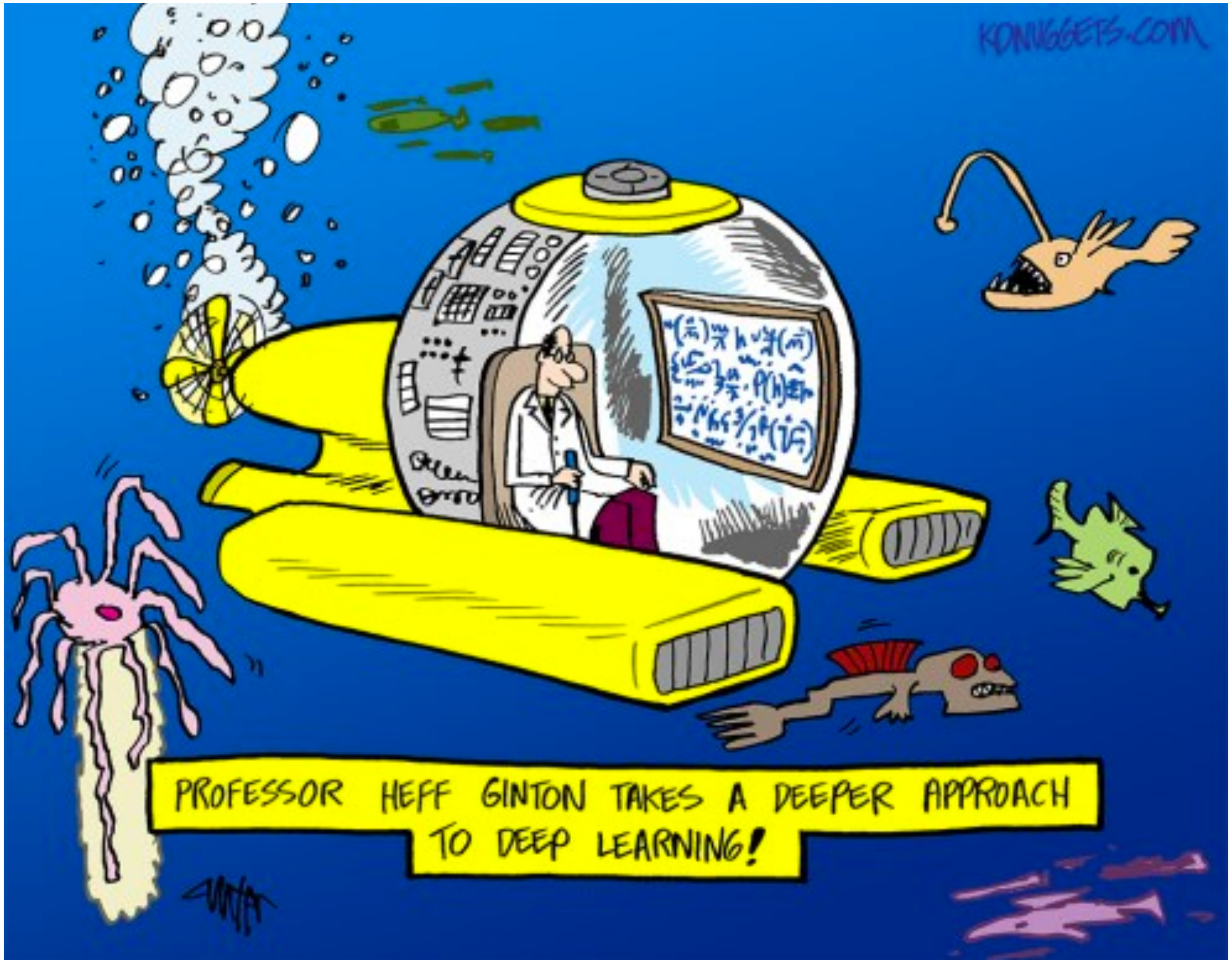
DEEP NETWORKS
10-405

Where we're going

- Assignment out Wed:
  - build framework for ANNs that will automatically differentiate and optimize any architecture
- Outline
  - History
  - Motivation
    - for ANN framework based on autodiff and matrix operations
  - Backprop 101
  - Autodiff 101

DEEP LEARNING AND NEURAL NETWORKS: BACKGROUND AND HISTORY



On-line Resources

- http://neuralnetworksanddeeplearning.com/index.html Online book by Michael Nielsen
- http://matlabtricks.com/post-5/3x3-convolution-kernels-with-online-demo - of convolutions
- https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html - demo of CNN
- http://scs.ryerson.ca/~aharley/vis/conv/ - 3D visualization
- http://cs231n.github.io/ Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

- http://www.deeplearningbook.org/ MIT Press book in prep from Bengio

A history of neural networks

- 1940s-60's:
  - McCulloch & Pitts; Hebb: modeling real neurons
  - Rosenblatt, Widrow-Hoff: : perceptrons
  - 1969: Minskey & Papert, *Perceptrons* book showed formal limitations of one-layer linear network
- 1970's-mid-1980's: …
- mid-1980's – mid-1990's:
  - backprop and multi-layer networks
  - Rumelhart and McClelland *PDP* book set
  - Sejnowski's NETTalk, BP-based text-to-speech
  - Neural Info Processing Systems (NIPS) conference starts
- Mid 1990's-early 2000's: …
- Mid-2000's to current:
  - More and more interest and experimental success

Recent history of neural networks

- Mid-2000's to current:
  - Convolutional neural nets (CNN) trained to classify large image collections (e.g., ImageNet) become widely used in computer vision
    - as representation of images
  - Word embeddings (word2vec, GloVE,…) and recurrent neural networks (RNNs – like LSTMs, GRUs, …) become widely used in NLP tasks
    - as representation of text
  - Generative adversarial networks (GANs) and variational autoencoders (VAEs)
    - as representation of distributions of images
  - …
- Progress in
  - **Hardware platforms**: GPUs
  - **Optimization**: minibatch SGD (and ADAM, RMSProp, …) with GPUs
  - **Experience**: which NN architectures work (CNNs, LSTM, …)
  - **Software platforms**: easily combine NN components

TECHNOLOGY

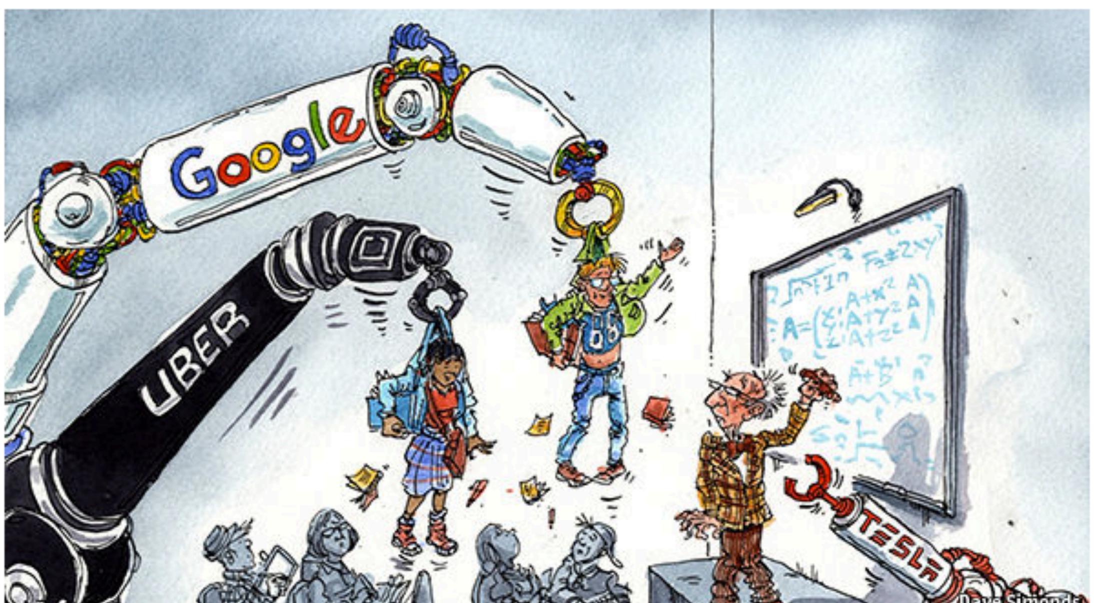
Silicon Valley Looks to Artificial Intelligence for the Next Big Thing

Artificial intelligence

Million-dollar babies

As Silicon Valley fights for talent, universities struggle to hold on to their stars

Apr 2nd 2016 | SAN FRANCISCO | From the print edition



TECHNOLOGY FEATURE · 20 FEBRUARY 2018 · CORRECTION 07 MARCH 2018

Deep learning for biology

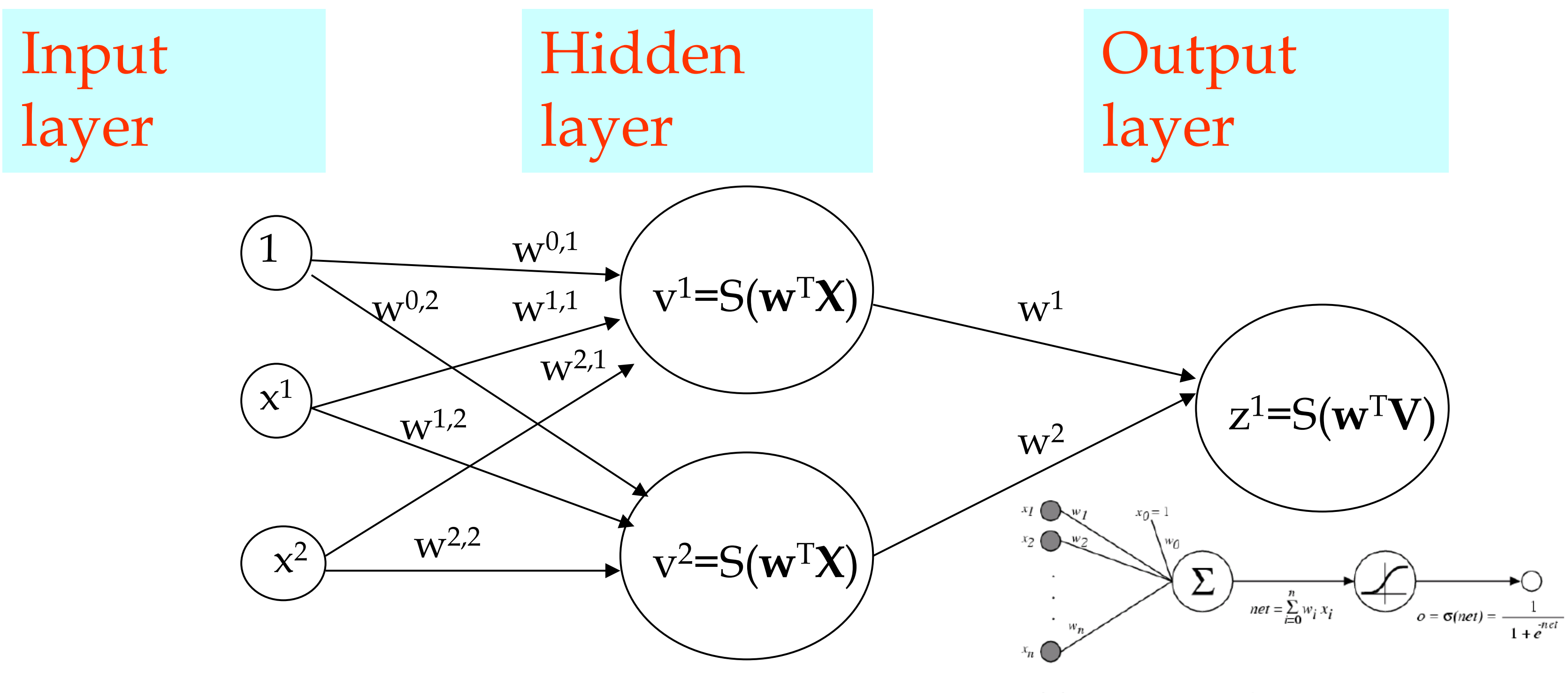
A popular artificial-intelligence method provides a powerful tool for surveying and classifying biological data. But for the uninitiated, the technology poses significant difficulties.

A Hippocratic Oath for artificial intelligence practitioners

Oren Etzioni @etzioni / Yesterday Comment

1990s Multilayer NN

- Simplest case: classifier is a multilayer *network* of *logistic units*
- Each *unit* takes some inputs and produces one output using a logistic classifier
- Output of one unit can be the input of another

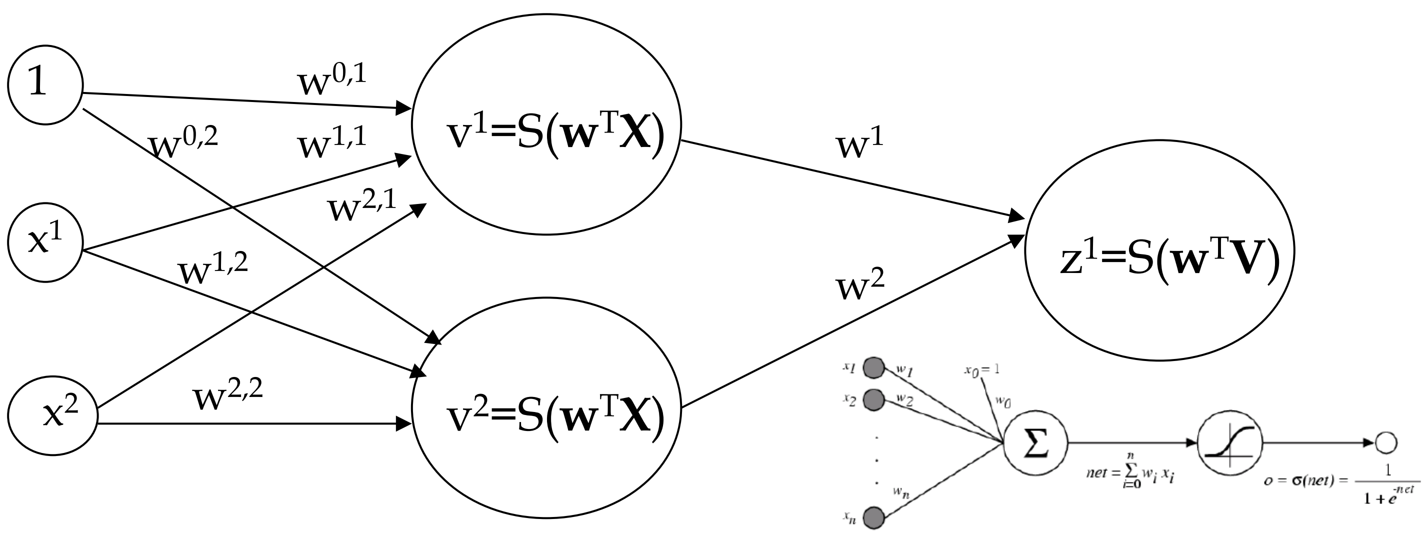


1990s Learning for NNs

- Define a loss (simplest case: squared error)
  - But over a network of "units"
- Minimize loss with gradient descent

$$J_{\mathbf{X},\mathbf{y}}(\mathbf{w}) = \sum_i \left(y^i - \hat{y}^i\right)^2$$

  - You can do this over complex networks if you can take the *gradient* of each unit: every computation is *differentiable*



1990s Learning for NNs

- Mostly 2-layer networks or else carefully constructed "deep" networks (eg CNNs)
- Worked well but training was slow and finicky

Nov 1998 – Yann LeCunn, Bottou, Bengio, Haffner

PROC. OF THE IEEE, NOVEMBER 1998 7

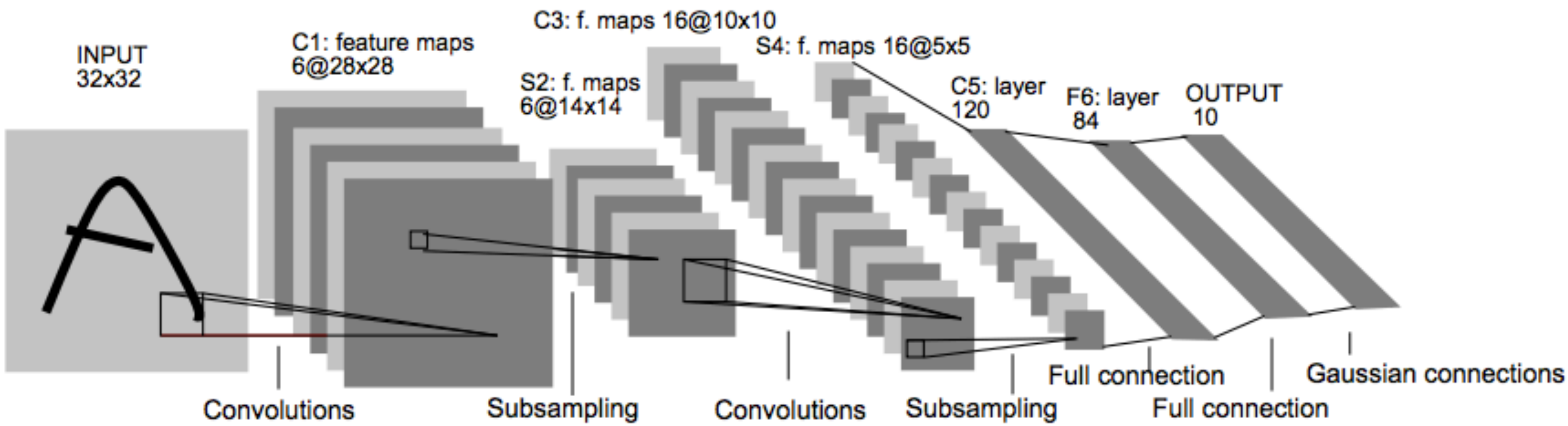
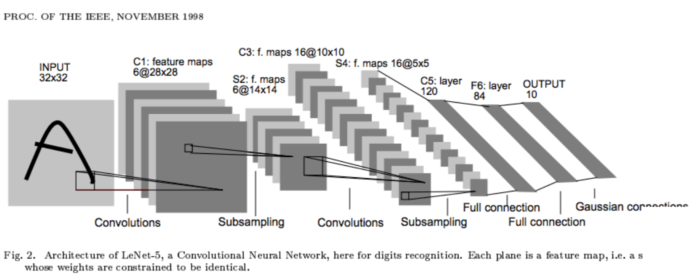


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

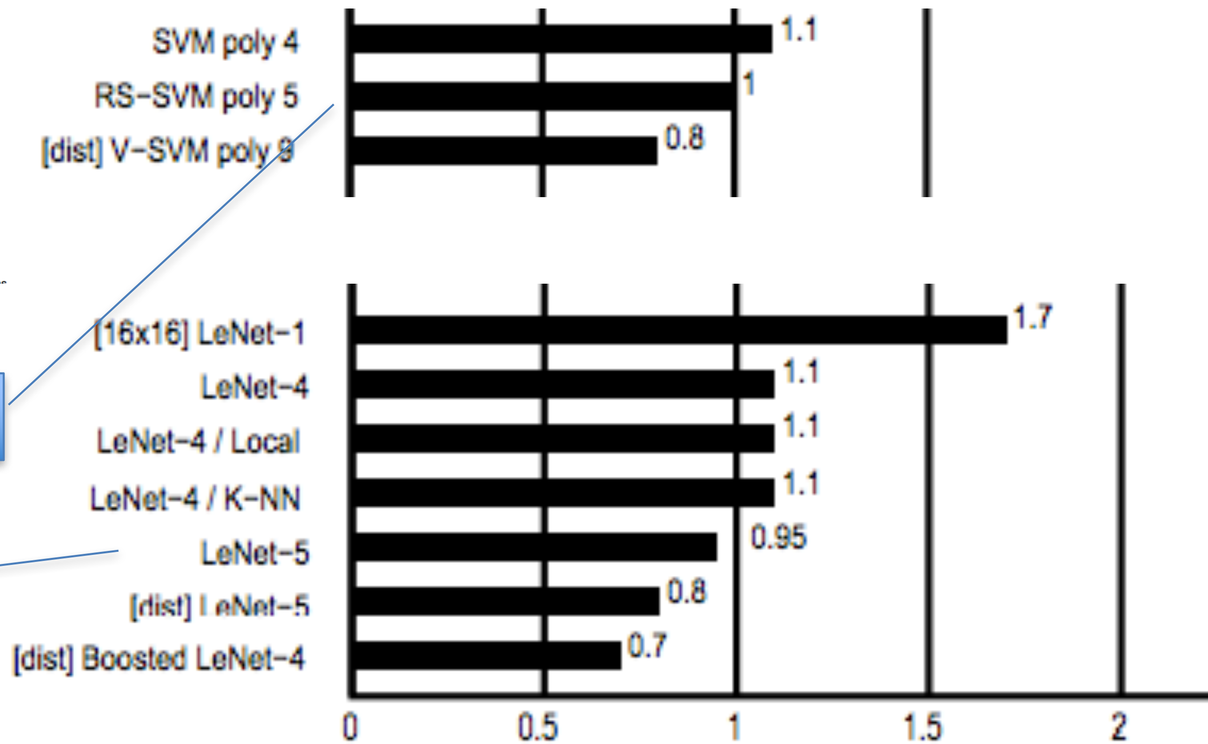
1990s Learning for NNs

- Mostly 2-layer networks or else carefully constructed "deep" networks
- Worked well but training typically took weeks



SVM: 98.9-99.2% accurate

CNNs: 98.3-99.3% accurate



1990s Teaching Learning for NNs

For nodes *k* in output layer:

$$\delta_k \equiv (t_k - a_k)\ a_k(1 - a_k)$$

For nodes *j* in hidden layer:

$$\delta_j \equiv \sum_k (\delta_k w_{kj})\ a_j(1 - a_j)$$

For all weights:

$$w_{kj} = w_{kj} - \varepsilon\ \delta_k a_j$$

$$w_{ji} = w_{ji} - \varepsilon\ \delta_j a_i$$

"Propagate errors backward"
BACKPROP

Can carry this recursion out further if you have multiple hidden layers

2018 Learning for NNs

- We need to understand **interaction of**: hardware platforms, software platforms, architectural components, optimization methods
- Start off with a **new high-level language for NNs**
  - vectors/matrices/tensors
    - tensor = k-dimensional array of floats
  - vector/matrix/tensor operations
  - built-in gradient computation and optimizers
  - architectural components as subroutines

- A lot like dataflow languages for map-reduce workflows (eg GuineaPig)

Vectorizing logistic regression

(review)

Vectorized minibatch logistic regression

- Computation we'd like to vectorize:
  - For each **x** in the minibatch, compute

$$p \equiv \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}} = \frac{1}{1+\exp(-\sum_j x^j w^j)}$$

    - For each feature *j*: update w^j using

$$\frac{\partial}{\partial w^j} \log P(Y=y|X=\mathbf{x},\mathbf{w}) = (y-p)x^j$$

Vectorizing logistic regression

- Computation we'd like to parallelize:
  - For each **x** in the minibatch X_{batch}, compute

$$p \equiv \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}} = \frac{1}{1+\exp(-\sum_j x^j w^j)}$$

$$\boldsymbol{X}_{batch}\boldsymbol{w} = \begin{bmatrix} x_1^1 & \cdots & x_1^J \\ \vdots & \ddots & \vdots \\ x_B^1 & \cdots & x_B^J \end{bmatrix} \begin{bmatrix} w^1 \\ \vdots \\ w^J \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}\cdot\boldsymbol{x_1} \\ \vdots \\ \boldsymbol{w}\cdot\boldsymbol{x_B} \end{bmatrix}$$

Vectorizing logistic regression

- Computation we'd like to parallelize:
  - For each **x** in the minibatch X_{batch}, compute

$$p \equiv \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}} = \frac{1}{1+\exp(-\sum_j x^j w^j)}$$

$$\begin{bmatrix} \boldsymbol{w}\cdot\boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{w}\cdot\boldsymbol{x}_B \end{bmatrix} + 1$$

in numpy if M is a matrix
M+1 does the "right thing"

so does np.exp(M)

Vectorizing logistic regression

- Computation we'd like to parallelize:
  - For each **x** in the minibatch, compute

$$p \equiv \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}} = \frac{1}{1+\exp(-\sum_j x^j w^j)}$$

$$\frac{\partial}{\partial w^j} \log P(Y=y|X=\mathbf{x},\mathbf{w}) = (y-p)x^j$$

```
def logistic(X):  return (-X.exp()+1).reciprocal()
p = logistic(Xb.dot(w))    # B rows, 1 column
grad = Xb.dot(y – p).rowsum() * 1/B
w = w + grad*rate
```

Binary to softmax logistic regression

$$p \equiv \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}} = \frac{1}{1+\exp(-\sum_j x^j w^j)}$$

$$X_{batch}\boldsymbol{w} = \begin{bmatrix} x_1^1 & \cdots & x_1^J \\ \vdots & \ddots & \vdots \\ x_B^1 & \cdots & x_B^J \end{bmatrix} \begin{bmatrix} w^1 \\ \vdots \\ w^J \end{bmatrix} = \begin{bmatrix} \boldsymbol{w} \cdot \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{w} \cdot \boldsymbol{x_B} \end{bmatrix}$$

Binary to softmax logistic regression

$$p \equiv \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}} = \frac{1}{1+\exp(-\sum_j x^j w^j)}$$

$$p^y \equiv \frac{\exp(\boldsymbol{x}\cdot\boldsymbol{w}^y)}{\sum_{y'}\exp(\boldsymbol{x}\cdot\boldsymbol{w}^{y'})}$$

$$XW = \begin{bmatrix} x_1^1 & \cdots & x_1^J \\ \vdots & \ddots & \vdots \\ x_B^1 & \cdots & x_B^J \end{bmatrix} \begin{bmatrix} w^1 \\ \vdots \\ w^J \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}\cdot\boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{w}\cdot\boldsymbol{x}_B \end{bmatrix}$$

$$XW = \begin{bmatrix} x_1^1 & \cdots & x_1^J \\ \vdots & \ddots & \vdots \\ x_B^1 & \cdots & x_B^J \end{bmatrix} \begin{bmatrix} w_1^{y1} & \dots & w_1^{yK} \\ \vdots & \ddots & \vdots \\ w_J^{y1} & \dots & w_J^{yK} \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}^{y1}\cdot\boldsymbol{x}_1 & \dots & \boldsymbol{w}^{yK}\cdot\boldsymbol{x}_1 \\ \vdots & \ddots & \vdots \\ \boldsymbol{w}^{y1}\cdot\boldsymbol{x}_B & \dots & \boldsymbol{w}^{yK}\cdot\boldsymbol{x}_B \end{bmatrix}$$

http://minpy.readthedocs.io/en/latest/get-started/logistic_regression.html

```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster

# Predict the class using multinomial logistic regr
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob
```

Matrix multiply,; then exponentiate component-wise

prob will have B rows and K columns, and each row will sum to 1

Sum the columns to get the denominator; keepdim=True means…

… that this line will work correctly even though 'a' and 'a_sum' have different shapes

$$p^y \equiv \frac{\exp(\boldsymbol{x} \cdot \boldsymbol{w}^y)}{\sum_{y'} \exp(\boldsymbol{x} \cdot \boldsymbol{w}^{y'})}$$

$$XW = \begin{bmatrix} x_1^1 & \cdots & x_1^J \\ \vdots & \ddots & \vdots \\ x_B^1 & \cdots & x_B^J \end{bmatrix} \begin{bmatrix} w_1^{y1} & \dots & w_1^{yK} \\ \vdots & \ddots & \vdots \\ w_J^{y1} & \dots & w_J^{yK} \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}^{y1} \cdot \boldsymbol{x}_1 & \dots & \boldsymbol{w}^{yK} \cdot \boldsymbol{x}_1 \\ \vdots & \ddots & \vdots \\ \boldsymbol{w}^{y1} \cdot \boldsymbol{x}_B & \dots & \boldsymbol{w}^{yK} \cdot \boldsymbol{x}_B \end{bmatrix}$$

http://minpy.readthedocs.io/en/latest/get-started/logistic_regression.html

```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```

http://minpy.readthedocs.io/en/latest/get-started/logistic_regression.html

```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```

http://minpy.readthedocs.io/en/latest/get-started/logistic_regression.html

```
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluste

# Predict the class using multinomial logistic regr
```

$$x.T\,dy = \begin{bmatrix} x_1^1 & \cdots & x_B^1 \\ \vdots & \ddots & \vdots \\ x_1^J & \cdots & x_B^J \end{bmatrix} \cdot \begin{bmatrix} dy_{x1}^{y1} & \dots & dy_{x1}^{yK} \\ \vdots & \ddots & \vdots \\ dy_{xB}^{y1} & \dots & dy_{xB}^{yK} \end{bmatrix}$$

Error on each example **x** in batch and each class y

```
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_sam
```

```
# gradient descent
dy = prob - label
dw = np.dot(data.T, dy) / num_samples
# update parameters; fixed learning rate
w -= 0.1 * dw
```

python bug: should be x.T (transpose)

The gradient step!

```
# initialize training data.
```

$$\frac{\partial}{\partial w^j} \log P(Y = y | X = \mathbf{x}, \mathbf{w}) = (y - p) x^j$$

```
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```

PARALLEL TRAINING FOR ANNS

How are ANNs trained?

- Typically, with some variant of streaming SGD
  - Keep the data on disk, in a preprocessed form
  - Loop over it multiple times
  - Keep the model in memory

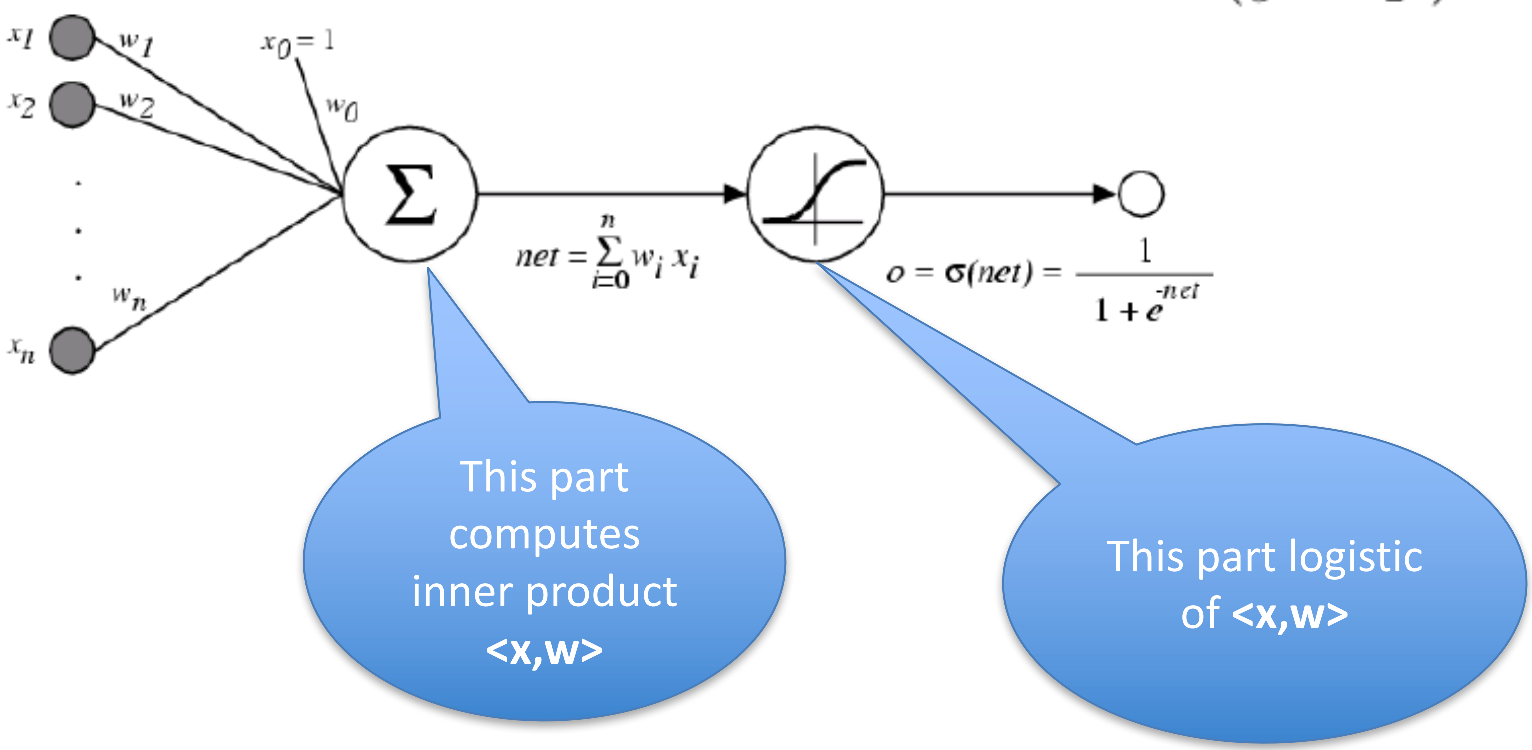
- Solution to big data: but long training times!

- However, *some* parallelism is often used….

Recap: logistic regression with SGD

$$P(Y=1 \mid X=\mathbf{x}) = p = \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}}$$

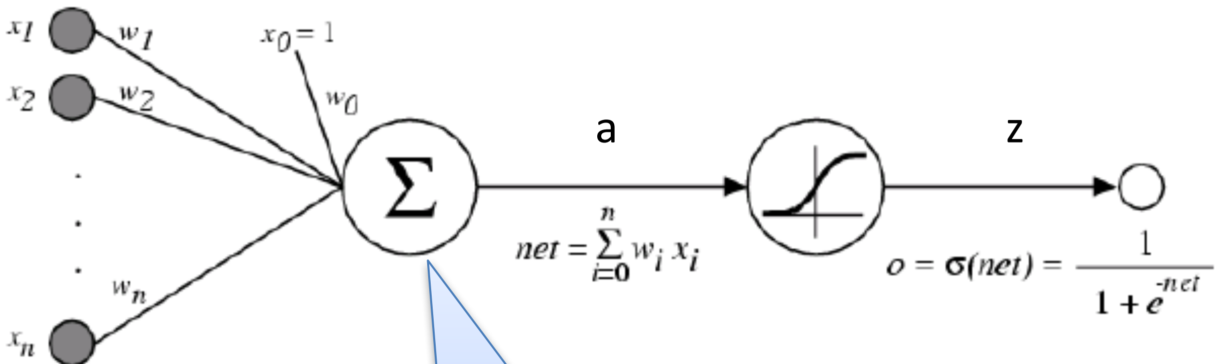
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \lambda(y-p)\mathbf{x}$$



Recap: logistic regression with SGD

$$P(Y=1 \mid X=\mathbf{x}) = p = \frac{1}{1+e^{-\mathbf{x}\cdot\mathbf{w}}}$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \lambda(y-p)\mathbf{x}$$

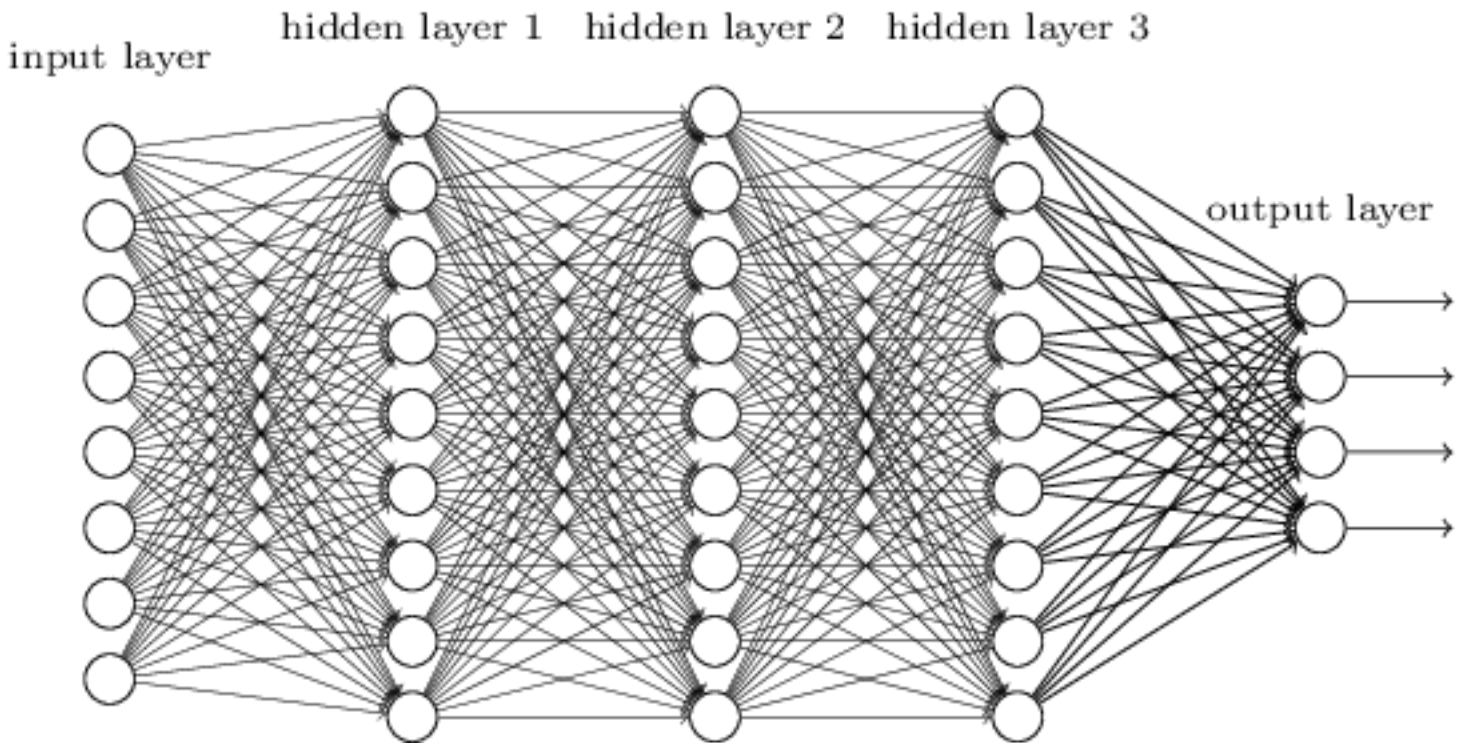


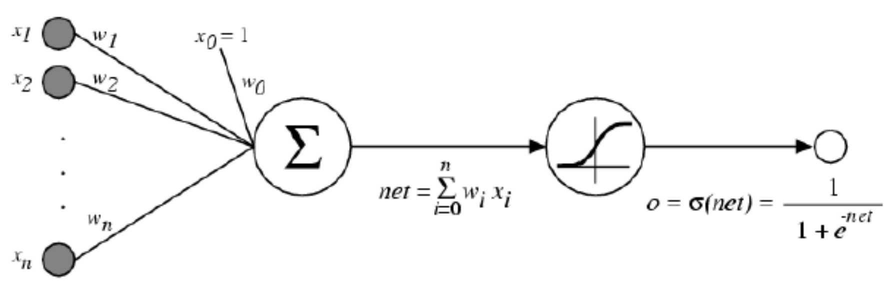
On one example: computes inner product **<x,w>**

$$\sum_j x^j w^j$$

There’s some chance to compute this in parallel…can we do more?

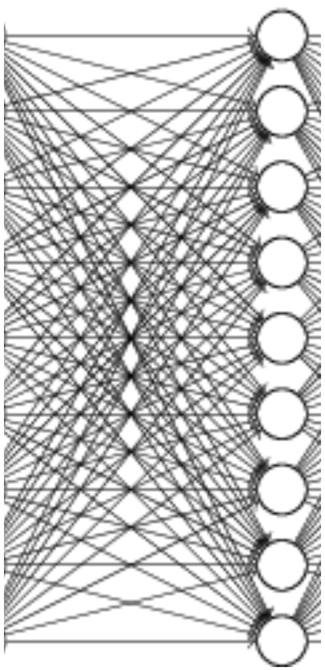
In ANNs we have many many logistic regression nodes

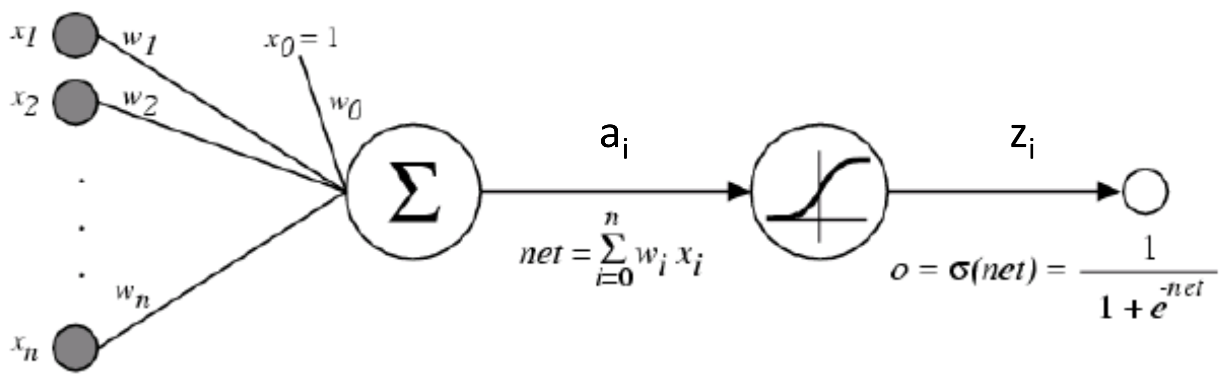




Recap: logistic regression with SGD

Let **x** be an example
Let $\mathbf{w}_i$ be the input weights for the i-th hidden unit
Then output $\mathbf{a_i} = \mathbf{x} \cdot \mathbf{w}_i$



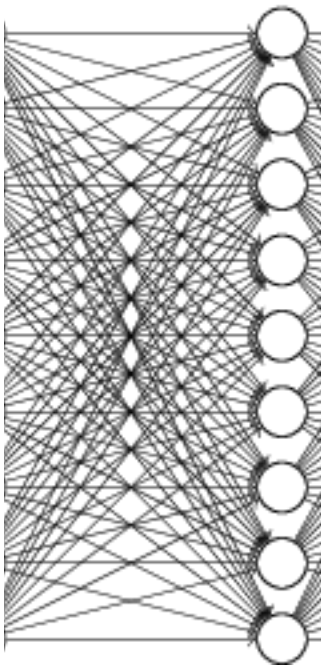


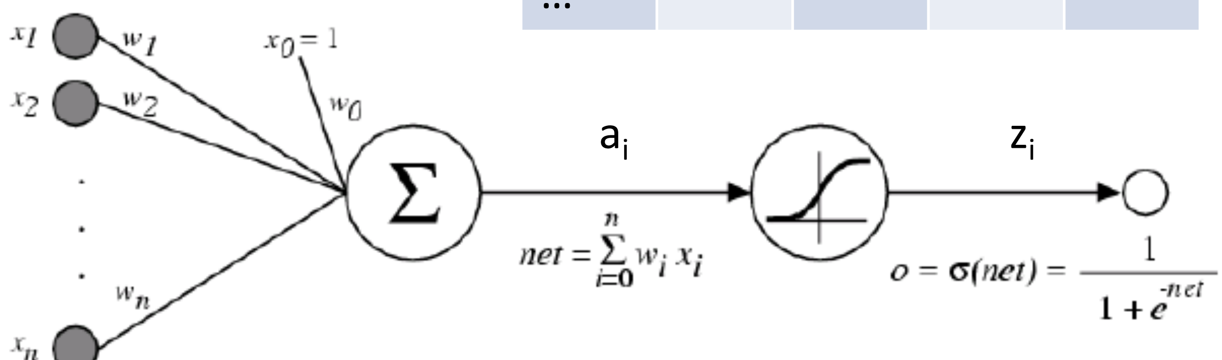
Recap: logistic regression with SGD

Let **x** be an example
Let $\mathbf{w}_i$ be the input weights for the i-th hidden unit
Then **a** = **x** W
is output for all *m* units

W =

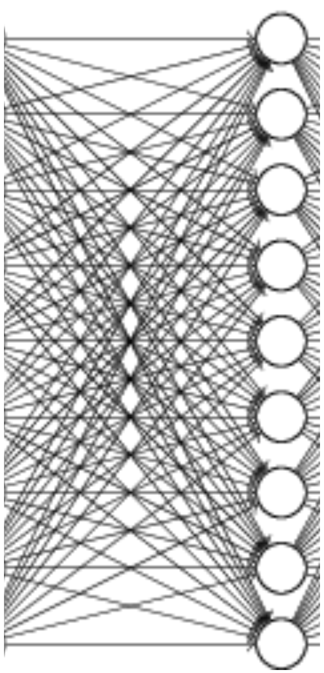
w_1	w_2	w_3	…	w_m
0.1	-0.3	…		
-1.7	…			
..				
…				





Recap: logistic regression with SGD

Let X be a matrix with *k* examples
Let $\mathbf{w}_i$ be the input weights for the i-th hidden unit
Then A = X W is output for all *m* units
for all *k* examples



$\mathbf{x}_1$	1	0	1	1
$\mathbf{x}_2$	…			
…				
$\mathbf{x}_k$				

w_1	w_2	w_3	…	w_m
0.1	-0.3	…		
-1.7	…			
0.3	…			
1.2				

XW =

$\mathbf{x}_1.\mathbf{w}_1$	$\mathbf{x}_1.\mathbf{w}_2$	…	$\mathbf{x}_1.\mathbf{w}_m$
			$\mathbf{x}_k.\mathbf{w}_m$

There's a lot of chances to do this in parallel

Minibatch SGD: batch size trades off parallellism vs memory

ANNs and multicore CPUs

- Modern libraries (Matlab, numpy, …) do matrix operations fast, in parallel
- Many ANN implementations exploit this parallelism automatically
- Key implementation issue is working with matrices comfortably

ANNs and GPUs

- GPUs do matrix operations very fast, in parallel
  - For dense matrixes, not sparse ones!
- Training ANNs on GPUs is common
  - SGD and minibatch sizes of 128
- Modern ANN implementations can exploit this
- GPUs are not super-expensive
  - $500 for high-end one
  - large models with $O(10^7)$ parameters can fit in a large-memory GPU (12Gb)
- Speedups of 20x-50x are typical

ANNs and multi-GPU systems

- There are ways to set up ANN computations so that they are spread across multiple GPUs
  - Sometimes involves some sort of IPM
  - Sometimes involves partitioning the model across multiple GPUs
  - Often needed for very large networks

Where we're going

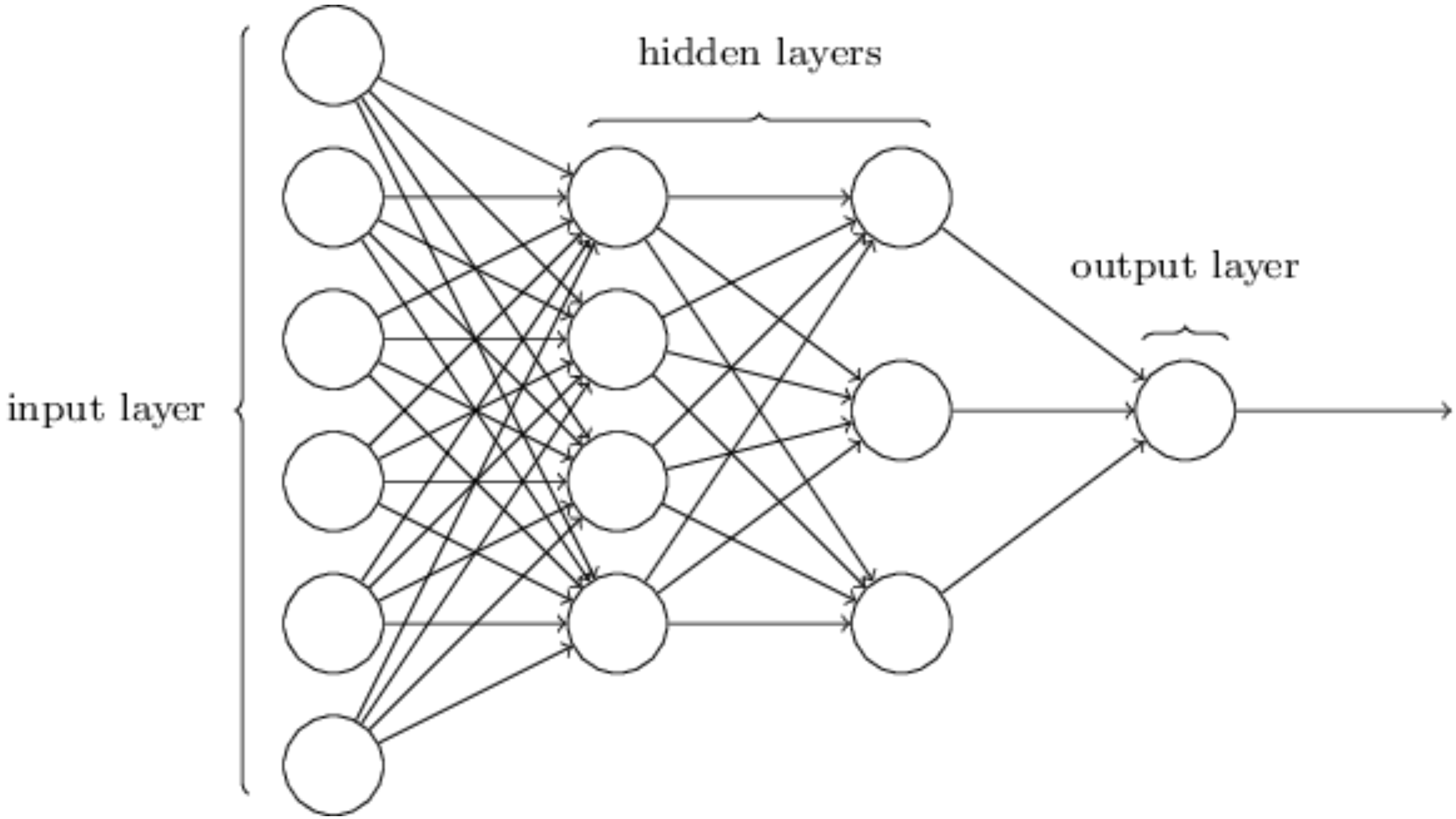
- Assignment out Wed:
  - build framework for ANNs that will automatically differentiate and optimize any architecture
- Outline
  - History
  - Motivation
    - for ANN framework based on autodiff and matrix operations
  - Backprop 101
  - Autodiff 101

Vectorizing BackProp

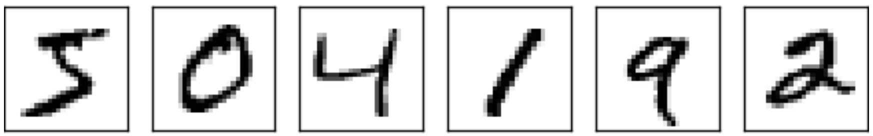
BackProp in Matrix-Vector Notation

Michael Nielson: http://neuralnetworksanddeeplearning.com/

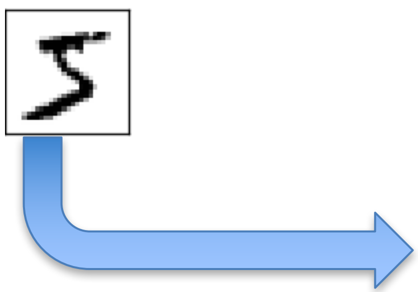
Notation

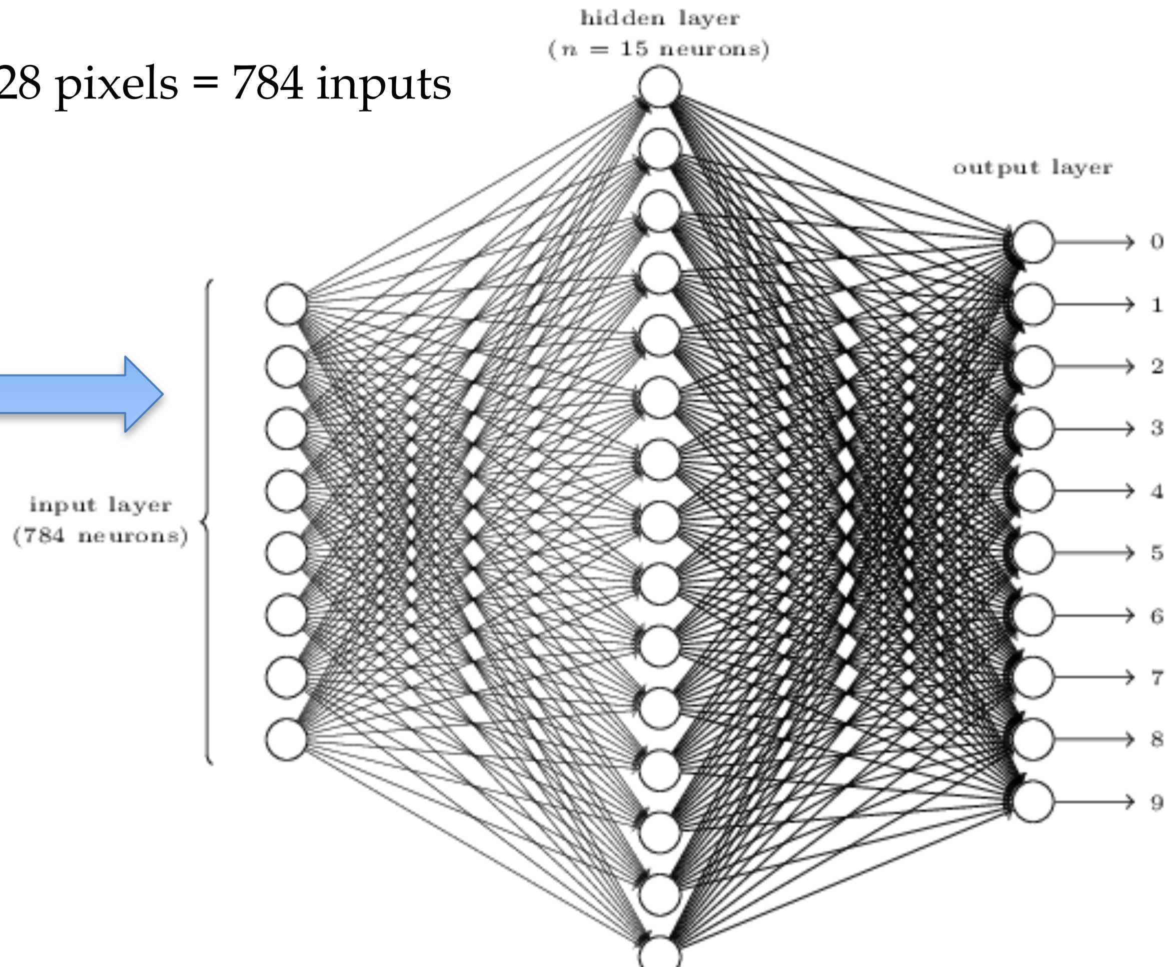


Notation

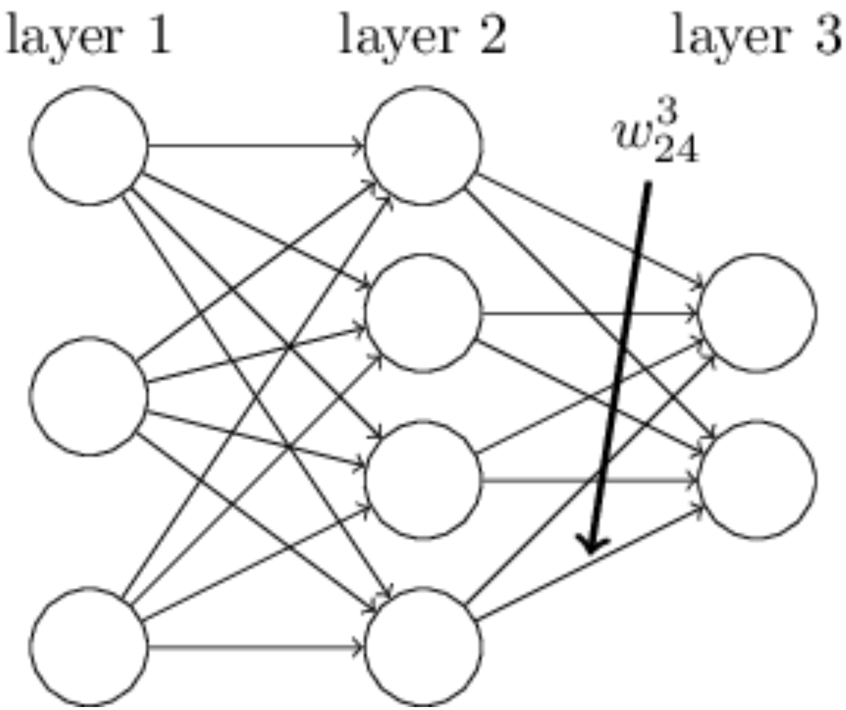


Each digit is 28x28 pixels = 784 inputs





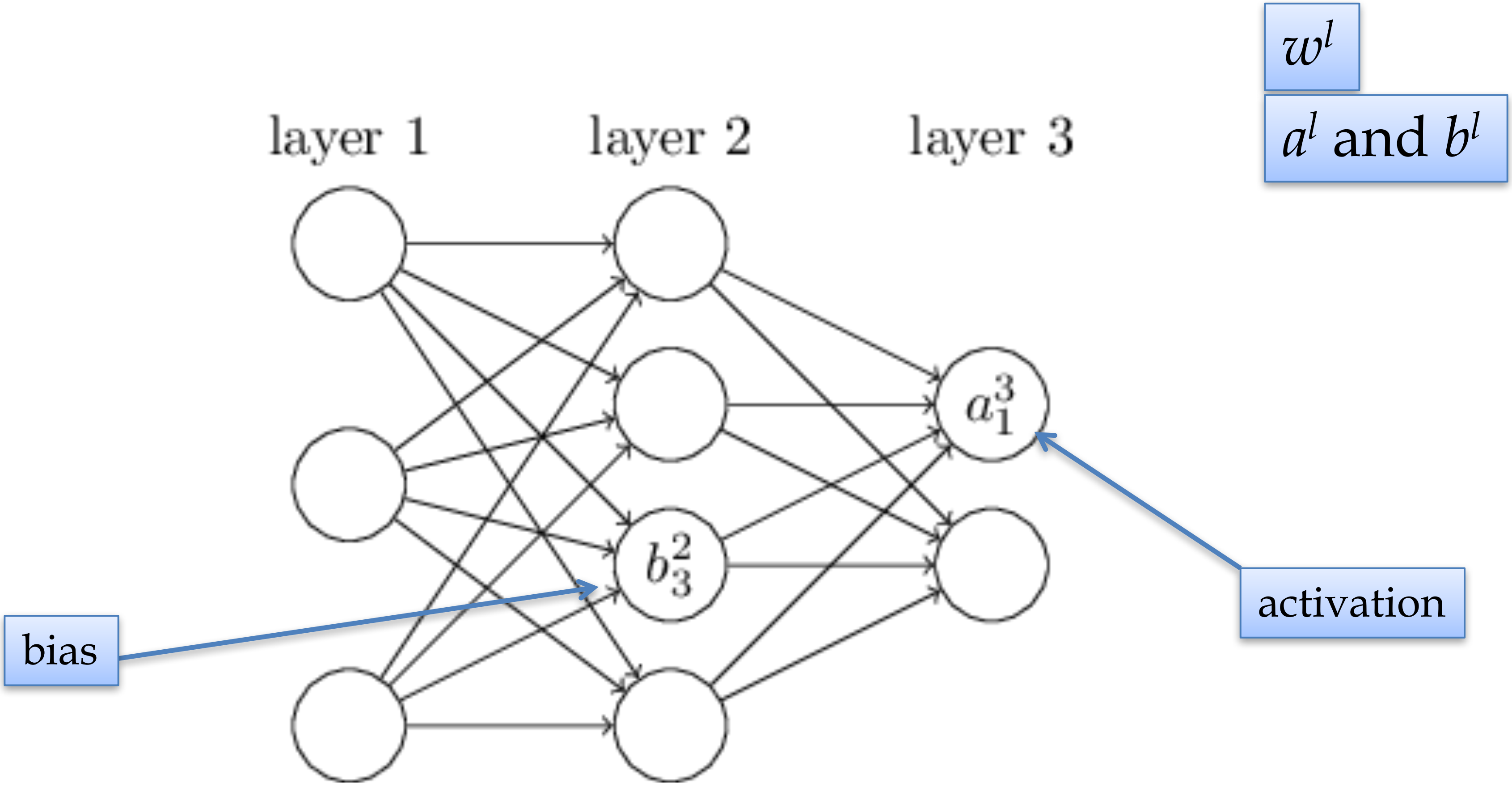
Notation



w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

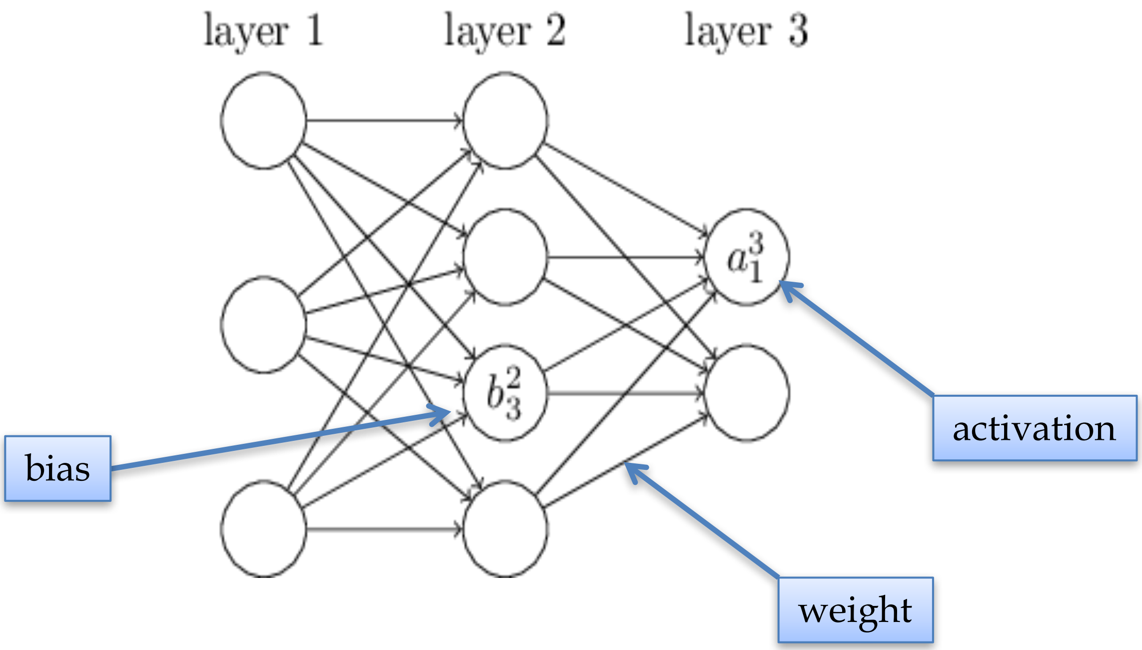
w^l is weight matrix for layer l

Notation



$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

Notation



Matrix: w^l
Vector: a^l
Vector: b^l

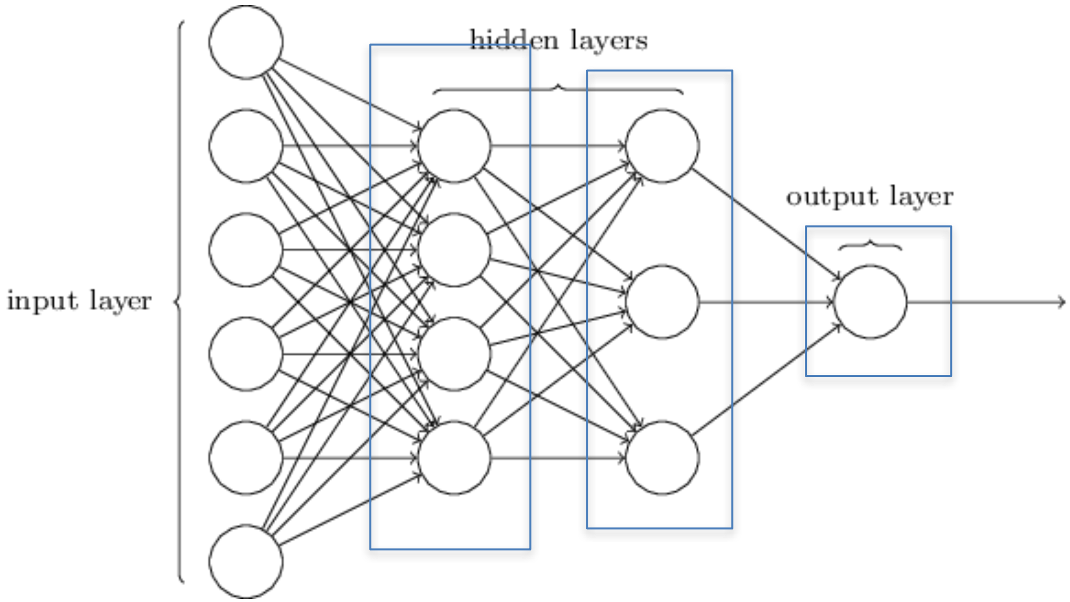
Vector: z^l

vector→vector function: componentwise logistic

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \Rightarrow a^l = \sigma(w^l a^{l-1} + b^l).$$

$$z^l \equiv w^l a^{l-1} + b^l$$

Computation is "feedforward"



for l=1, 2, … L:

$$a^l = \sigma(w^l a^{l-1} + b^l).$$

Notation

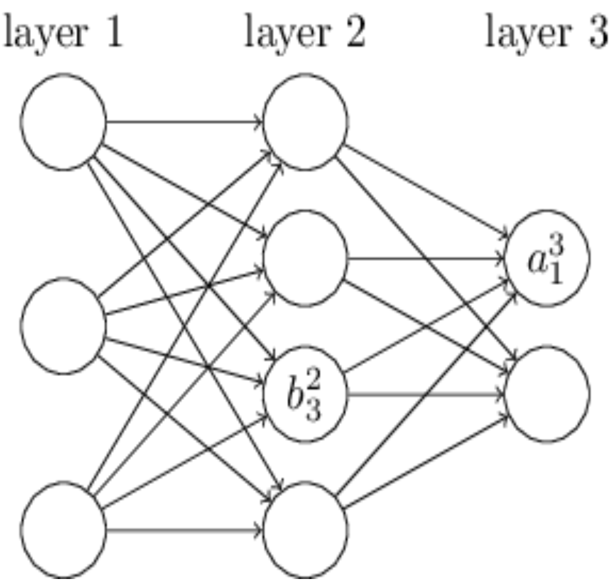
Cost function to optimize: sum over examples x

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2,$$

$$= \frac{1}{n} \sum_x C_x$$

where

$$C_x = \frac{1}{2} \|y - a^L\|^2.$$

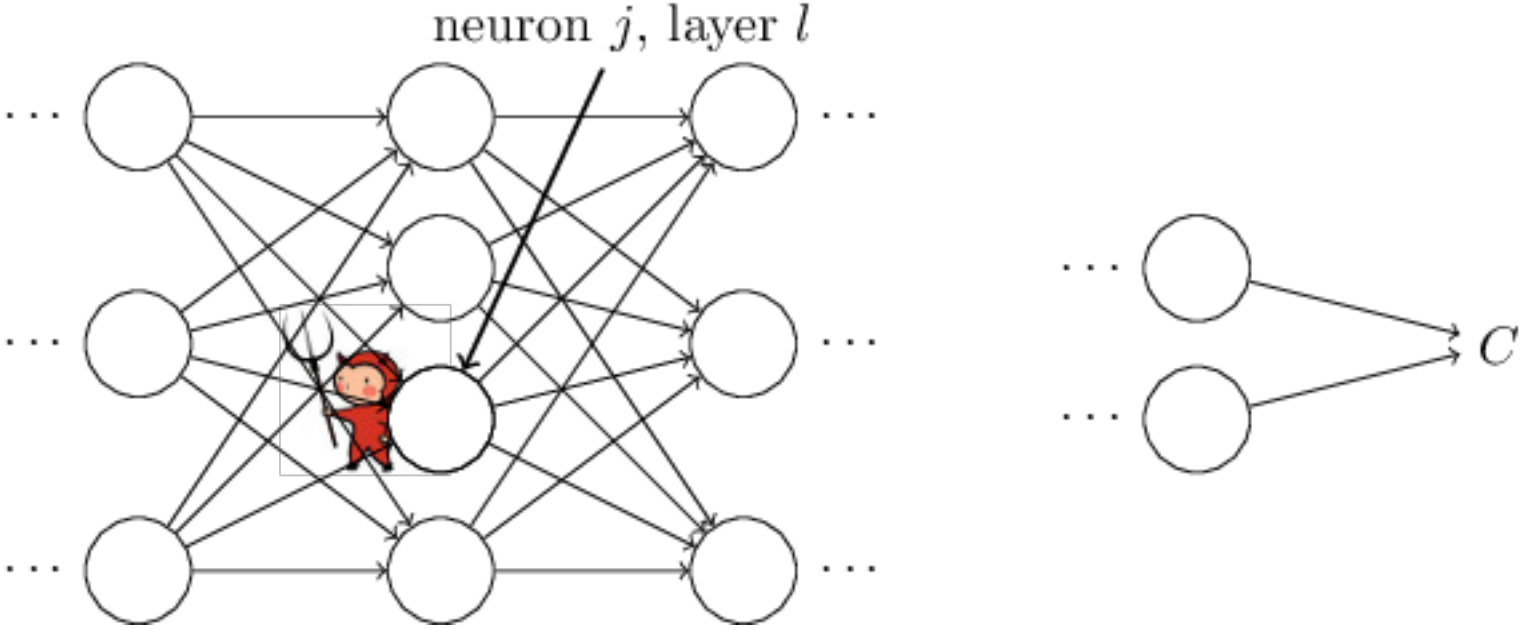


Matrix: w^l
Vector: a^l
Vector: b^l

Vector: z^l

Vector: y

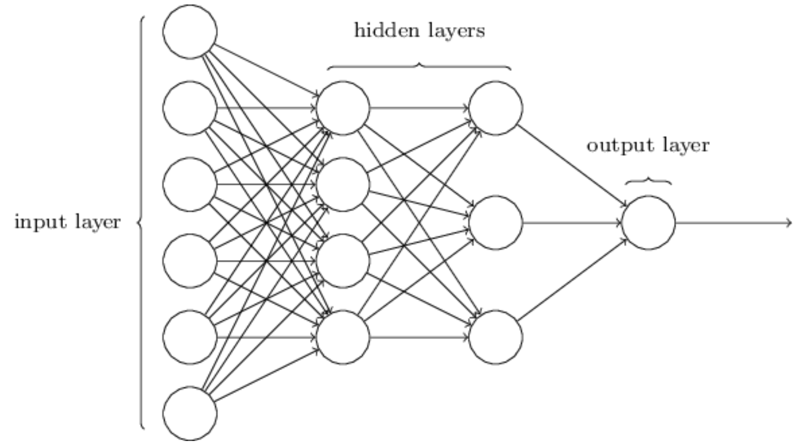
Notation



$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}.$$

BackProp: last layer

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L).$$



Matrix form:

$$\delta^L = \nabla_a C \odot \sigma'(z^L).$$

components are $\frac{\partial C}{\partial a_j^L}$

components are $\sigma'(z_j^L)$

Level l for l=1,…,L

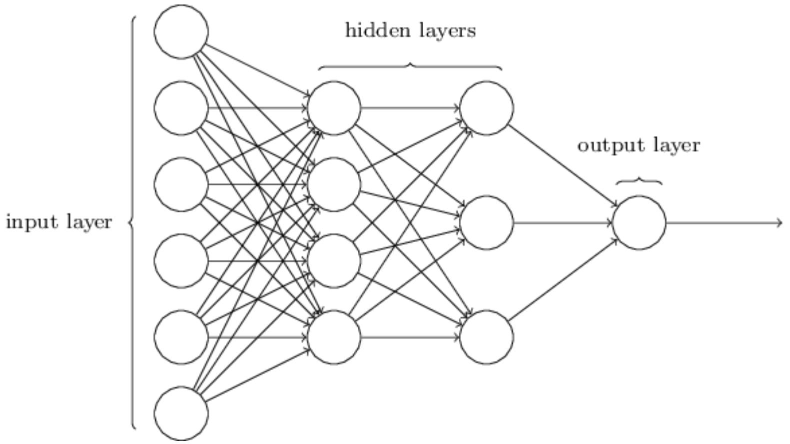
Matrix: w^l

Vectors:

- bias b^l
- activation a^l
- pre-sigmoid activ: z^l
- target output y
- "local error" δ^l

BackProp: last layer

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L).$$



Matrix form for square loss:

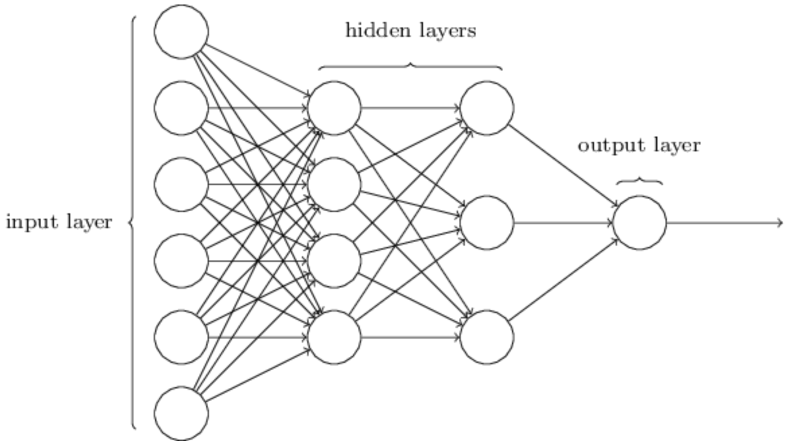
$$\delta^L = (a^L - y) \odot \sigma'(z^L).$$

Level l for l=1,…,L
Matrix: w^l
Vectors:

- bias b^l
- activation a^l
- pre-sigmoid activ: z^l
- target output y
- "local error" δ^l

BackProp: error at level *l* in terms of error at level *l+1*

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$



which we can use to compute

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \quad \Rightarrow \quad \frac{\partial C}{\partial b} = \delta$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \Rightarrow \quad \frac{\partial C}{\partial w} = a_{\rm in} \delta_{\rm out}$$

Level l for l=1,…,L
Matrix: w^l
Vectors:

- bias b^l
- activation a^l
- pre-sigmoid activ: z^l
- target output y
- "local error" δ^l

BackProp: summary

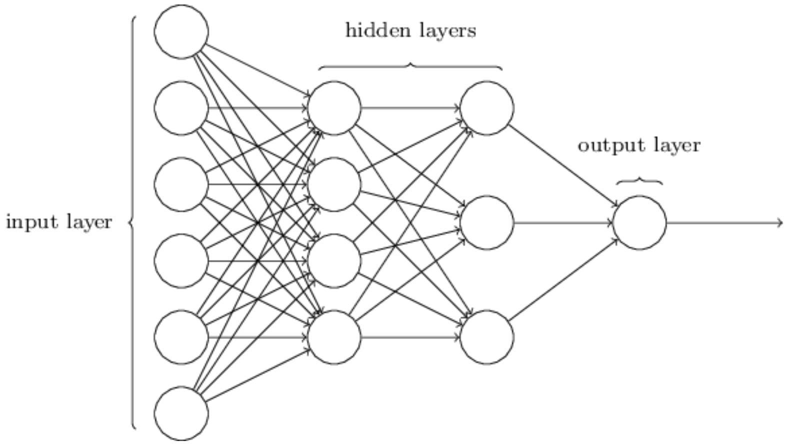
$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

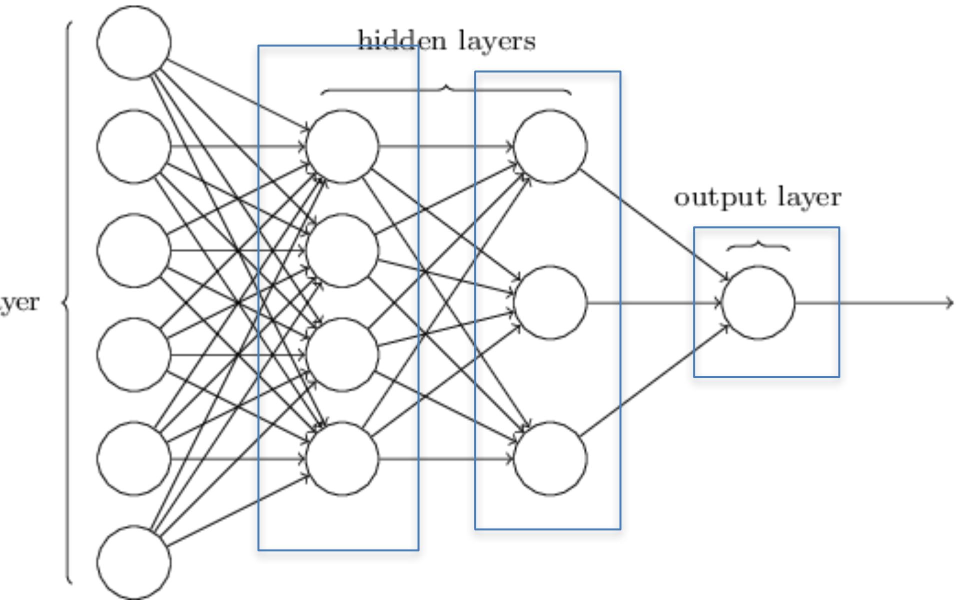
$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

for SGD updates!



Level l for l=1,…,L
Matrix: w^l
Vectors:
- bias b^l
- activation a^l
- pre-sigmoid activ: z^l
- target output y
- "local error" δ^l

Computation propagates errors backward



$$\delta^L = (a^L - y) \odot \sigma'(z^L).$$

for *l*=*L*-1,…1:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

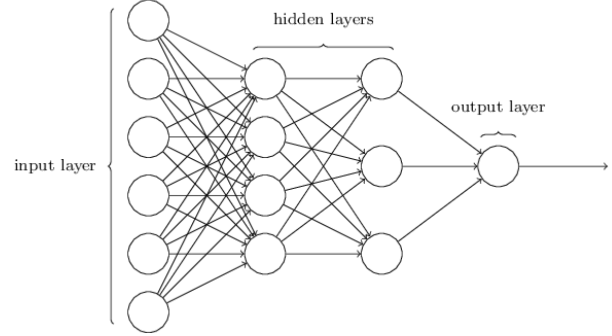
$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

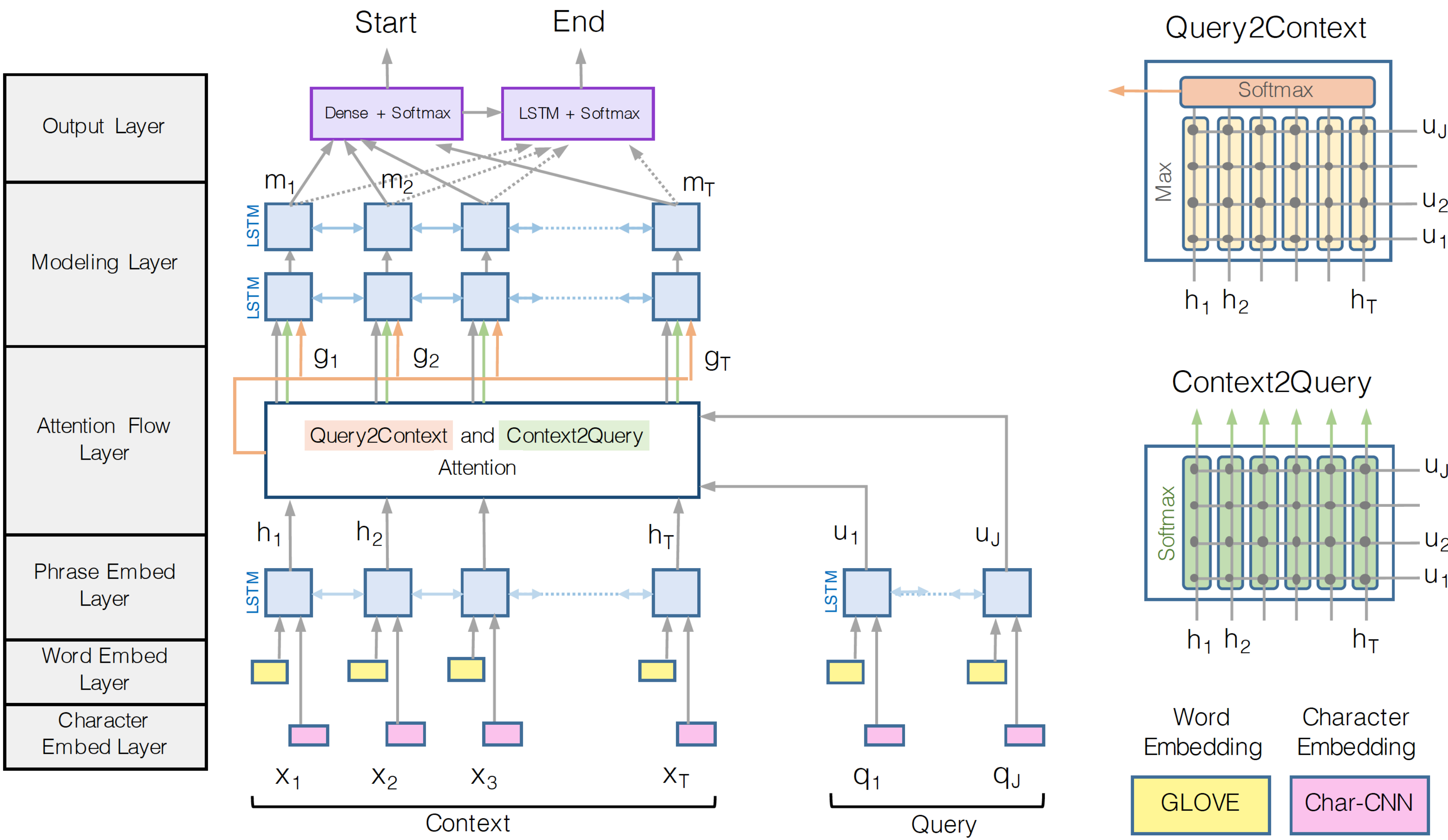
BackProp 101

- Forward pass computes *z's* and *a's*
- Backward pass uses these to compute δ's
- Simple to define each with matrix operators
  - Hence easy to run in parallel minibatch SGD process

We can build much more complex networks by combining subnetworks – sort of like subroutines



How do we derive backprop for these networks?



How can we generalize BackProp to other ANNs?

How can we automate BackProp for other ANNs?

Deep Neural Network Toolkits: What's Under the Hood?

Recap: Wordcount in GuineaPig

```
# always start like this
from guineapig import *
import sys

# supporting routines can go here
def tokens(line):
    for tok in line.split():
        yield tok.lower()

#always subclass Planner
class WordCount(Planner):

    wc = ReadLines('corpus.txt') | Flatten(by=tokens) | Group(by=lambda x:x, reducingWith=ReduceToCount())

# always end like this
if __name__ == "__main__":
    WordCount().main(sys.argv)
```

```
class WordCount(Planner):
    lines = ReadLines('corpus.txt')
    words = Flatten(lines,by=tokens)
    wordCount = Group(words, by=lambda x:x, reducingTo=ReduceToCount())
```

class variables in the planner are ***data structures***

```
wordCount = Group(words,by=<function <lambda> at
| words = Flatten(lines, by=<function tokens at 0
| | lines = ReadLines("corpus.txt")
```

Recap: Wordcount in GuineaPig

```
wordCount = Group(words,by=<function <lambda> at 0x10497aa28>,reducingTo=<guineapig.ReduceT
| words = Flatten(lines, by=<function tokens at 0x1048965f0>).opts(stored=True)
| | lines = ReadLines("corpus.txt")
```

The general idea:

- Embed something that looks like code but, when executed, builds a data structure
- The data structure defines a computation you want to do
  - "computation graph"
- Then you use the data structure to do the computation
  - stream-and-sort
  - streaming Hadoop
  - …

- We're going to re-use the same idea: but now the graph both supports **computation** of a function and **differentiation** of that computation

$$f(x_1, x_2) \equiv (2x_1 + x_2)^2$$

$$\begin{aligned} z_1 &= \texttt{add}(x_1, x_1) \\ z_2 &= \texttt{add}(z_1, x_2) \\ f &= \texttt{square}(z_2) \end{aligned}$$

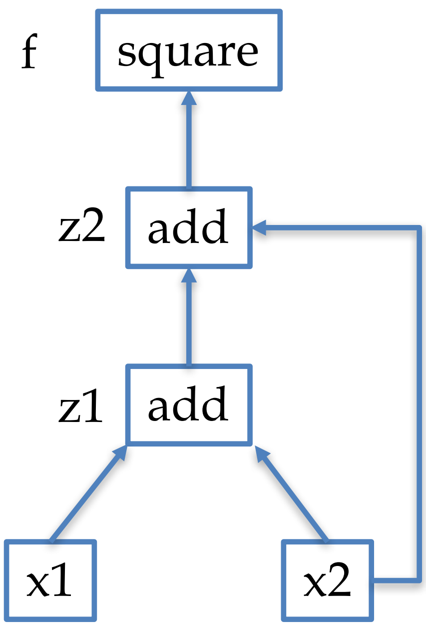
computation graph, aka tape, aka Wengert list

$$\begin{aligned} f(x_1, x_2) &= (2x_1 + x_2)^2 = 4x_1^2 + 4x_1x_2 + x_2^2 \\ \frac{df}{dx_1} &= 8x_1 + 4x_2 \\ \frac{df}{dx_2} &= 4x_1 + 2x_2 \end{aligned}$$

$$f(x_1, x_2) \equiv (2x_1 + x_2)^2$$

$$\begin{aligned} z_1 &= \mathtt{add}(x_1, x_1) \\ z_2 &= \mathtt{add}(z_1, x_2) \\ f &= \mathtt{square}(z_2) \end{aligned}$$

computation graph



$$f(x_1, x_2) \equiv (2x_1 + x_2)^2$$

$$\begin{aligned} z_1 &= \texttt{add}(x_1, x_1) \\ z_2 &= \texttt{add}(z_1, x_2) \\ f &= \texttt{square}(z_2) \end{aligned}$$

An algorithm to evaluate f' at fixed x1=c1,x2=c2

Definition
AP calculus rules

Derivation Step	Reason
$\frac{df}{dx_1} = \frac{dz_2^2}{dz_2} \cdot \frac{dz_2}{dx_1}$	$f = z_2^2$
$\frac{df}{dx_1} = 2z_2 \cdot \frac{dz_2}{dx_1}$	$\frac{d(a^2)}{da} = 2a$
$\frac{df}{dx_1} = 2z_2 \cdot \frac{d(z_1+x_2)}{dx_1}$	$z_2 = z_1 + x_2$
$\frac{df}{dx_1} = 2z_2 \cdot \left(1 \cdot \frac{dz_1}{dx_1} + 1 \cdot \frac{dx_2}{dx_1}\right)$	$\frac{d(a+b)}{da} = \frac{d(a+b)}{db} = 1$

…

$$f(x_1, x_2) \equiv (2x_1 + x_2)^2$$

$$\begin{aligned} z_1 &= \texttt{add}(x_1, x_1) \\ z_2 &= \texttt{add}(z_1, x_2) \\ f &= \texttt{square}(z_2) \end{aligned}$$

Derivation Step	
$\frac{df}{dx_1} = \frac{dz_2^2}{dz_2} \cdot \frac{dz_2}{dx_1}$	$f = z_2^2$
$\frac{df}{dx_1} = 2z_2 \cdot \frac{dz_2}{dx_1}$	$\frac{d(a^2)}{da} = 2a$
$\frac{df}{dx_1} = 2z_2 \cdot \frac{d(z_1+x_2)}{dx_1}$	$z_2 = z_1 + x_2$
$\frac{df}{dx_1} = 2z_2 \cdot \left(1 \cdot \frac{dz_1}{dx_1} + 1 \cdot \frac{dx_2}{dx_1}\right)$	$\frac{d(a+b)}{da} = \frac{d(a+b)}{db} = 1$

https://justindomke.wordpress.com/

Generalizing backprop

- Starting point: a function of *n* variables
- Step 1: eval your function as a series of assignments **Wengert list**
- Step 2: back propagate by going thru the list in reverse order, starting with… $\frac{dx_N}{dx_N} \leftarrow 1$
- …and using the chain rule

$$\frac{dx_N}{dx_i} = \sum_{j:i \in \pi(j)} \frac{dx_N}{dx_j} \frac{\partial x_j}{\partial x_i}$$

e.g.

$$x_7 = x_2 + x_5$$
$$\pi(7) = (2, 5)$$
$$f_7 = \text{add}$$

Step 1: forward

inputs: $x_1, x_2, \ldots, x_n$

for $i = n+1, n+2, \ldots, N$

$$x_i \leftarrow f_i(\mathbf{x}_{\pi(i)})$$

return x_N

Step 2: backprop

for $i = N-1, N-2, \ldots, 1$

$$\frac{dx_N}{dx_i} \leftarrow \sum_{j:i \in \pi(j)} \frac{dx_N}{dx_j} \frac{\partial f_j}{\partial x_i}$$

A function eval'd at this point

Values

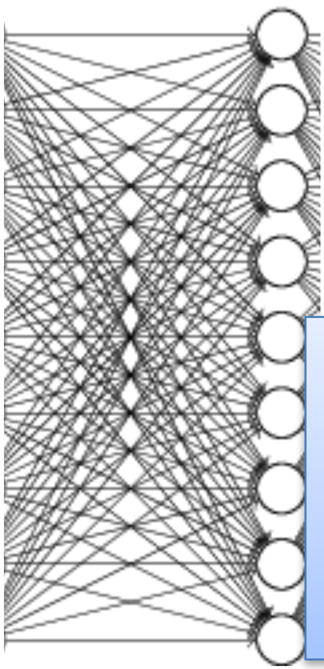
Computed in previous step

Recap: logistic regression with SGD

Let X be a matrix with *k* examples
Let $\mathbf{w}_i$ be the input weights for the i-th hidden unit
Then Z = X W is output (pre-sigmoid) for all *m* units for all *k* examples

$\mathbf{x}_1$	1	0	1	1
$\mathbf{x}_2$	…			
…				
$\mathbf{x}_k$				

w_1	w_2	w_3	…	w_m
0.1	-0.3	…		
-1.7	…			
0.3	…			
1.2				



There's a *lot* of chances to do this in parallel…. with parallel matrix multiplication

XW =

$\mathbf{x}_1.\mathbf{w}_1$	$\mathbf{x}_1.\mathbf{w}_2$	…	$\mathbf{x}_1.\mathbf{w}_m$
$\mathbf{x}_k.\mathbf{w}_1$	**…**	…	$\mathbf{x}_k.\mathbf{w}_m$

Example: 2-layer neural network

Step 1: forward

inputs: $x_1, x_2, \ldots, x_n$

for $i = n+1, n+2, \ldots, N$

$$x_i \leftarrow f_i(\mathbf{x}_{\pi(i)})$$

return x_N

Step 1: backprop

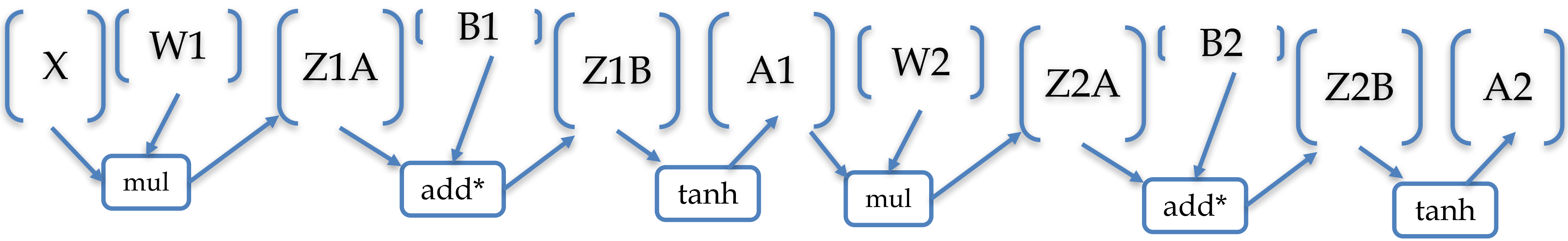
for $i = N-1, N-2, \ldots, 1$

$$\frac{dx_N}{dx_i} \leftarrow \sum_{j:i\in\pi(j)} \frac{dx_N}{dx_j}\frac{\partial f_j}{\partial x_i}$$

```
Inputs: X,W1,B1,W2,B2
Z1a = mul(X,W1)          // matrix mult
Z1b = add*(Z1a,B1)       // add bias vec
A1 = tanh(Z1b)           //element-wise
Z2a = mul(A1,W2)
Z2b = add*(Z2a,B2)
A2 = tanh(Z2b)           // element-wise
P = softMax(A2)          // vec to vec
-----------------------------------------
C = crossEnt_Y(P)        // cost function
```

Target Y; *N* examples; *K* outs; *D* feats, *H* hidden

Example: 2-layer neural network




```
Inputs: X,W1,B1,W2,B2
Z1a = mul(X,W1)          // matrix mult
Z1b = add*(Z1a,B1)       // add bias vec
A1 = tanh(Z1b)           //element-wise
Z2a = mul(A1,W2)
Z2b = add*(Z2a,B2)
A2 = tanh(Z2b)           // element-wise
P = softMax(A2)          // vec to vec
-----------------------------------------
C = crossEnt_Y(P)        // cost function
```

Target Y; *N* examples; *K* outs; *D* feats, *H* hidden

X is *N*D*, W1 is *D*H*, B1 is *1*H*, W2 is *H*K*, …

Z1a is *N*H*

Z1b is *N*H*

A1 is *N*H*

Z2a is *N*K*

Z2b is *N*K*

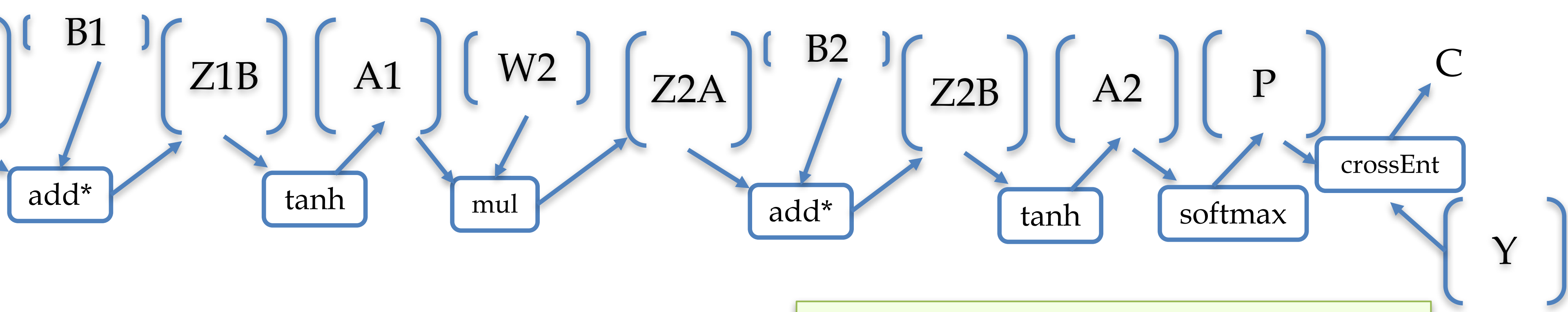
A2 is *N*K*

P is *N*K*

C is a scalar

$$\mathbf{p}_i = \frac{\exp(\mathbf{a_i})}{\sum_j \exp(\mathbf{a}_j)}$$

Example: 2-layer neural network



X is *N*D*, W1 is *D*H*, B1 is *1*H*, W2 is *H*K*, …

```
Inputs: X,W1,B1,W2,B2
Z1a = mul(X,W1)        // matrix mult
Z1b = add*(Z1a,B1)     // add bias vec
A1 = tanh(Z1b)         //element-wise
Z2a = mul(A1,W2)
Z2b = add*(Z2a,B2)
A2 = tanh(Z2b)         // element-wise
P = softMax(A2)        // vec to vec
```
- -

$C = \text{crossEnt}_Y(P)$ // cost function

Z1a is *N*H*

Z1b is *N*H*

A1 is *N*H*

Z2a is *N*K*

Z2b is *N*K*

A2 is *N*K*

P is *N*K*

C is a scalar

$$\mathbf{p}_i = \frac{\exp(\mathbf{a}_i)}{\sum_j \exp(\mathbf{a}_j)}$$

Target Y; *N* examples; *K* outs; *D* feats, *H* hidden

Example: 2-layer neural network

Step 1: forward

inputs: $x_1, x_2, \ldots, x_n$

for $i = n+1, n+2, \ldots, N$

$$x_i \leftarrow f_i(\mathbf{x}_{\pi(i)})$$

return x_N

Step 1: backprop

for $i = N-1, N-2, \ldots, 1$

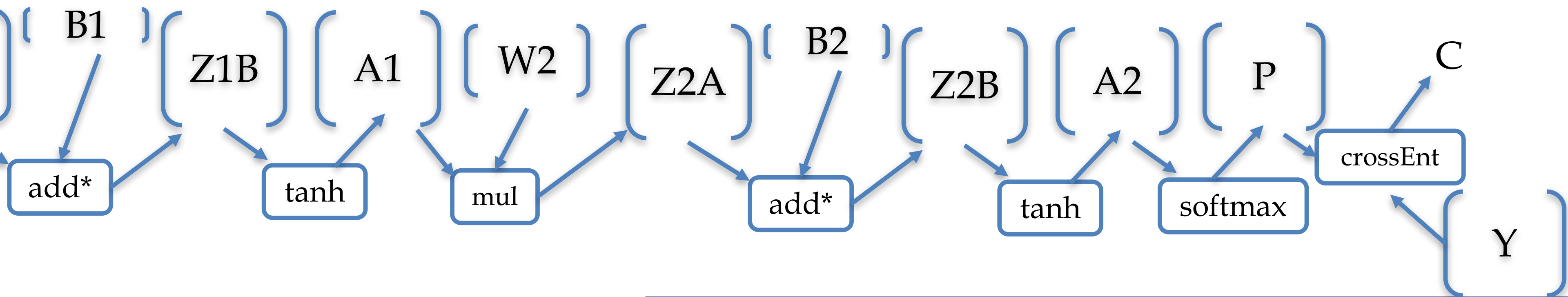
$$\frac{dx_N}{dx_i} \leftarrow \sum_{j:i\in\pi(j)} \frac{dx_N}{dx_j}\frac{\partial f_j}{\partial x_i}$$

```
Inputs: X,W1,B1,W2,B2
Z1a = mul(X,W1)         // matrix mult
Z1b = add*(Z1a,B1)      // add bias vec
A1 = tanh(Z1b)          //element-wise
Z2a = mul(A1,W2)
Z2b = add*(Z2a,B2)
A2 = tanh(Z2b)          // element-wise
P = softMax(A2)         // vec to vec
-----------------------------------
C = crossEnt_Y(P)       // cost function
```

```
dC/dC = 1
dC/dP = dC/dC * dCrossEnt_Y/dP
dC/dA2 = dC/dP * dsoftmax/dA2
dC/Z2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a
dC/dB2 = dC/dZ2b * dadd*/dB2
dC/dA1 = ...
```

Target Y; *N* rows; *K* outs; *D* feats, *H* hidden

Example: 2-layer neural network



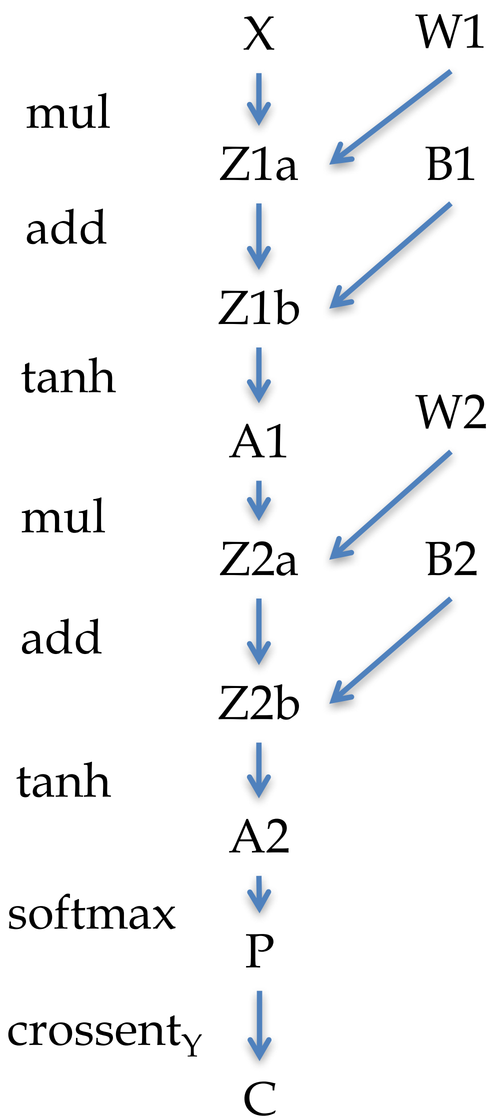
Inputs: X,W1,B1,W2,B2
Z1a = mul(X,W1) // matrix mult
Z1b = add*(Z1a,B1) // add bias vec
A1 = tanh(Z1b) //element-wise
Z2a = mul(A1,W2)
Z2b = add*(Z2a,B2)
A2 = tanh(Z2b) // element-wise
P = softMax(A2) // vec to vec

C = crossEnt$_Y$(P) // cost function

dC/dC = 1
dC/dP = dC/dC * dCrossEnt$_Y$/dP   *N*K*
dC/dA2 = dC/dP * dsoftmax/dA2   *N*K*
dC/Z2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a
dC/dB2 = dC/dZ2b * dadd*/dB2
dC/dA1 = …

Target Y; *N* rows; *K* outs; *D* feats, *H* hidden

Example: 2-layer neural network



dC/dC = 1
dC/dP = dC/dC * dCrossEnt$_Y$/dP
dC/dA2 = dC/dP * dsoftmax/dA2
dC/dZ2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a

- dC/dB2 = dC/dZ2b * dadd*/dB2

dC/dA1 = dC/dZ2a * dmul/dA1

- dC/dW2 = dC/dZ2a * dmul/dW2

dC/dZ1b = dC/dA1 * dtanh/dZ1b
dC/dZ1a = dC/dZ1b * dadd*/dZ1a

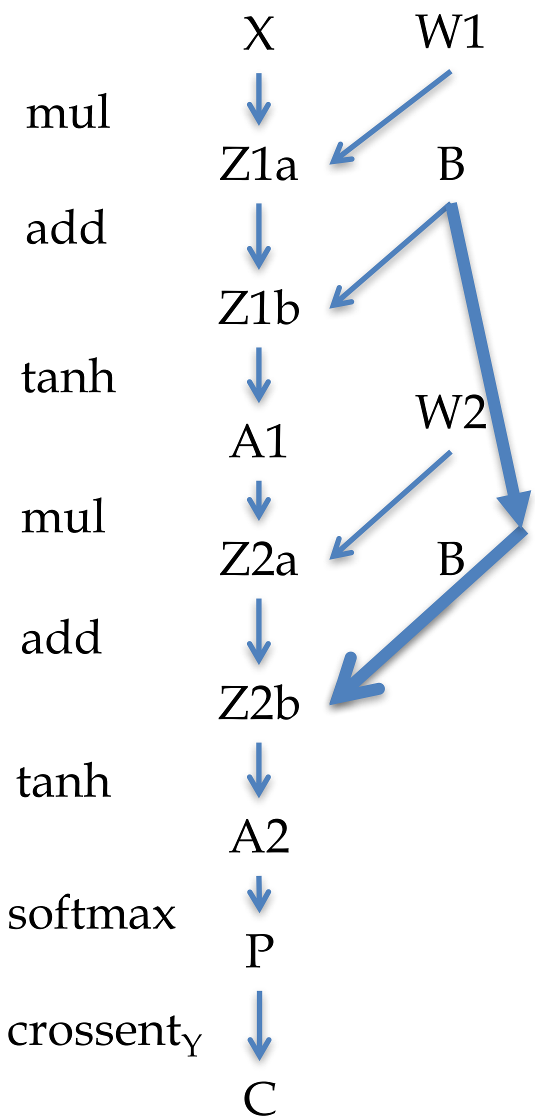
- dC/dB1 = dC/dZ1b * dadd*/dB1

dC/dX = dC/dZ1a * dmul*/dZ1a

- dC/dW1 = dC/dZ1a * dmul*/dW1

Example: 2-layer neural network

with "tied parameters"



dC/dC = 1
dC/dP = dC/dC * $dCrossEnt_Y$/dP
dC/dA2 = dC/dP * dsoftmax/dA2
dC/dZ2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a
- dC/dB2 = dC/dZ2b * dadd*/dB

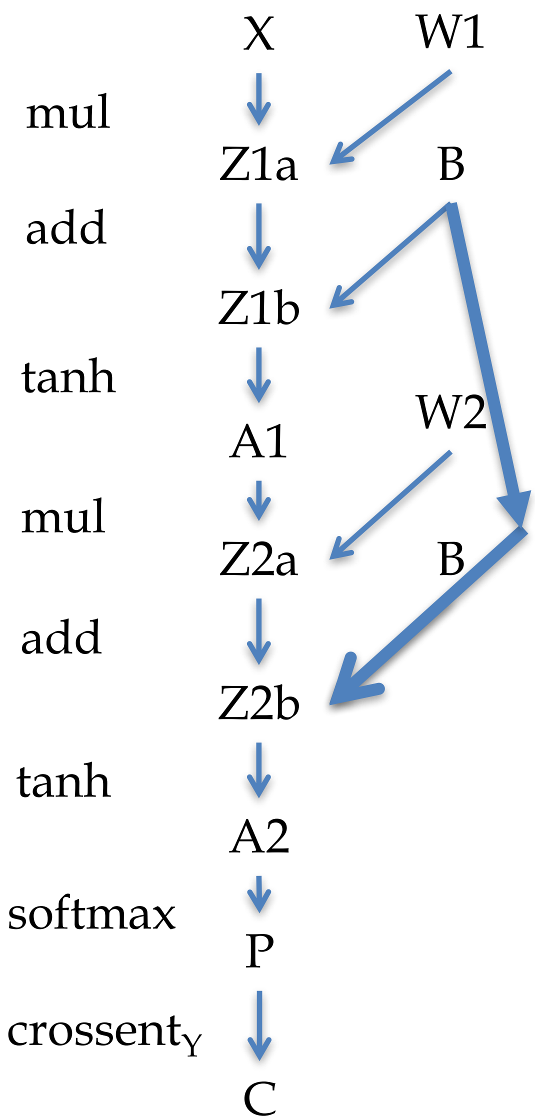
dC/dA1 = dC/dZ2a * dmul/dA1
- dC/dW2 = dC/dZ2a * dmul/dW2

dC/dZ1b = dC/dA1 * dtanh/dZ1b
dC/dZ1a = dC/dZ1b * dadd*/dZ1a
- dC/dB1 = dC/dZ1b * dadd*/dB

dC/dX = dC/dZ1a * dmul*/dZ1a
- dC/dW1 = dC/dZ1a * dmul*/dW1

Example: 2-layer neural network

with "tied parameters"



dC/dC = 1
dC/dP = dC/dC * dCrossEnt$_Y$/dP
dC/dA2 = dC/dP * dsoftmax/dA2
dC/dZ2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a
dC/dB += dC/dZ2b * dadd*/dB
dC/dA1 = dC/dZ2a *dmul/dA1

- dC/dW2 = dC/dZ2a *dmul/dW2

dC/dZ1b = dC/dA1 * dtanh/dZ1b
dC/dZ1a = dC/dZ1b * dadd*/dZ1a
dC/dB += dC/dZ1b * dadd*/dB
dC/dX = dC/dZ1a * dmul*/dZ1a

- dC/dW1 = dC/dZ1a * dmul*/dW1

Example: 2-layer neural network

Step 1: forward

inputs: $x_1, x_2, \ldots, x_n$

for $i = n+1, n+2, \ldots, N$

$$x_i \leftarrow f_i(\mathbf{x}_{\pi(i)})$$

return x_N

Step 1: backprop

for $i = N-1, N-2, \ldots, 1$

$$\frac{dx_N}{dx_i} \leftarrow \sum_{j: i \in \pi(j)} \frac{dx_N}{dx_j} \frac{\partial f_j}{\partial x_i}$$

```
Inputs: X,W1,B1,W2,B2
Z1a = mul(X,W1)        // matrix mult
Z1b = add(Z11,B1)      // add bias vec
A1 = tanh(Z1b)         //element-wise
Z2a = mul(A1,W2)
Z2b = add(Z2a,B2)
A2 = tanh(Z2b)         // element-wise
P = softMax(A2)        // vec to vec
C = crossEnt_Y(P)      // cost function
```

*N*H*

```
dC/dC = 1
dC/dP = dC/dC * dCrossEnt_Y/dP
dC/dA2 = dC/dP * dsoftmax/dA2
dC/Z2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a
dC/dB2 = dC/dZ2b * dadd*/dB2
dC/dA1 = …
```

*N*K*

$$-\frac{1}{N}\sum_i \left(\frac{\mathbf{p}_i - \mathbf{y}_i}{\mathbf{y}_i(1-\mathbf{y}_i)} \right)$$

Need a backward form for each matrix operation used in forward

Target Y; *N* rows; *K* outs; *D* feats, *H* hidden

Example: 2-layer neural network

Step 1: forward

inputs: $x_1, x_2, \ldots, x_n$

for $i = n+1, n+2, \ldots, N$

$$x_i \leftarrow f_i(\mathbf{x}_{\pi(i)})$$

return x_N

Step 1: backprop

for $i = N-1, N-2, \ldots, 1$

$$\frac{dx_N}{dx_i} \leftarrow \sum_{j:i\in\pi(j)} \frac{dx_N}{dx_j}\frac{\partial f_j}{\partial x_i}$$

```
Inputs: X,W1,B1,W2,B2
Z1a = mul(X,W1)       // matrix mult
Z1b = add(Z11,B1)     // add bias vec
A1 = tanh(Z1b)        //element-wise
Z2a = mul(A1,W2)
Z2b = add(Z2a,B2)
A2 = tanh(Z2b)        // element-wise
P = softMax(A2)       // vec to vec
C = crossEnt_Y(P)     // cost function
```

*N*H*

```
dC/dC = 1
dC/dP = dC/dC * dCrossEnt_Y/dP
dC/dA2 = dC/dP * dsoftmax/dA2
dC/Z2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a
dC/dB2 = dC/dZ2b * dadd*/dB2
dC/dA1 = ...
```

*N*K*

Need a backward form for each matrix operation used in forward, **with respect to each argument**

Target Y; *N* rows; *K* outs; *D* feats, *H* hidden

Example: 2-layer neural network

Step 1: forward

inputs: $x_1, x_2, \ldots, x_n$

for $i = n+1, n+2, \ldots, N$

$$x_i \leftarrow f_i(\mathbf{x}_{\pi(i)})$$

return x_N

```
Inputs: X,W1,B1,W2,B2
Z1a = mul(X,W1)        // matrix mult
Z1b = add(Z11,B1)      // add bias vec
A1 = tanh(Z1b)         //element-wise
Z2a = mul(A1,W2)       N*H
Z2b = add(Z2a,B2)
A2 = tanh(Z2b)         // element-wise
P = softMax(A2)        // vec to vec
C = crossEnt_Y(P)      // cost function
```

An autodiff package usually includes

- A collection of matrix-oriented operations (mul, add*, …)
- For each operation
  - A forward implementation
  - A backward implementation for each argument

Need a backward form for each matrix operation used in forward, **with respect to each argument**

Target Y; *N* rows; *K* outs; *D* feats, *H* hidden

Stopped Monday

What's Going On Here?

Differentiating a Wengert list: a simple case

High school: *symbolic differentiation*, compute a symbolic form of the deriv of *f*

Now: *automatic differentiation*, find an algorithm to compute *f*′(*a*) at any point *a*

$$
\begin{aligned}
z_1 &= f_1(z_0) \\
z_2 &= f_2(z_1) \\
&\ldots \\
z_m &= f_m(z_{m-1})
\end{aligned}
$$

$$
\begin{aligned}
\frac{dz_m}{dz_0} &= \frac{dz_m}{dz_{m-1}} \frac{dz_{m-1}}{dz_0} \\
&= \frac{dz_m}{dz_{m-1}} \frac{dz_{m-1}}{dz_{m-2}} \frac{dz_{m-2}}{dz_0} \\
\ldots \\
&= \frac{dz_m}{dz_{m-1}} \frac{dz_{m-1}}{dz_{m-2}} \ldots \frac{dz_1}{dz_0}
\end{aligned}
$$

Differentiating a Wengert list: a simple case

Now: *automatic differentiation*, find an algorithm to compute $f'(a)$ at any point a

$$
\begin{aligned}
z_1 &= f_1(z_0) & a_1 &= f_1(a) \\
z_2 &= f_2(z_1) & a_2 &= f_2(f_1(a)) \\
&\dots & &\dots \\
z_m &= f_m(z_{m-1}) & a_m &= f_m(f_{m-1}(f_{m-2}(\dots f_1(a)\dots)))
\end{aligned}
$$

Notation: $h_{i,j} \rightarrow \frac{dz_i}{dz_j}$

a_i is the i-th output on input a

Differentiating a Wengert list: a simple case

What did Liebnitz mean with this?

$$
\begin{aligned}
z_1 &= f_1(z_0) \\
z_2 &= f_2(z_1) \\
&\dots \\
z_m &= f_m(z_{m-1})
\end{aligned}
$$

$$\frac{dz_m}{dz_0} = \frac{dz_m}{dz_{m-1}} \frac{dz_{m-1}}{dz_0}$$

for all a

$$h_{m,0}(a) = f'_m(a_m) * h_{m-1,0}(a)$$

Notation: $h_{i,j} \rightarrow \frac{dz_i}{dz_j}$

a_i is the i-th output on input a

Differentiating a Wengert list: a simple case

$$
\begin{aligned}
z_1 &= f_1(z_0) \\
z_2 &= f_2(z_1) \\
&\dots \\
z_m &= f_m(z_{m-1})
\end{aligned}
$$

$$\frac{dz_m}{dz_0} = \frac{dz_m}{dz_{m-1}} \frac{dz_{m-1}}{dz_{m-2}} \dots \frac{dz_1}{dz_0}$$

for all a

$$h_{m,0}(a) = f'_m(a_m) \cdot f'_{m-1}(a_{m-1}) \cdots f'_2(a_1) \cdot f'_1(a)$$

Notation: $h_{i,j} \rightarrow \frac{dz_i}{dz_j}$

Differentiating a Wengert list: a simple case

$$\frac{dz_m}{dz_0} = \frac{dz_m}{dz_{m-1}} \frac{dz_{m-1}}{dz_{m-2}} \cdots \frac{dz_1}{dz_0}$$

for all a

$$h_{m,0}(a) = f'_m(a_m) \cdot f'_{m-1}(a_{m-1}) \cdots f'_2(a_1) \cdot f'_1(a)$$

backprop routine compute order

$$h_{m,0}(a) = \Big(\Big(\Big(\big(f'_m(a_m) \cdot f'_{m-1}(a_{m-1})\big)\Big) \cdot f'_{m-2}(a_{m-2})\Big) \cdots f'_2(a_1)\big)\Big) \cdot f'_1(a)$$

$$\texttt{delta}[z_i] = f'_m(a_m) \cdots f'_i(a_i)$$

Differentiating a Wengert list

```
DG = { "add" : [ (lambda a,b: 1),  (lambda a,b: 1) ],
       "square": [ lambda a:2*a ] }
```

```
[ ("z1", "add", ("x1","x1")),
  ("z2", "add", ("z1","x2")),
  ("f", "square", ("z2")) ]
```

def backprop(f,val)
    initialize delta: delta$[f] = 1$
    for (z,g,($y_1, \ldots, y_k$)) in the list, in reverse order:
        for $i = 1, \ldots, k$:
            op$_i$ = DG[g][i]
            if delta[y_i] is not defined set delta[y_i] = 0
            delta$[y_i]$ = delta$[y_i]$ + delta[z] * op$_i$(val[y_1],...,val[y_k])

https://justindomke.wordpress.com/

Generalizing backprop

- Starting point: a function of *n* variables
- Step 1: ~~code your function as a series of assignments~~ Wengert list

Step 1: forward

inputs: $x_1, x_2, \ldots, x_n$

for $i = n+1, n+2, \ldots, N$

$$x_i \leftarrow f_i(\mathbf{x}_{\pi(i)})$$

return x_N

- Better plan: overload your matrix operators so that when you use them in-line they build an **expression graph**
- Convert the expression graph to a Wengert list when necessary