stanford.edu/people/karpathy/convnetjs/
- https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

Network with a Hidden Layer

- It is a universal function approximator
  - It can model any continuous function (Hornik 1991)
- This is not true for a simpler network
  - It cannot learn XOR, for example (no one-layer linear classifier can)
- The deeper network can represent XOR
  - as a disjunction of two lower-level features
- An even deeper network can also represent an arbitrary function
  - but with fewer nodes

Deeper Networks

- Multiple layers
  - Input layer
  - Low-level feature layer
  - Higher-level feature layer
  - Output layer
- Intuition
  - Learning features automatically



Example (from Lee et al. 2009)

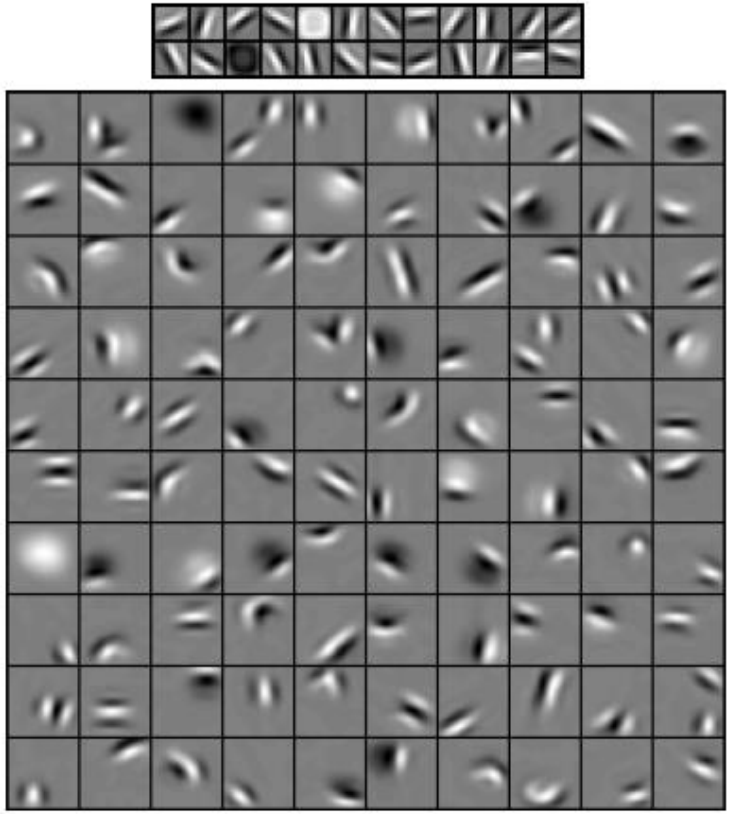


Figure 2. The first layer bases (top) and the second layer bases (bottom) learned from natural images. Each second layer basis (filter) was visualized as a weighted linear combination of the first layer bases.

Example (from Lee et al. 2009)

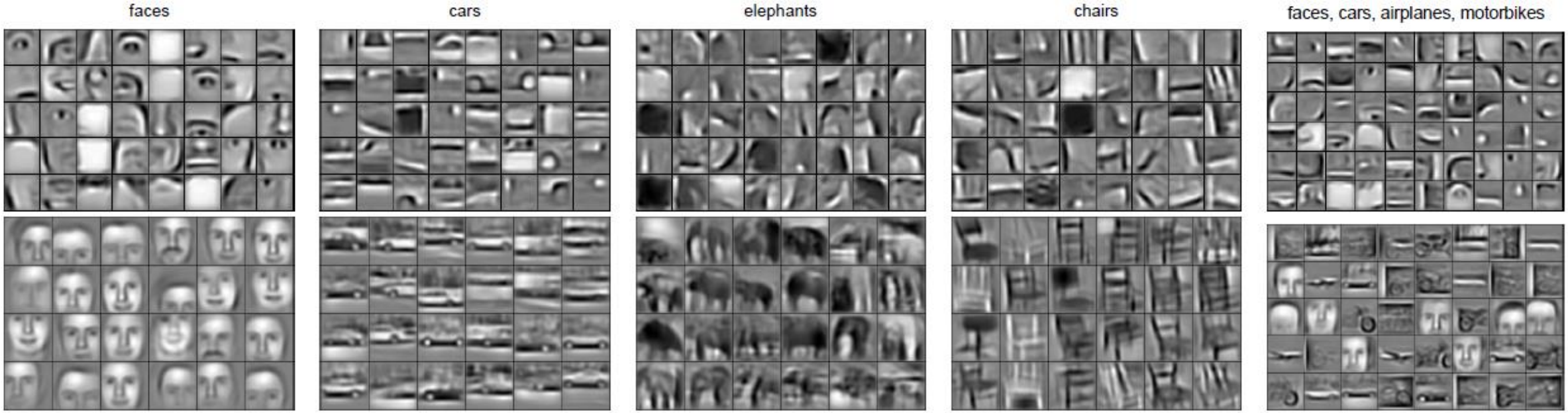


Figure 3. Columns 1-4: the second layer bases (top) and the third layer bases (bottom) learned from specific object categories. Column 5: the second layer bases (top) and the third layer bases (bottom) learned from a mixture of four object categories (faces, cars, airplanes, motorbikes).

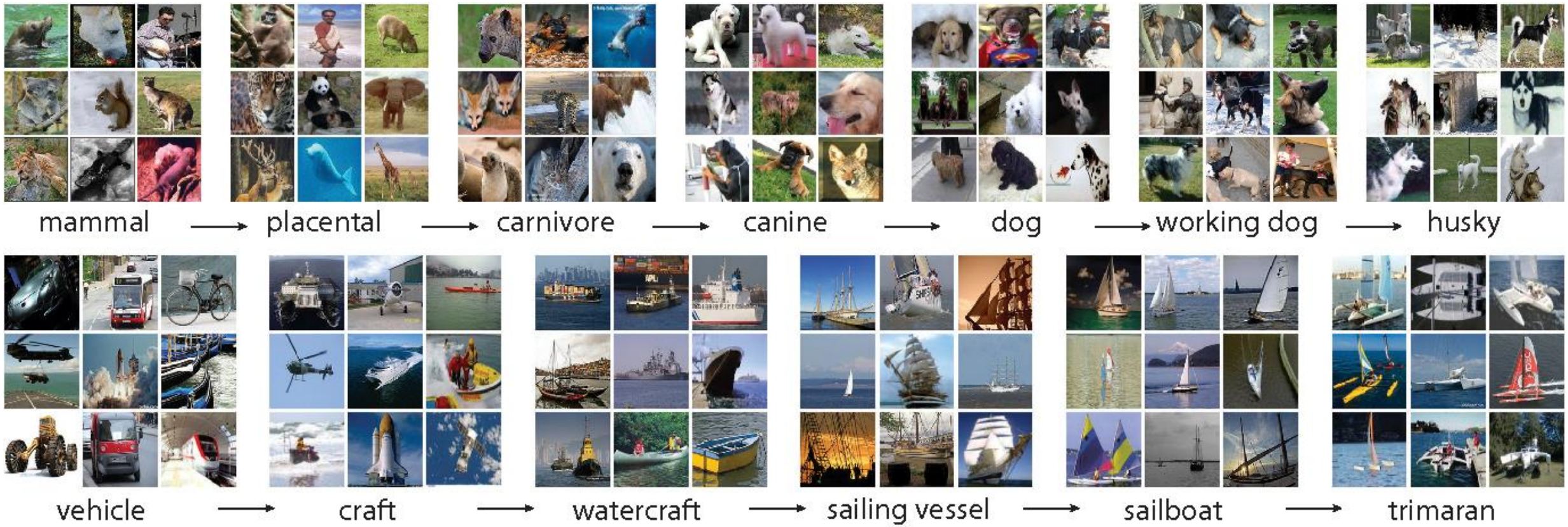
What is Deep Learning

- Architecture
  - Neural Networks with multiple layers
  - Non-linearities (e.g., sigmoid, tanh)
- Use
  - Single-layer NN used as simple classifier
  - Multi-layer NN can express more complex functions and learn complex features
- Hot area of research
  - Very popular these days, especially in speech and vision, but also in NLP
  - Works best with high-performance computers using GPUs and very large data sets

ImageNet

- 14 M images, manually labeled through crowdsourcing
- 20,000 categories
- https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world/

ImageNet

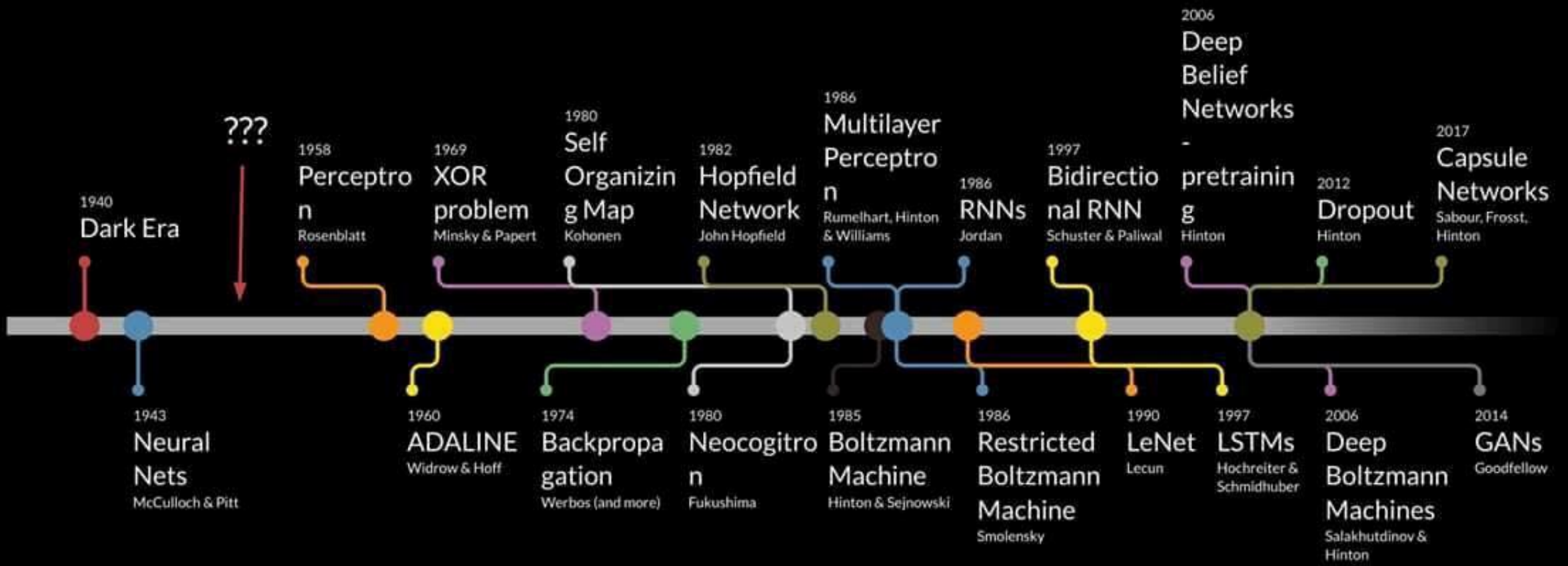


[Image from Deng et al. and Fei-Fei Li 2011]

Why Deep Learning?

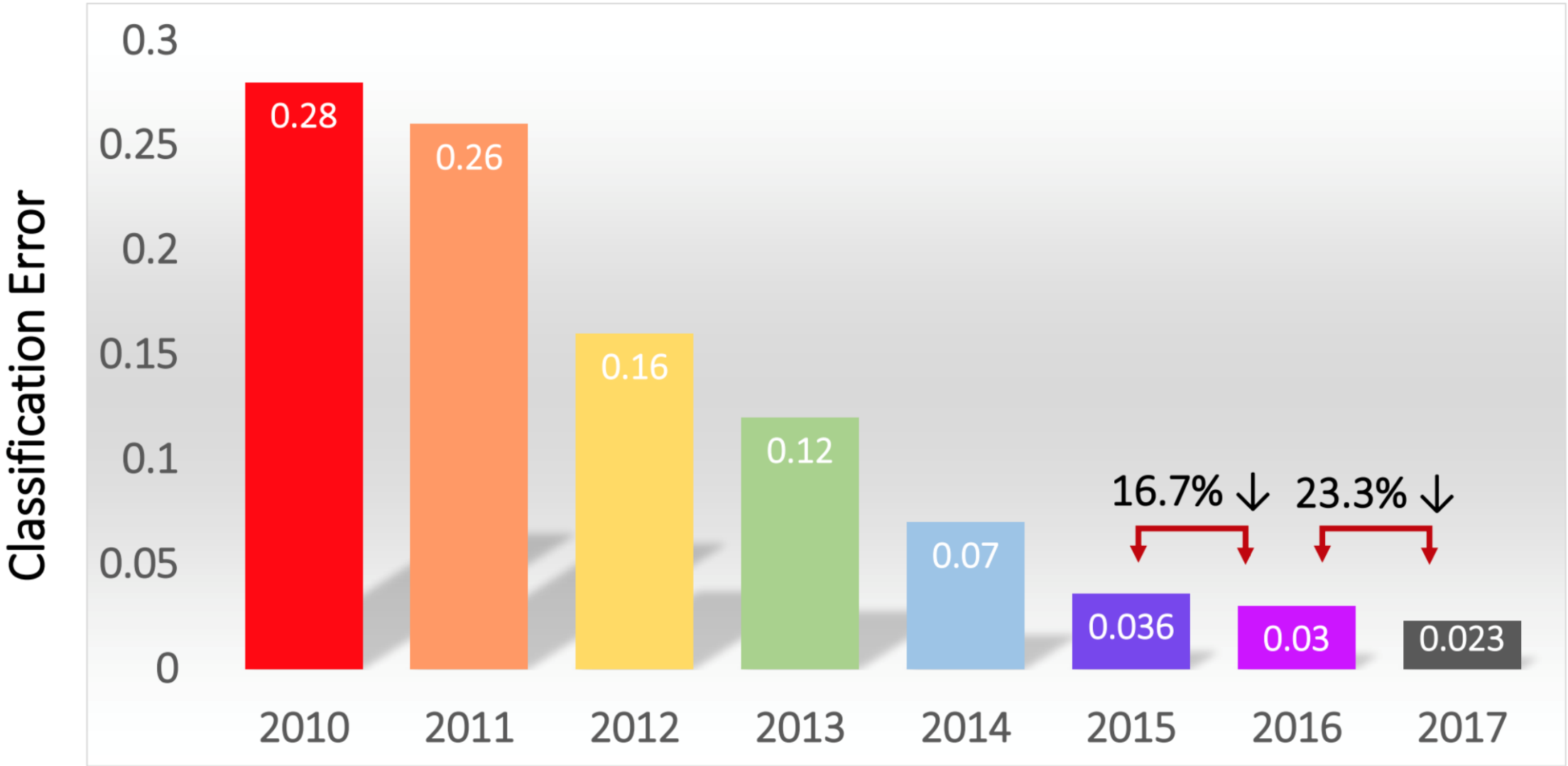
- Amazing results
  - Speech: Dahl et al. 2010 – 30% improvement in WER (word error rate)
  - Vision: ImageNet (Krizhevsky et al. 2012) – scarily complex architecture
- The big players (Google, Facebook, Baidu, Microsoft, IBM…) are investing billions in DL
- The hot new thing?
  - Actually, many of the architectures that we'll talk about were invented in the 1980s and 1990s
  - What's new is hardware (GPU) that can use these architectures at scale

Deep Learning Timeline



Made by Favio Vázquez

Classification Results (CLS)



https://arstechnica.com/science/2018/12/how-computers-got-shockingly-good-at-recognizing-images/

ImageNet Architecture

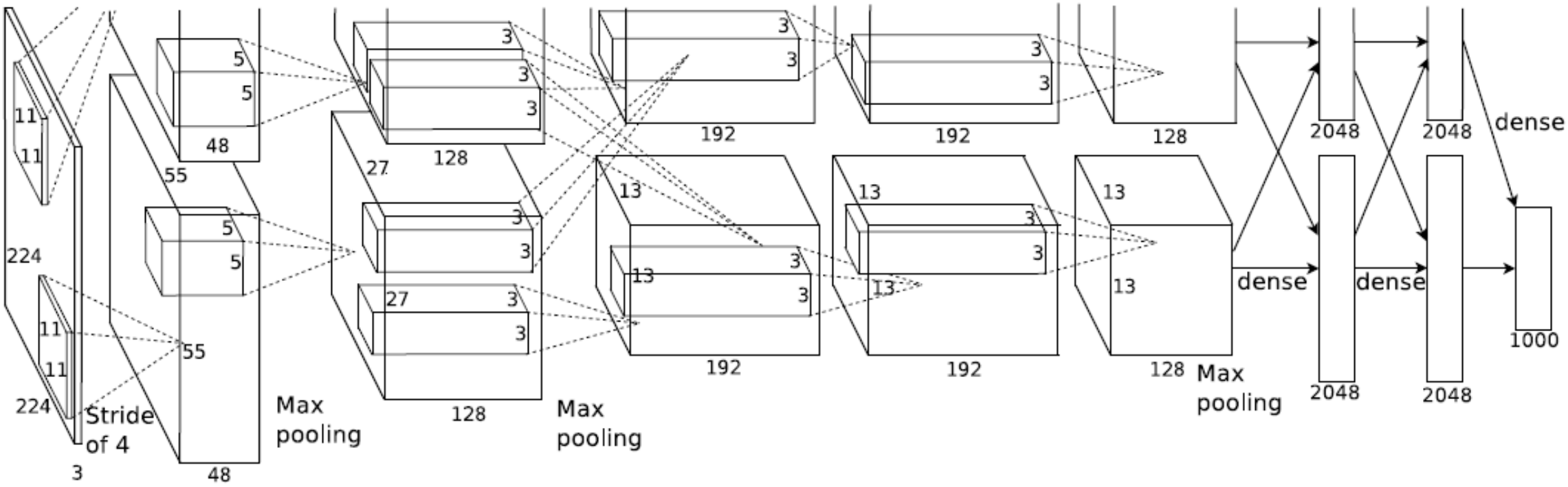
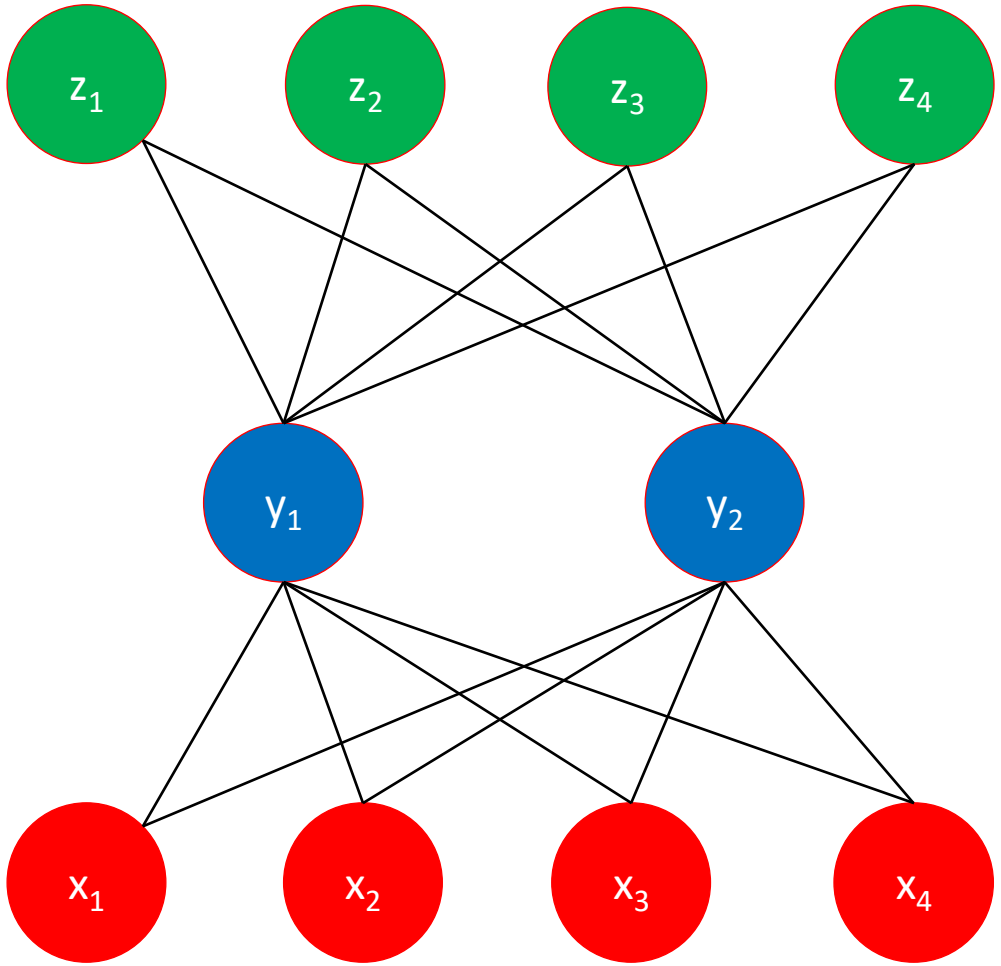


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Autoencoder



[output layer]

$z = x$

[hidden layer]

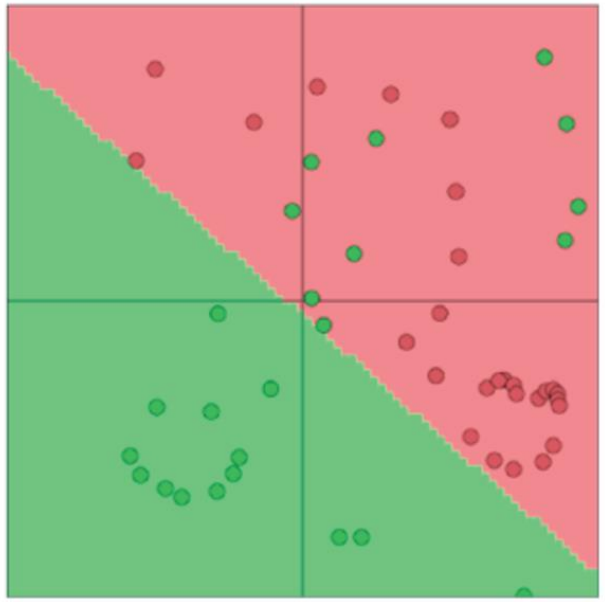
[input layer]

Autoencoder

- The size of the hidden layer should be smaller than the size of the input layer.
  - Otherwise the hidden layer will learn the identity mapping
- Minimize loss
  - difference between z_i and x_i
- Denoising autoencoder
  - Add some domain-independent noise to the input
  - Example: rotate or stretch the image
  - Still try to have the output match the input
- The autoencoder is closely related to SVD but is nonlinear

Why the non-linearities?

- Neural networks can learn much more complex functions and nonlinear decision boundaries



The capacity of the network increases with more hidden units and more hidden layers

How if we remove activation function? [Chris Manning]

Loss Functions

- Binary classification

$$y = \sigma(\mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)})$$

$$\mathcal{L}(y, y^*) = -y^* \log y - (1 - y^*) \log (1 - y)$$

- Regression

$$y = \mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)}$$

$$\mathcal{L}_{\text{MSE}}(y, y^*) = (y - y^*)^2$$

- Multi-class classification (C classes)

$$y_i = \text{softmax}_i(\mathbf{W}^{(o)}\mathbf{h}_2 + \mathbf{b}^{(o)}) \quad \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}, \mathbf{b}^{(o)} \in \mathbb{R}^C$$

$$\mathcal{L}(y, y^*) = -\sum_{i=1}^{C} y_i^* \log y_i$$

The question again becomes how to compute: $\nabla_\theta \mathcal{L}(\theta)$

$$\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \mathbf{w}^{(o)}, b^{(o)}\}$$

[Danqi Chen]

Optimization

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta)$$

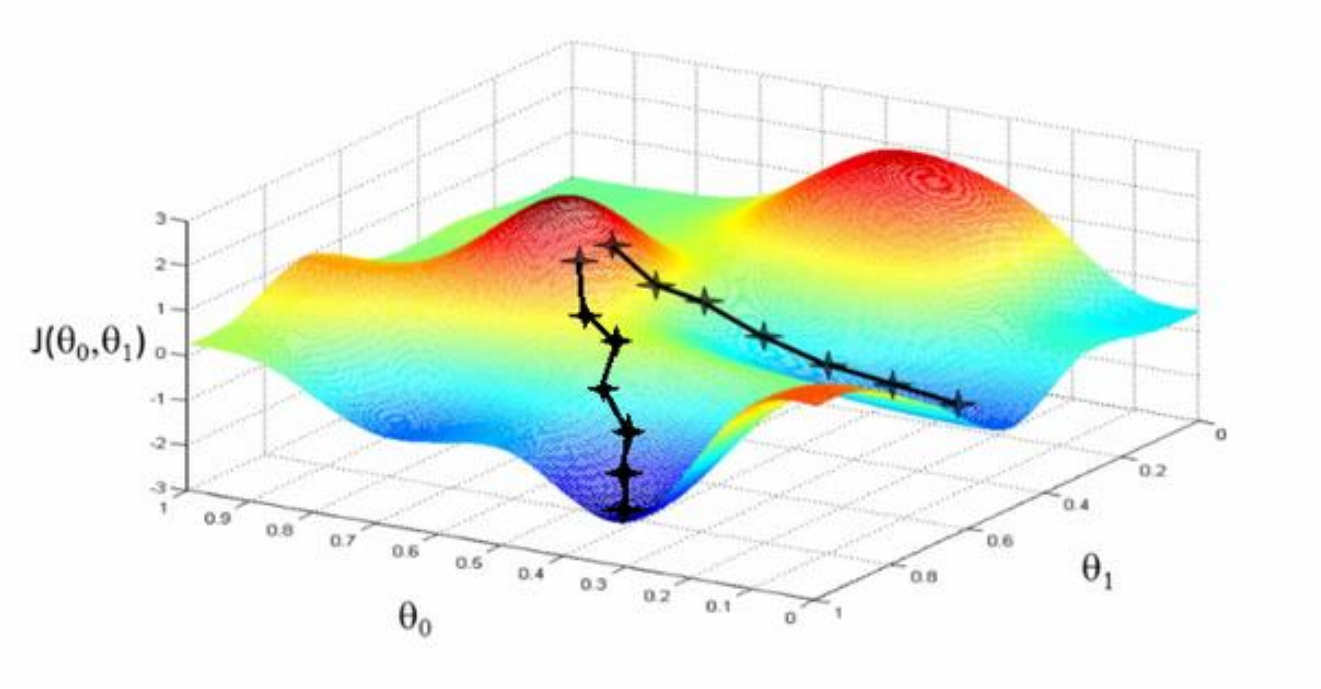
- Logistic regression is convex: one global minimum
- Neural networks are non-convex and not easy to optimize
- A class of more sophisticated "adaptive" optimizers that scale the parameter adjustment by an accumulated gradient.
  - **Adam**
  - Adagrad
  - RMSprop
  - ...

(Ruder 2016): An overview of gradient descent optimization algorithms

[Danqi Chen]

Gradient Descent & Backpropagation

- Backpropagation is an efficient way to compute gradients in a computational graph
- It is based on the chain rule for derivatives/gradients.
- So, it doesn't need to update the parameters blindly.

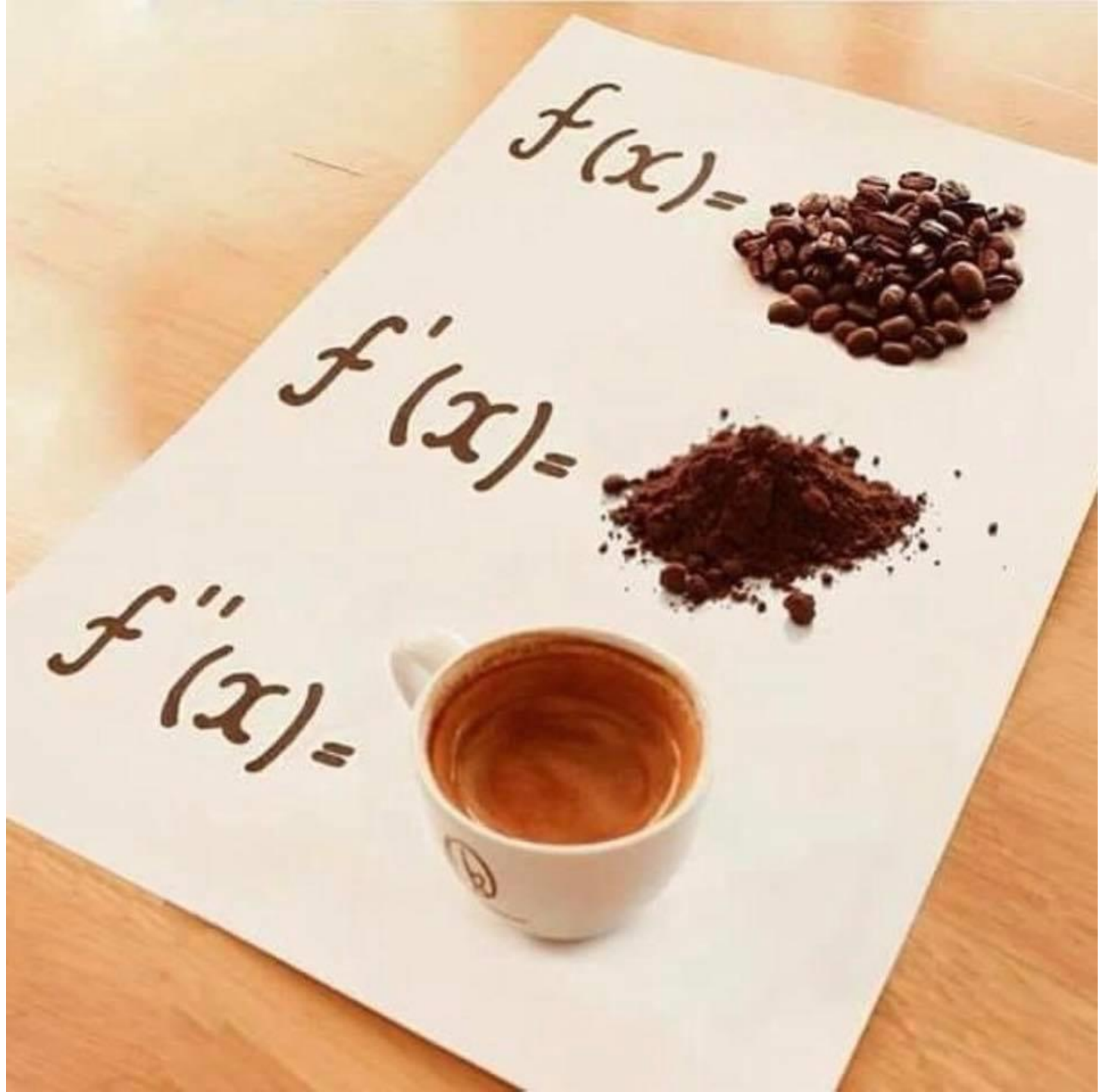


Introduction to NLP

713.

Training Neural Networks

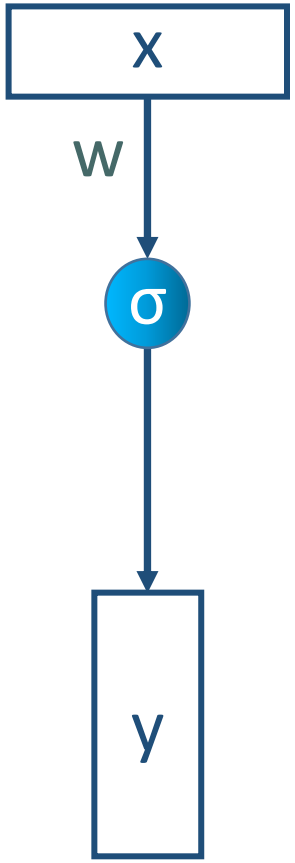
Backpropagation


f(x)=
f'(x)=
f''(x)=

$$\frac{d\,\text{  }}{dx} = \text{  }$$

$$\int \text{  }\, dx = \text{  } + \mathbf{C}$$

A Simplified Diagram



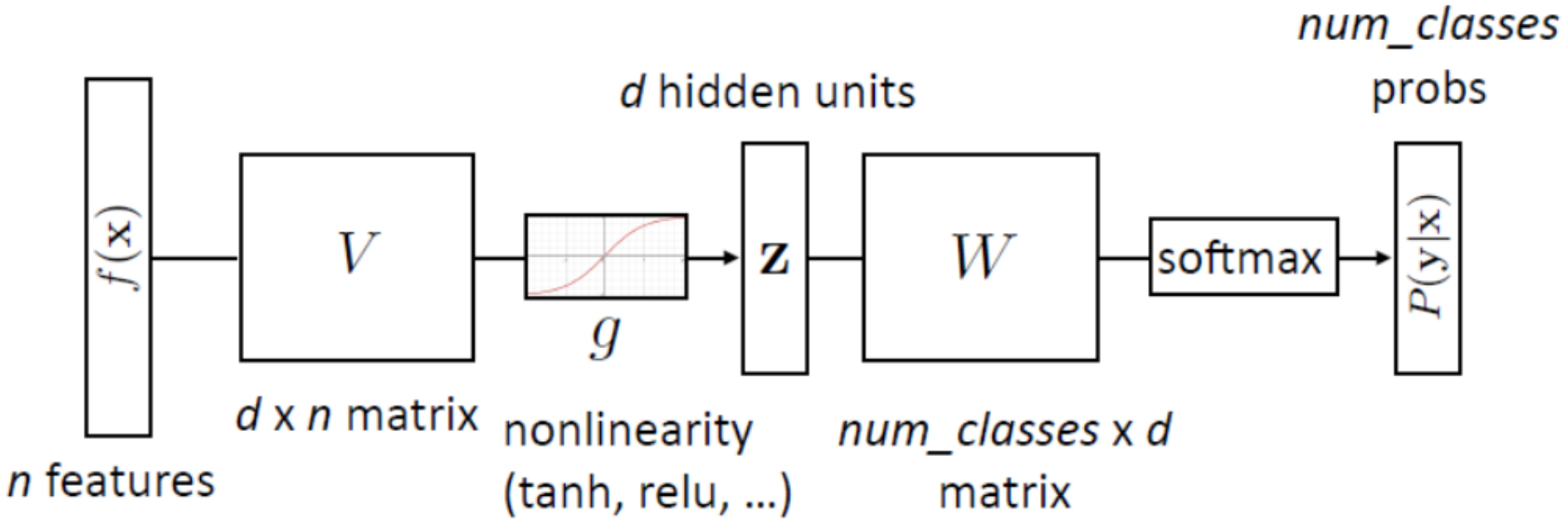
Example

- Without a hidden layer

$$p(\boldsymbol{y}|\boldsymbol{x}) = \text{softmax}_{\text{y}}(Wf(\boldsymbol{x}))$$

- With a hidden layer

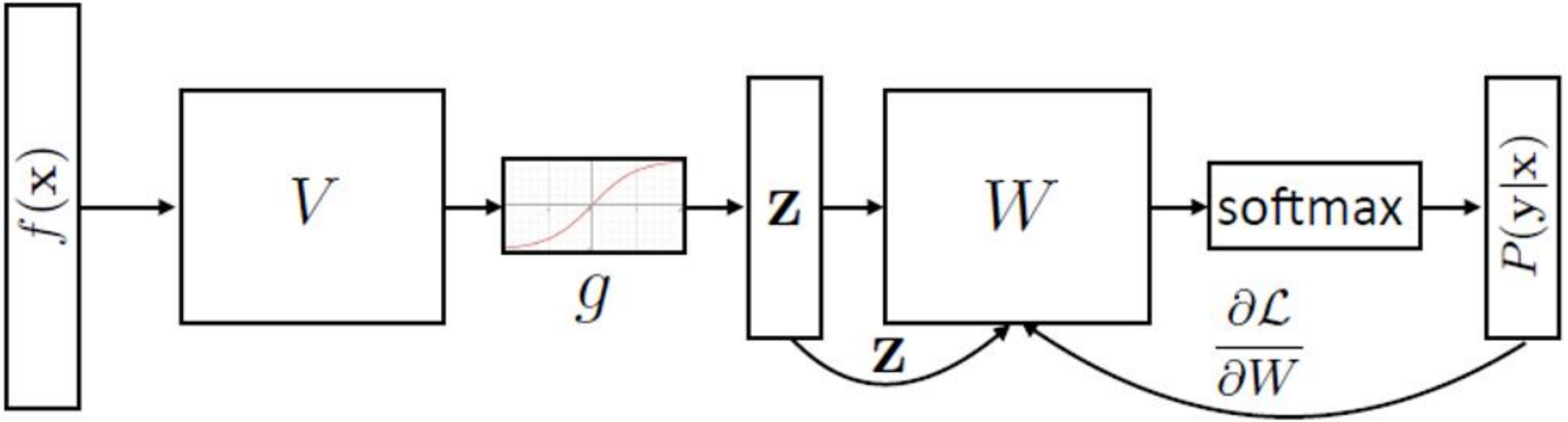
$$p(\boldsymbol{y}|\boldsymbol{x}) = \text{softmax}_{\text{y}}(Wg(Vf(\boldsymbol{x})))$$



[Greg Durrett]

Classification Example

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



[Greg Durrett]

Useful Derivatives (1)

$$\frac{d(a \cdot z)}{dz} = a$$

$$\frac{d \log(z)}{dz} = \frac{1}{z}$$

$$\frac{de^z}{dz} = e^z$$

Chain rule: $\frac{df(g(z))}{dz} = \frac{df(g(z))}{dg(z)} \cdot \frac{dg(z)}{dz}$

Useful Derivatives (2)

$$g_{\text{logistic}}(z) = \frac{1}{1+e^{-z}}$$

$$\begin{aligned}
g'_{\text{logistic}}(z) &= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) \\
&= \frac{e^{-z}}{(1+e^{-z})^2}(\text{chain rule}) \\
&= \frac{1+e^{-z}-1}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}}{(1+e^{-z})^2} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= \frac{1}{(1+e^{-z})} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= g_{\text{logistic}}(z) - g_{\text{logistic}}(z)^2 \\
&= g_{\text{logistic}}(z)(1 - g_{\text{logistic}}(z))
\end{aligned}$$

Two negatives cancel out

https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-activation-functions/

Useful Derivatives (3)

$$
\begin{aligned}
g_{\tanh}(z) &= \frac{\sinh(z)}{\cosh(z)} \\
&= \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}
\end{aligned}
$$

$$
\begin{aligned}
g'_{\tanh}(z) &= \frac{\partial}{\partial z}\frac{\sinh(z)}{\cosh(z)} \\
&= \frac{\frac{\partial}{\partial z}\sinh(z)\times\cosh(z)-\frac{\partial}{\partial z}\cosh(z)\times\sinh(z)}{\cosh^{2}(z)} \\
&= \frac{\cosh^{2}(z)-\sinh^{2}(z)}{\cosh^{2}(z)} \\
&= 1-\frac{\sinh^{2}(z)}{\cosh^{2}(z)} \\
&= 1-\tanh^{2}(z)
\end{aligned}
$$

https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-activation-functions/

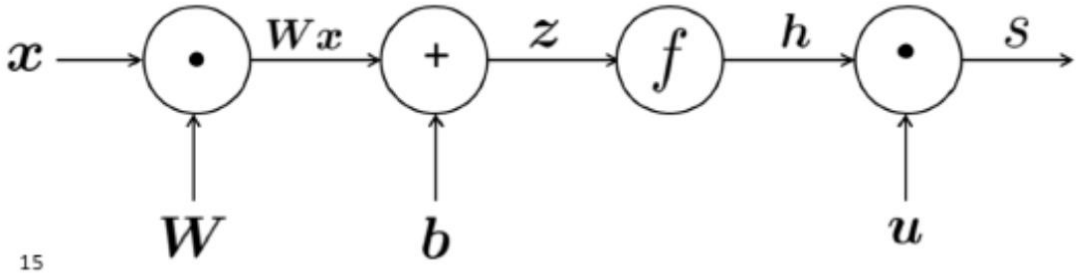
Derivatives of Activation Functions

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

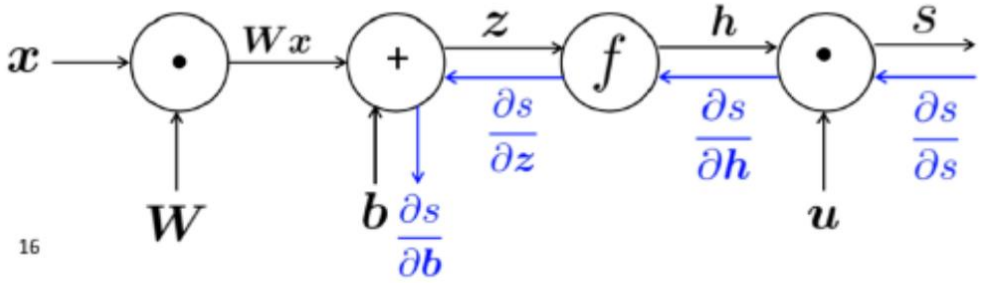
$$\frac{d\tanh(z)}{dz} = 1-\tanh^2(z)$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \;\; x<0 \\ 1 & for \;\; x \geq 0 \end{cases}$$

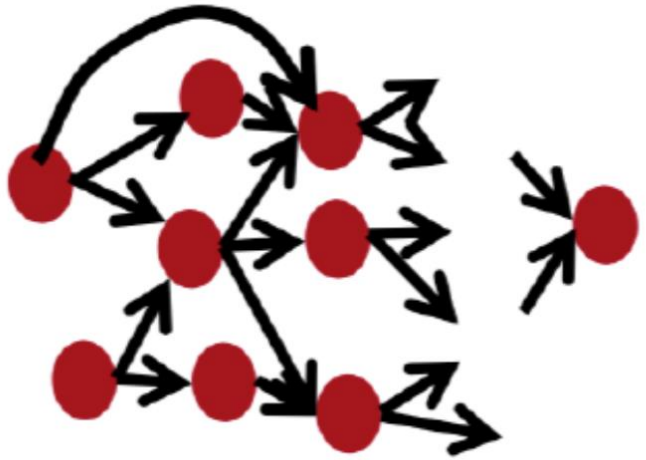
Computational graphs

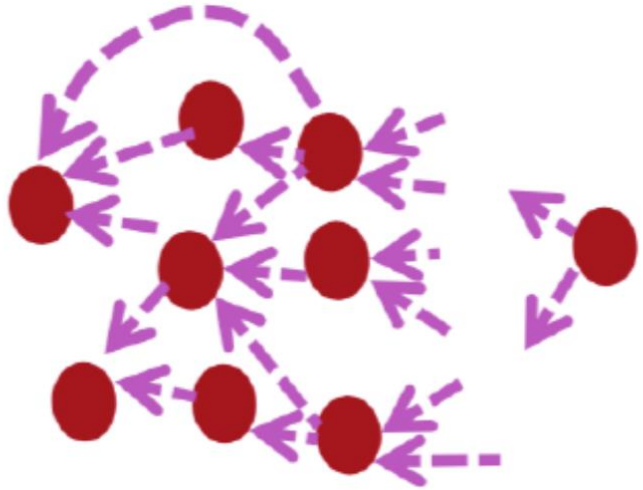


15



16





[Chris Manning]

Computational graphs

A computation graph is a representation of the process of computing a mathematical expression, in which the computation is broken down into separate operations, each of which is modeled as a node in a graph.

Consider computing the function $L(a,b,c) = c(a+2b)$. If we make each of the component addition and multiplication operations explicit, and add names (d and e) for the intermediate outputs, the resulting series of computations is:

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$

We can now represent this as a graph, with nodes for each operation, and directed edges showing the outputs from each operation as the inputs to the next, as in Fig. 7.10. The simplest use of computation graphs is to compute the value of the function with some given inputs. In the figure, we've assumed the inputs $a = 3$, $b = 1$, $c = -2$, and we've shown the result of the **forward pass** to compute the result $L(3,1,-2) = -10$. In the forward pass of a computation graph, we apply each operation left to right, passing the outputs of each computation as the input to the next node.

Computation Graphs

$$
\begin{aligned}
d &= 2 * b \\
e &= a + d \\
L &= c * e
\end{aligned}
$$

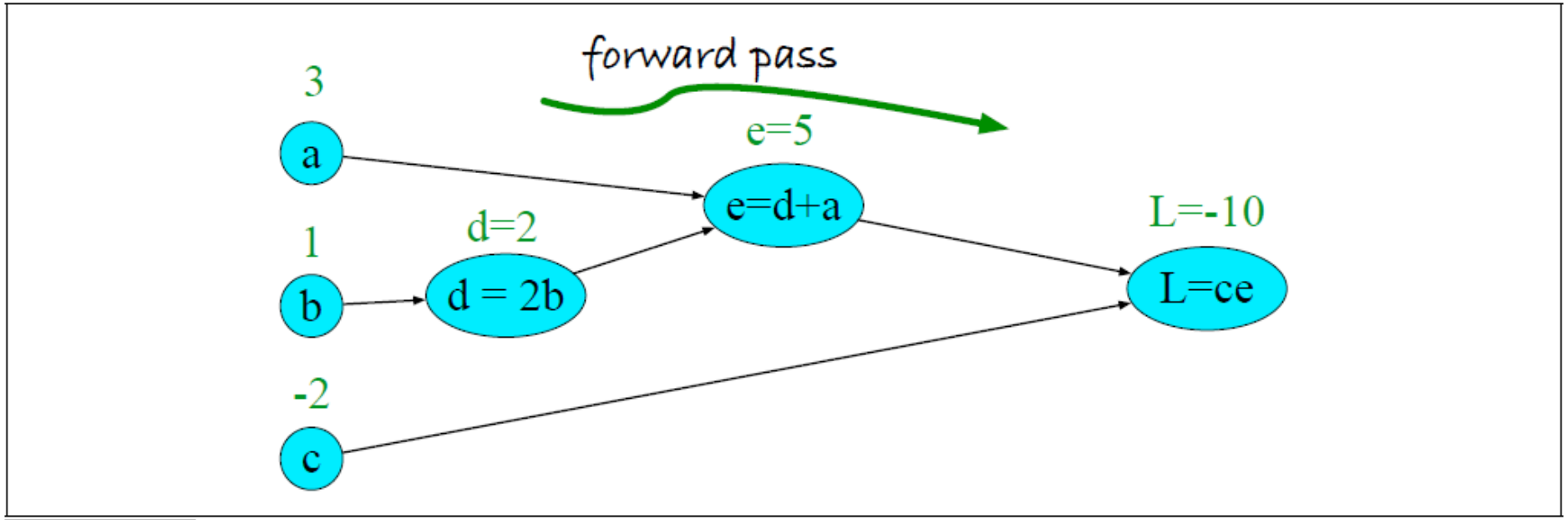


Figure 7.9 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

Chain rule for gradients

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$

$$\frac{\partial L}{\partial c} = e$$

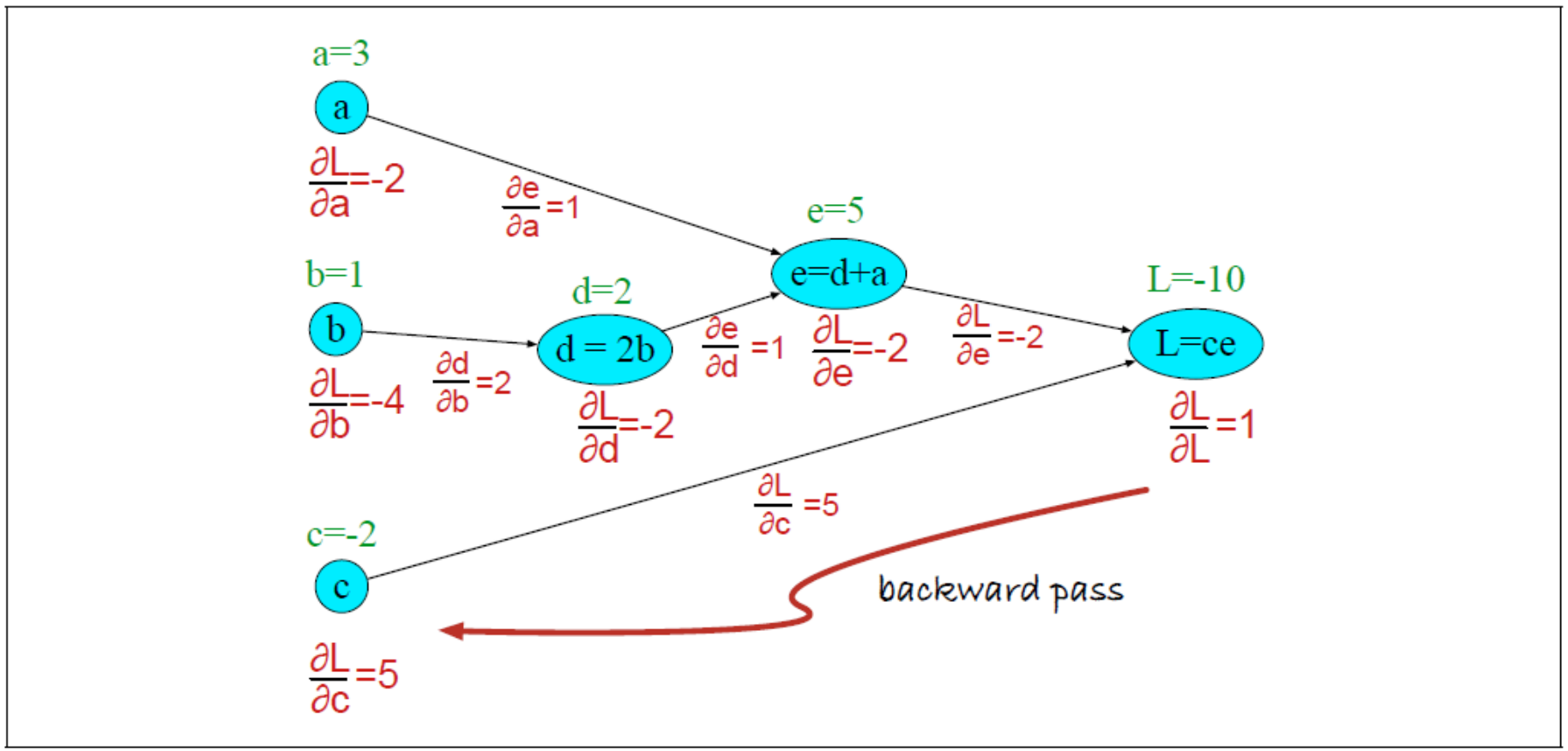
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

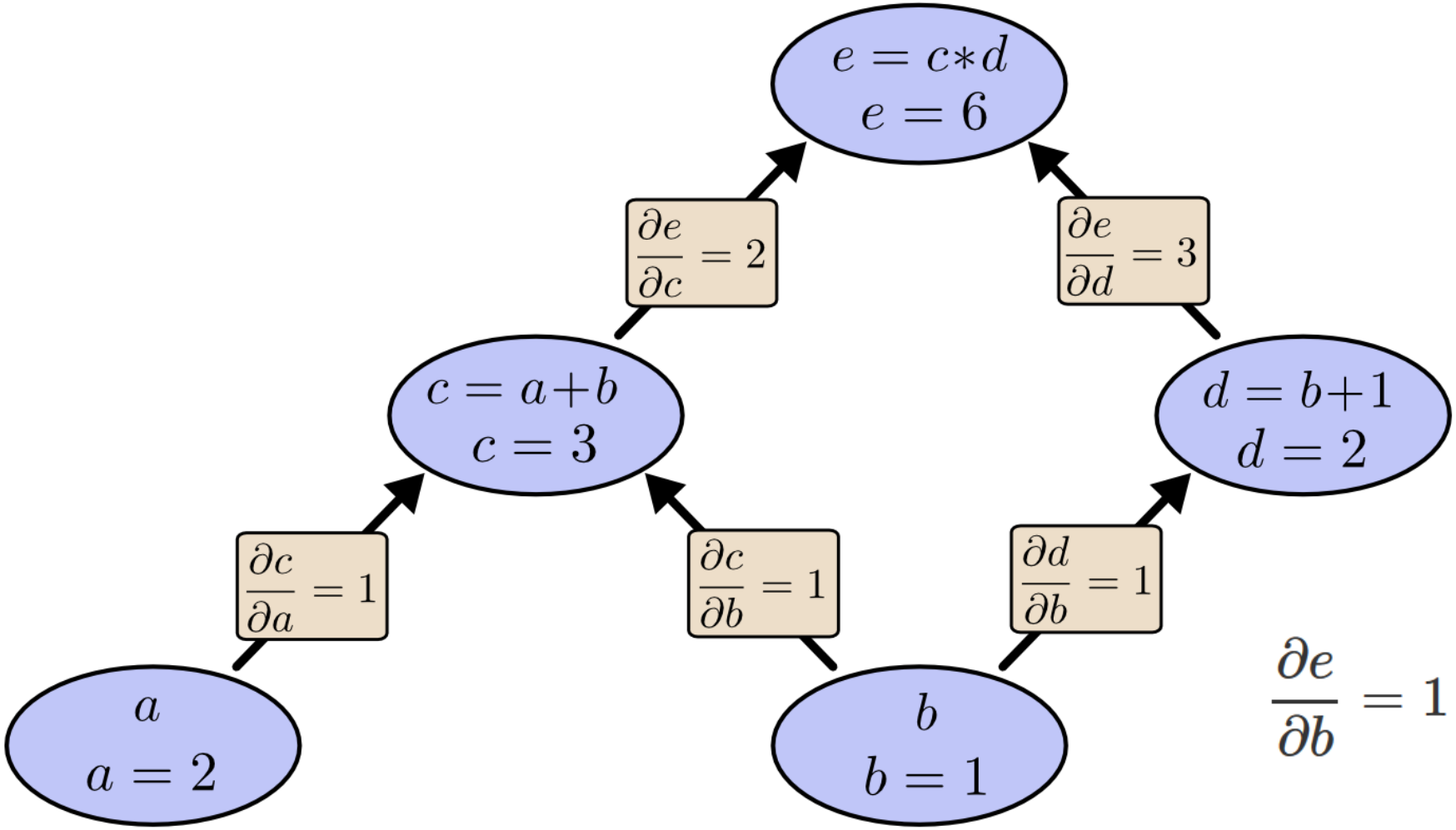
$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$



$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \text{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

Figure 7.10 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

Example



$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

[Chris Olah]

```
# Back-Propagation Neural Networks
# Placed in the public domain.
# Neil Schemenauer <nas@arctrix.com>

import math
import random
import string

random.seed(0)

# calculate a random number where:  a <= rand < b
def rand(a, b):
    return (b-a)*random.random() + a

# Make a matrix (we could use NumPy to speed this up)
def makeMatrix(I, J, fill=0.0):
    m = []
    for i in range(I):
        m.append([fill]*J)
    return m

# our tanh function, tanh is a little nicer than the standard 1/(1+e^-x)
def tanh(x):
    return math.tanh(x)

# derivative of our tanh function, in terms of the output (i.e. y)
def dtanh(y):
    return 1.0 - y**2
```

```
class NN:
    def __init__(self, ni, nh, no):
        # number of input, hidden, and output nodes
        self.ni = ni + 1 # +1 for bias node
        self.nh = nh
        self.no = no

        # activations for nodes
        self.ai = [1.0]*self.ni
        self.ah = [1.0]*self.nh
        self.ao = [1.0]*self.no

        # create weights
        self.wi = makeMatrix(self.ni, self.nh)
        self.wo = makeMatrix(self.nh, self.no)
        # set them to random vaules
        for i in range(self.ni):
            for j in range(self.nh):
                self.wi[i][j] = rand(-0.2, 0.2)
        for j in range(self.nh):
            for k in range(self.no):
                self.wo[j][k] = rand(-2.0, 2.0)

        # last change in weights for momentum
        self.ci = makeMatrix(self.ni, self.nh)
        self.co = makeMatrix(self.nh, self.no)
```

```
def update(self, inputs):
        if len(inputs) != self.ni-1:
            raise ValueError('wrong number of inputs')

        # input activations
        for i in range(self.ni-1):
            #self.ai[i] = tanh(inputs[i])
            self.ai[i] = inputs[i]

        # hidden activations
        for j in range(self.nh):
            sum = 0.0
            for i in range(self.ni):
                sum = sum + self.ai[i] * self.wi[i][j]
            self.ah[j] = tanh(sum)

        # output activations
        for k in range(self.no):
            sum = 0.0
            for j in range(self.nh):
                sum = sum + self.ah[j] * self.wo[j][k]
            self.ao[k] = tanh(sum)

        return self.ao[:]
```

```
def backPropagate(self, targets, N, M):
        if len(targets) != self.no:
            raise ValueError('wrong number of target values')

        # calculate error terms for output
        output_deltas = [0.0] * self.no
        for k in range(self.no):
            error = targets[k]-self.ao[k]
            output_deltas[k] = dtanh(self.ao[k]) * error

        # calculate error terms for hidden
        hidden_deltas = [0.0] * self.nh
        for j in range(self.nh):
            error = 0.0
            for k in range(self.no):
                error = error + output_deltas[k]*self.wo[j][k]
            hidden_deltas[j] = dtanh(self.ah[j]) * error
```

```
# update output weights
        for j in range(self.nh):
            for k in range(self.no):
                change = output_deltas[k]*self.ah[j]
                self.wo[j][k] = self.wo[j][k] + N*change + M*self.co[j][k]
                self.co[j][k] = change
                #print N*change, M*self.co[j][k]

        # update input weights
        for i in range(self.ni):
            for j in range(self.nh):
                change = hidden_deltas[j]*self.ai[i]
                self.wi[i][j] = self.wi[i][j] + N*change + M*self.ci[i][j]
                self.ci[i][j] = change

        # calculate error
        error = 0.0
        for k in range(len(targets)):
            error = error + 0.5*(targets[k]-self.ao[k])**2
        return error
```

```
    def test(self, patterns):
        for p in patterns:
            print(p[0], '->', self.update(p[0]))

    def weights(self):
        print('Input weights:')
        for i in range(self.ni):
            print(self.wi[i])
        print()
        print('Output weights:')
        for j in range(self.nh):
            print(self.wo[j])

    def train(self, patterns, iterations=1000, N=0.5, M=0.1):
        # N: learning rate
        # M: momentum factor
        for i in range(iterations):
            error = 0.0
            for p in patterns:
                inputs = p[0]
                targets = p[1]
                self.update(inputs)
                error = error + self.backPropagate(targets, N, M)
            if i % 100 == 0:
                print('error %-.5f' % error)
```

```
def demo():
    # Teach network XOR function
    pat = [
        [[0,0], [0]],
        [[0,1], [1]],
        [[1,0], [1]],
        [[1,1], [0]]
    ]

    # create a network with two input, two hidden, and one output nodes
    n = NN(2, 2, 1)
    # train it with some patterns
    print "Train"
    n.train(pat)
    # test it
    print 'Input weights:'
    for i in range(n.ni):
      print n.wi[i]
    print
    print 'Output weights:'
    for j in range(n.nh):
      print n.wo[j]
    print ''
    print "Test"
    n.test(pat)

if __name__ == '__main__':
    demo()
```

```
Train
error 0.94250
error 0.04287
error 0.00348
error 0.00164
error 0.00106
error 0.00078
error 0.00063
error 0.00053
error 0.00044
error 0.00038
Input weights:
[3.366612417442899, -1.1983458654063894]
[3.345693031589667, -1.194806124059644]
[-1.6784196591864349, 1.104256244910066]

Output weights:
[2.851039122470194]
[3.320704560933144 5]

Test
([0, 0], '->', [0.0042410815506258 9])
([0, 1], '->', [0.9821508029410748])
([1, 0], '->', [0.9820129388618121])
([1, 1], '->', [-0.0011469114721422528])
```

Computation graph for a 2-layer neural network

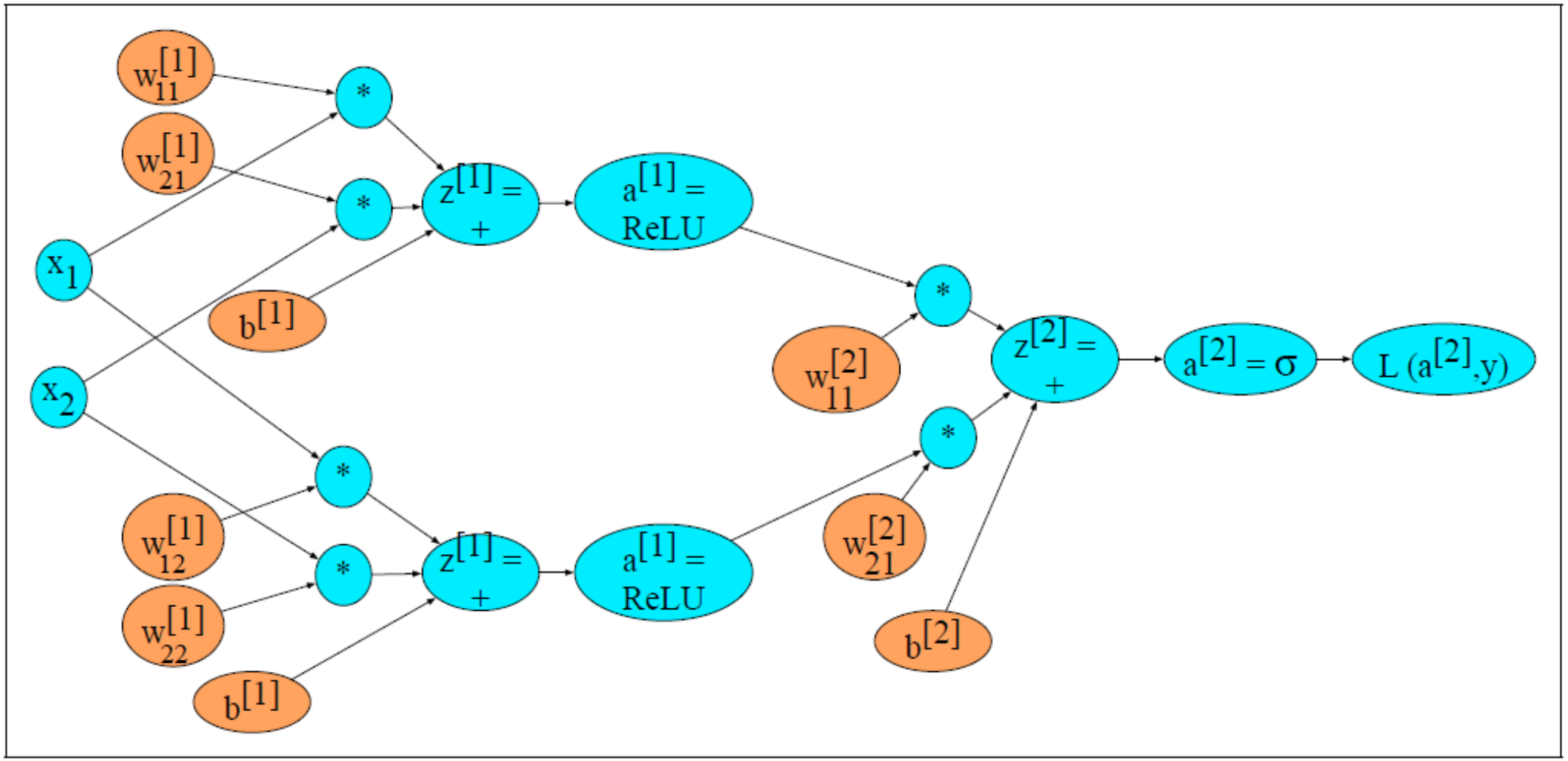


Figure 7.11 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input dimensions and 2 hidden dimensions.

Links about Backpropagation

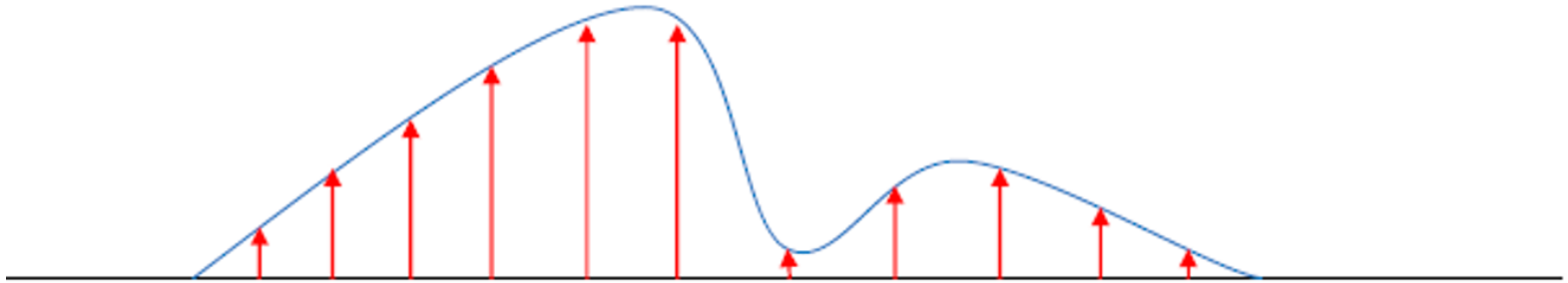
- http://colah.github.io/posts/2015-08-Backprop/
- https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b
- https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/
- http://staff.itee.uq.edu.au/janetw/cmc/chapters/BackProp/index2.html
- https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c
- https://stats.stackexchange.com/questions/235528/backpropagation-with-softmax-cross-entropy
- http://neuralnetworksanddeeplearning.com/chap2.html
- https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/
- https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Beyond training

- Hyperparameters
  - Number of layers
  - Activation functions
  - Overall architecture
  - Number of epochs
  - Learning rate
  - Regularization
  - Input features
- Suggestion
  - Start from a working model, then adapt it to your task

Deep Learning

- Using multiple layers allows for multi-step computations.
- However, since one hidden layer allows for the computation of any function, why do we need multiple layers?
- The universal approximator principle is consistent with a network in which some neurons only activate on specific inputs



Deep Learning

- A look-up table is not *efficient*. For some functions, it would require an exponential number of hidden units.
- The parity function is a good example. The input is *n* bits, of which some are activated. The output is the parity (even/odd) of the number of activated inputs (Minsky and Papert 1969).
- “If we cannot represent patterns in a compact way, we have to memorize them. This would require an exponential number of training examples.