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

Autodiff for AP Calculus Problems

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

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

computation graph

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

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

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

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	
$\frac{df}{dx_1} = \frac{dz_2^2}{dz_2} \cdot \frac{dz_2}{dx_1}$	$f = z_2^2$
$\frac{df}{dx_1} = 2z_2 \cdot \frac{dz_2}{dx_1}$	$\frac{d(a^2)}{da} = 2a$
$\frac{df}{dx_1} = 2z_2 \cdot \frac{d(z_1+x_2)}{dx_1}$	$z_2 = z_1 + x_2$
$\frac{df}{dx_1} = 2z_2 \cdot \left(1 \cdot \frac{dz_1}{dx_1} + 1 \cdot \frac{dx_2}{dx_1}\right)$	$\frac{d(a+b)}{da} = \frac{d(a+b)}{db} = 1$
$\frac{df}{dx_1} = 2z_2 \cdot \left(1 \cdot \frac{d(x_1+x_1)}{dx_1} + 1 \cdot \frac{dx_2}{dx_1}\right)$	$z_1 = x_1 + x_1$
$\frac{df}{dx_1} = 2z_2 \cdot \left(1 \cdot \left(1 \cdot \frac{dx_1}{dx_1} + 1 \cdot \frac{dx_1}{dx_1}\right) + 1 \cdot \frac{dx_2}{dx_1}\right)$	$\frac{d(a+b)}{da} = \frac{d(a+b)}{db} = 1$
$\frac{df}{dx_1} = 2z_2 \cdot (1 \cdot (1 \cdot 1 + 1 \cdot 1) + 1 \cdot 0)$	$\frac{da}{da} = 1$ and $\frac{da}{db} = 0$ for inputs a, b
$\frac{df}{dx_1} = 2z_2 \cdot 2 = 8x_1 + 4x_2$	simplify

https://justindomke.wordpress.com/

Generalizing backprop

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

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

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

A function

Step 2: backprop

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

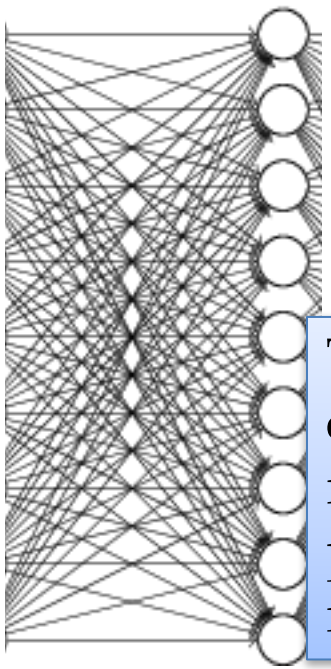
$$\frac{dx_N}{dx_i} \leftarrow \sum_{j:i\in\pi(j)} \frac{dx_N}{dx_j}\frac{\partial f_j}{\partial x_i}$$

Computed in previous step

Autodiff for a NN

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

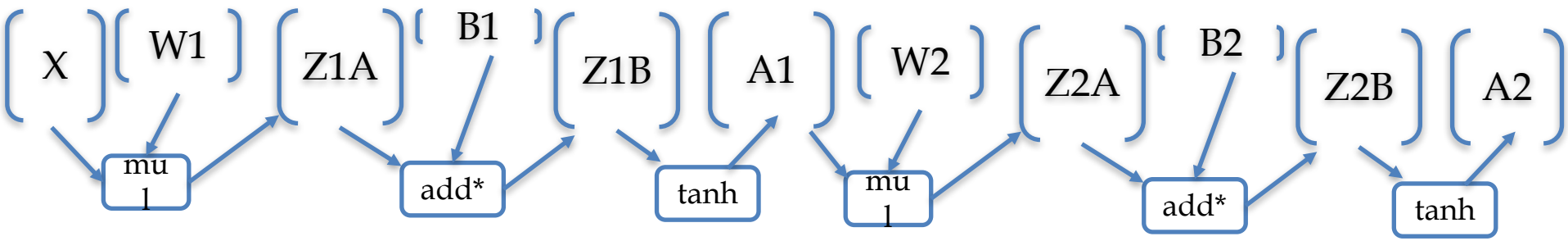
for $i = N-1, N-2, \ldots, 1$

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

```
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*

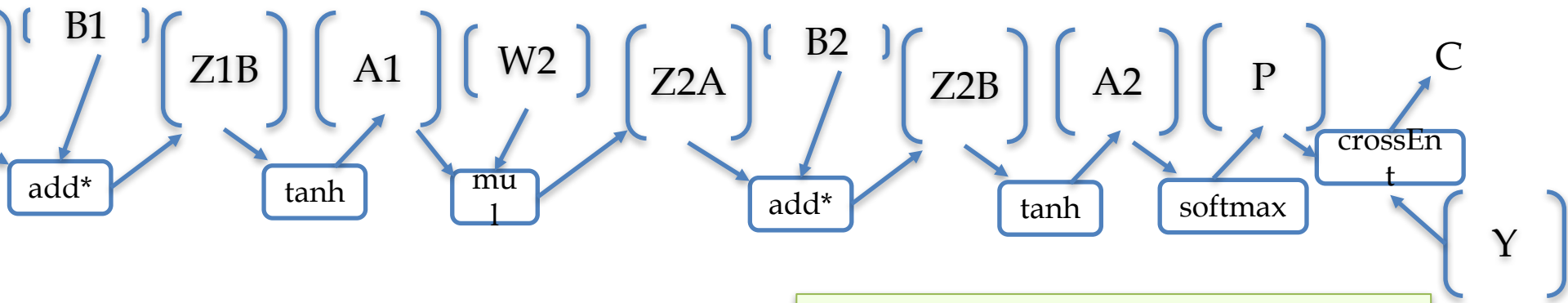
A2 is *N*K*

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




```
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
```

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

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

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

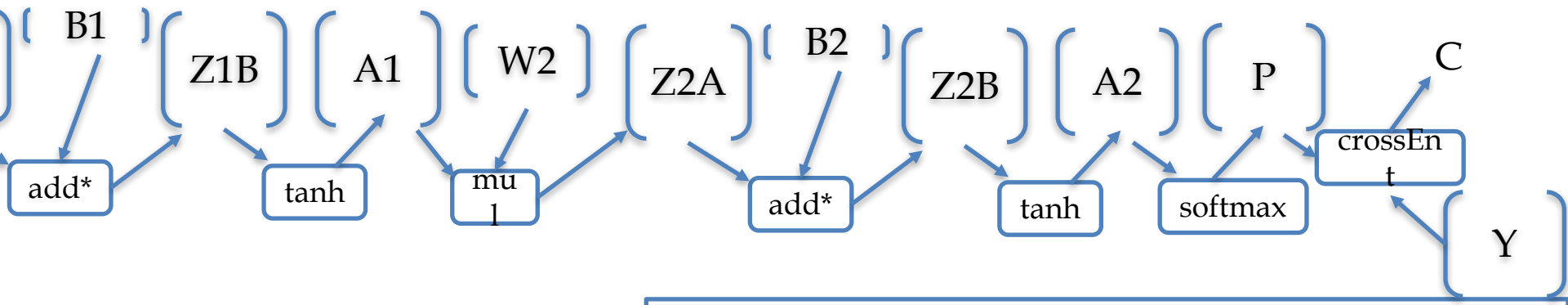
$$\frac{dx_N}{dx_i} \leftarrow \sum_{j:i\in\pi(j)} \frac{dx_N}{dx_j}\frac{\partial f_j}{\partial x_i}$$

```
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

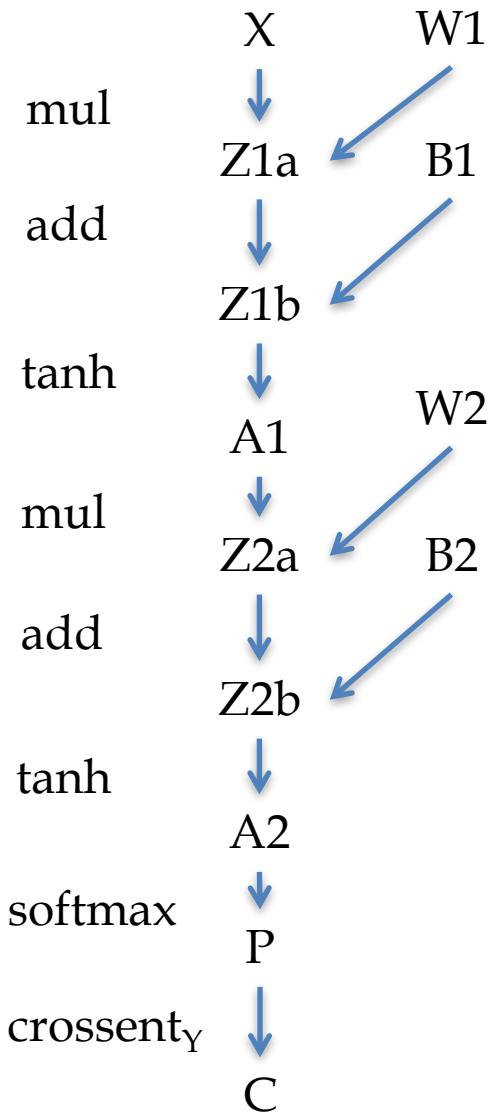
dC/dC = 1
dC/dP = dC/dC * dCrossEnt_Y/dP
dC/dA2 = dC/dP * dsoftmax/dA2
dC/Z2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a
dC/dB2 = dC/dZ2b * dadd*/dB2
dC/dA1 = ...

*N*K*

*N*K*

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

Example: 2-layer neural network

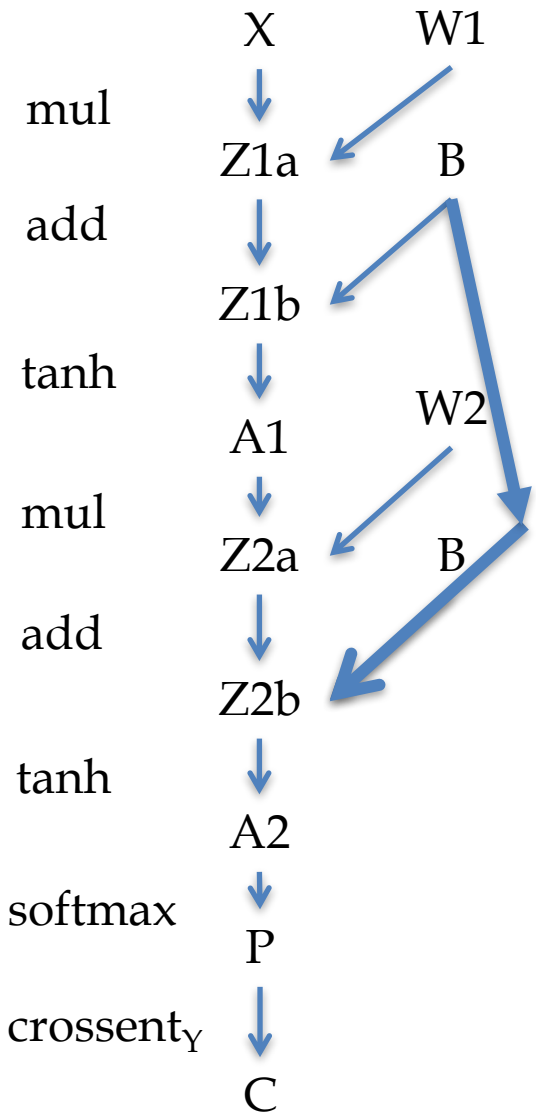


dC/dC = 1
dC/dP = dC/dC * dCrossEnt$_Y$/dP
dC/dA2 = dC/dP * dsoftmax/dA2
dC/dZ2b = dC/dA2 * dtanh/dZ2b
dC/dZ2a = dC/dZ2b * dadd*/dZ2a
- dC/dB2 = dC/dZ2b * dadd*/dB2
dC/dA1 = dC/dZ2a *dmul/dA1
- dC/dW2 = dC/dZ2a *dmul/dW2

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

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)
Z2b = add(Z2a,B2)
A2 = tanh(Z2b)         // element-wise
P = softMax(A2)        // vec to vec
C = crossEnt(P)        // cost function
```

*N*H*

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

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

- It's still a little awkward to program with a list of assignments so….

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

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*

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

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

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 ouput on input a

Differentiating a Wengert list: a simple case

What did Liebnitz mean with this?

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

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

for all a

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

Notation: $h_{i,j} \rightarrow \frac{dz_i}{dz_j}$ *a_i is the i-th ouput on input a*

Differentiating a Wengert list: a simple case

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

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

for all a

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

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

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

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

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,...,y_k)) in the list, in reverse order:
        for i = 1,...,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

Systems that implement Automatic Differentiation

Example of an Autodiff System: Theano

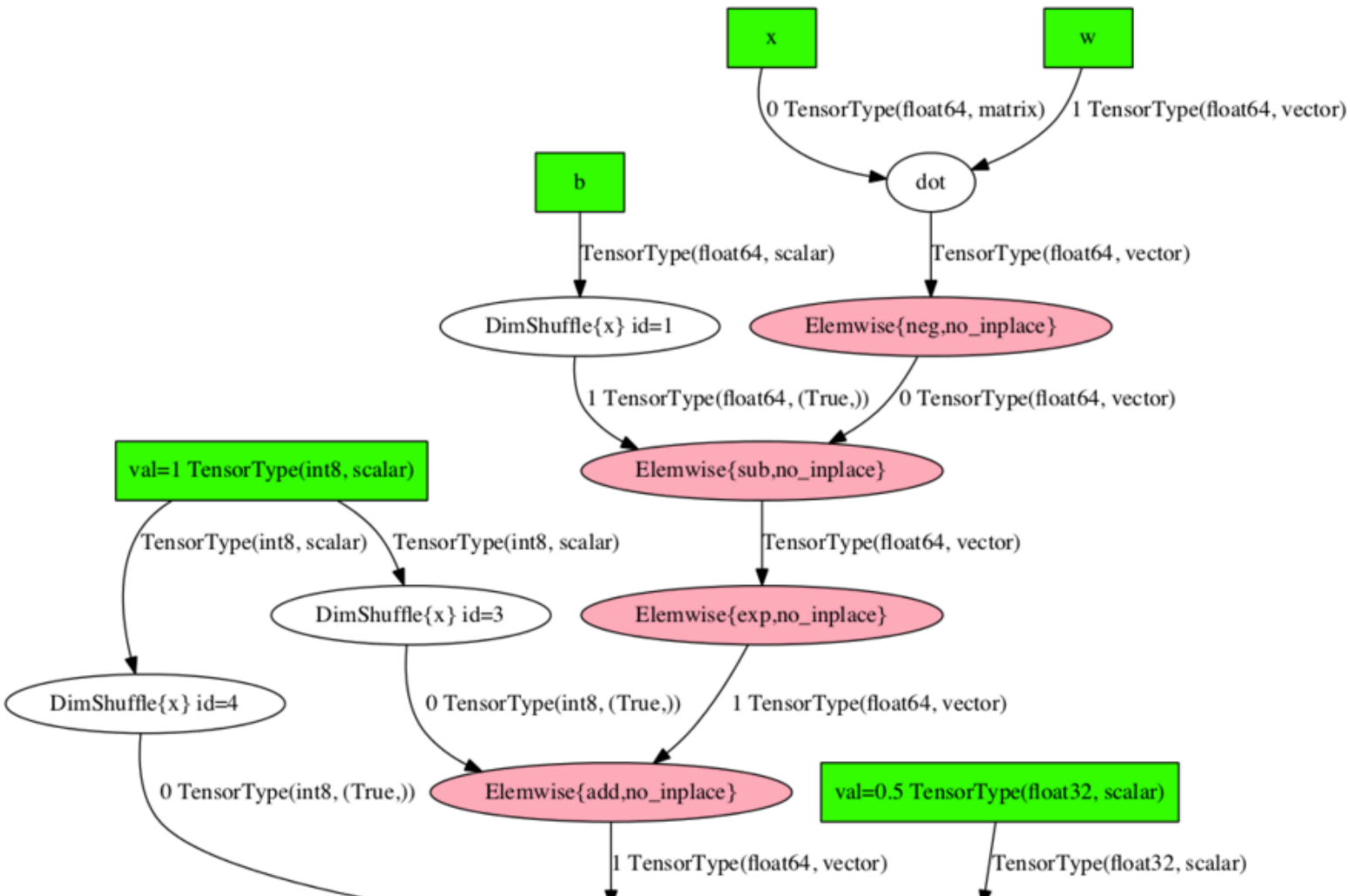
```
> import numpy
> import theano
> import theano.tensor as T
> rng = numpy.random
> # Training data
> N = 400
> feats = 784
> D = … # generate some training data
> training_steps = 10000
> # Declare Theano symbolic variables
> x = T.matrix("x")
> y = T.vector("y")
> w = theano.shared(rng.randn(feats).astype(theano.config.floatX), name="w")
> b = theano.shared(numpy.asarray(0., dtype=theano.config.floatX), name="b")
> x.tag.test_value = D[0]
> y.tag.test_value = D[1]
> # Construct Theano expression graph
> p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b)) # Probability of having a one
> pre
> # C
> xen
> cos
> gw,
```

```
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
prediction = p_1 > 0.5
```

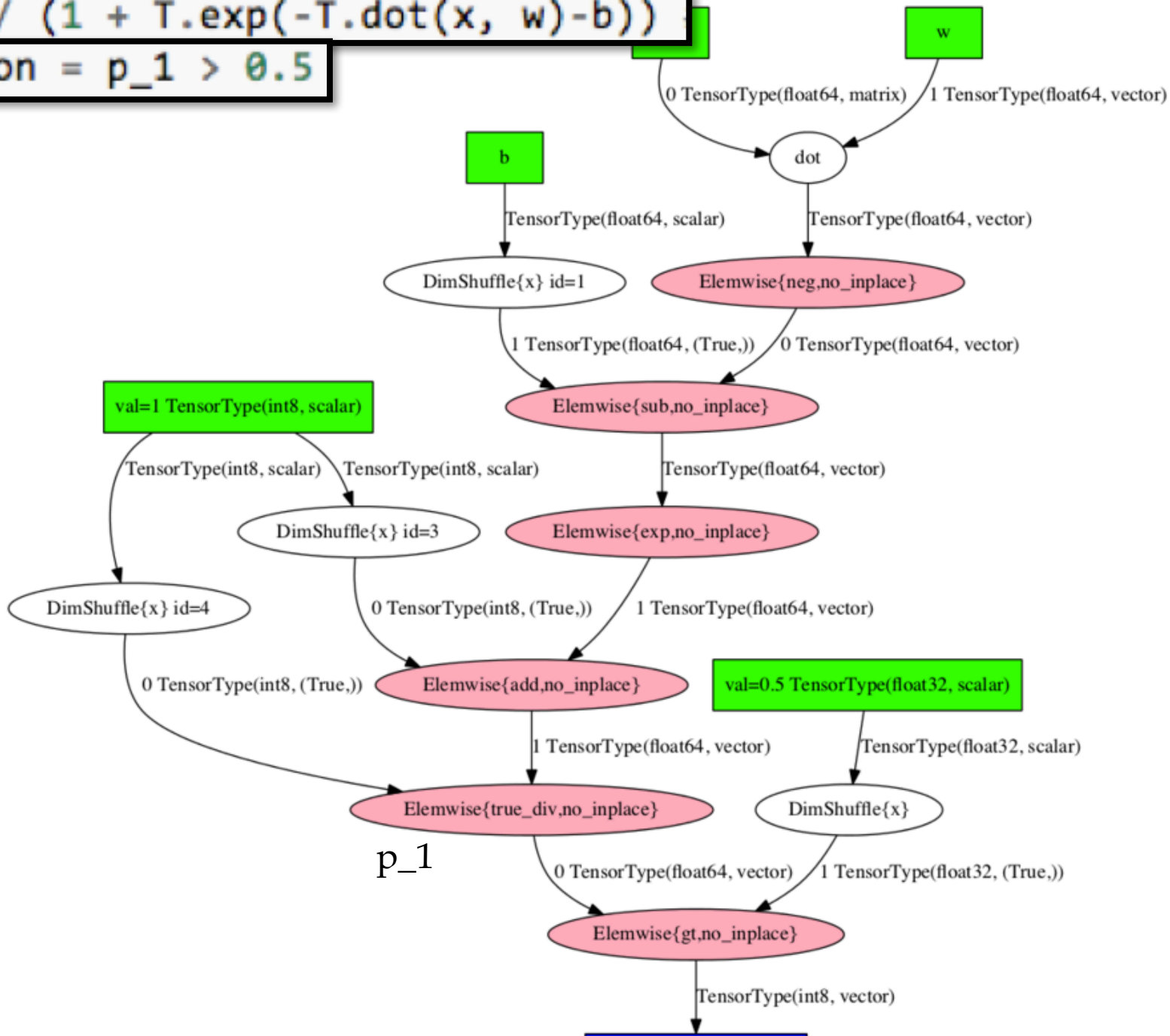
```
> xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
> cost = xent.mean() + 0.01*(w**2).sum() #
> gw,gb = T.grad(cost, [w,b])
```

Example: Theano

```
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
prediction = p_1 > 0.5
```



```
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
prediction = p_1 > 0.5
```



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)
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

- A way of composing operations into expressions (often using operator overloading) which evaluate to **expression trees**
- Expression simplification/compilation

Some tools that use autodiff

- Theano:
  - Univ Montreal system, Python-based, first version 2007 now v0.8, integrated with GPUs
  - Many libraries build over Theano (Lasagne, Keras, Blocks..)
- Torch:
  - Collobert et al, used heavily at Facebook, based on Lua. Similar performance-wise to Theano
- PyTorch
  - Similar API to Torch but in Python
- TensorFlow:
  - Google system
  - Also supported by Keras
- …

MORE ABOUT SYSTEMS THAT DO AUTODIFF

Inputs and parameters

- Our abstract reverse-mode autodiff treats all nodes in the graph the same
- But they are different:
  - some are functions of other nodes
  - some are ”inputs” (like X,Y)
  - some are “parameters” (like W)
  - inputs and parameters are both autodiff leaves
    - but what happens to them is different
    - in HW they have different names

Inputs and parameters

- TensorFlow: inputs are ”placeholders”
  - have type and shape (for the matrix) but not a value
  - you bind the placeholders to values (eg numpy arrays) when you execute the computation graph
- TensorFlow: parameters are graph nodes
  - usually with an explicit assign operator added
    - **w_update = w.assign(w – rate*grad)**
  - then the system knows to reuse memory
  - that’s called an “update” step
- TensorFlow: an optimizer will output an update

More detail

- Lots of tutorials
- One nice walkthrough of same task in TensorFlow, PyTorch, Theano:
  - http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf

AUTODIFF IN MORE DETAIL

Walkthru of an Autodiff Program

Matrix and vector operations are very useful

$$f(\mathbf{x}, \mathbf{y}, W) \equiv \texttt{crossEntropy}(\texttt{softmax}(\mathbf{x} \cdot W), \mathbf{y}) + \texttt{frobeniusNorm}(W)$$

$$\begin{aligned}
\mathbf{z}_1 &= \texttt{dot}(\mathbf{x}, W) \\
\mathbf{z}_2 &= \texttt{softmax}(\mathbf{z}_1) \\
z_3 &= \texttt{crossEntropy}(\mathbf{z}_2, \mathbf{y}) \\
z_4 &= \texttt{frobeniusNorm}(W) \\
f &= \texttt{add}(z_3, z_4)
\end{aligned}$$

Walkthru of an Autodiff Program

Matrix and vector operations are very useful

$$f(\mathbf{x}, \mathbf{y}, W) \equiv \texttt{crossEntropy}(\texttt{softmax}(\mathbf{x} \cdot W), \mathbf{y}) + \texttt{frobeniusNorm}(W)$$

$$\begin{aligned}
\texttt{softmax}(\langle a_1, \ldots, a_d \rangle) &\equiv \langle \frac{e^{a_1}}{\sum_{i=1}^{d} e_i^a}, \ldots, \frac{e^{a_d}}{\sum_{i=1}^{d} e_i^a} \rangle \\
\texttt{crossEntropy}(\langle a_1, \ldots, a_d \rangle, \langle b_1, \ldots, b_d \rangle) &\equiv -\sum_{i=1}^{d} a_i \log b_i \\
\texttt{frobeniusNorm}(A) &\equiv \sqrt{\sum_{i=1}^{d} \sum_{j=1}^{k} a_{i,j}^2}
\end{aligned}$$

Walkthru of an Autodiff Program

Matrix and vector operations are very useful…but lets start with some simple scalar operations.

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

$$G = \{\texttt{add}, \texttt{multiply}, \texttt{square}\}$$

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

Python encoding

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

The Interesting Part: Evaluation and Differentiation of a Wengert list

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

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

```
G = { "add" : lambda a,b: a+b,
      "square": lambda a:a*a }

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

To compute f(3,7):

```
def eval(f)
    initialize val = { "x1" : 3, "x2" : 7 }
    for (z, g, (y_1, ..., y_k)) in the list:
        op = G[g]
        val[z] = op(val[y_1],...,val[y_k])
    return the last entry stored in val.
```

The Interesting Part: Evaluation and Differentiation of a Wengert list

$f(x_1, x_2$

$z_1 =$

$z_2 = \mathrm{add}(z_1, z_2)$

$f = \mathrm{square}(z_2)$

```
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")) ]
```

To compute f(3,7):

Populated with call to "eval"

```
def backprop(f,val)
    initialize delta: delta[f] = 1
    for (z,g,(y_1,...,y_k)) in the list, in reverse order:
        for i = 1,...,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])
```

The Tricky Part: Constructing a List

Goal: write code like this:

```
from xman import *
...
class f(XManFunctions):
    @staticmethod
    def half(a):
        ...
class Triangle(XMan):
    h = f.input()
    w = f.input()
    area = f.half(h*w)
...
xm = Triangle().setup()
print xm.operationSequence(xm.area)
```

Add your problem-specific functions: but just the syntax, not the G and DG definitions

Define the function to optimize with gradient descent

Use your function

Python code to create a Wengert list

```
from xman import *
...
class f(XManFunctions):
    @staticmethod
    def half(a):
        ...
class Triangle(XMan):
    h = f.input()
    w = f.input()
    area = f.half(h*w)
...
xm = Triangle().setup()
print xm.operationSequence(xm.area)
```

h, w, area are *registers*

also an automaticall y-named register z1 to hold h*w

```
[('z1', 'mul', ['h', 'w']),
 ('area', 'half', ['z1'])]
```

Python code to create a Wengert list

```
from xman import *
...
class f(XManFunctions):
    @staticmethod
    def half(a):
        ...
class Triangle(XMan):
    h = f.input()
    w = f.input()
    area = f.half(h*w)
...
xm = Triangle().setup()
print xm.operationSequence(xm.area)
```

some Python hacking can use the class variable name 'area' and insert it into the 'name' field of a register

and invent names for the others at setup() time

name	"h"
role	"input"

name	"w"
role	"input"

name	z1
role	"opOut"
defAs	01

name	"area"
role	"opOut"
defAs	02

Python code to create a Wengert list

```
from xman import *
...
class f(XManFunctions):
    @staticmethod
    def half(a):
        ...
class Triangle(XMan):
    h = f.input()
    w = f.input()
    area = f.half(h*w)
...
xm = Triangle().setup()
print xm.operationSequence(xm.area)
```

"half" is an operator

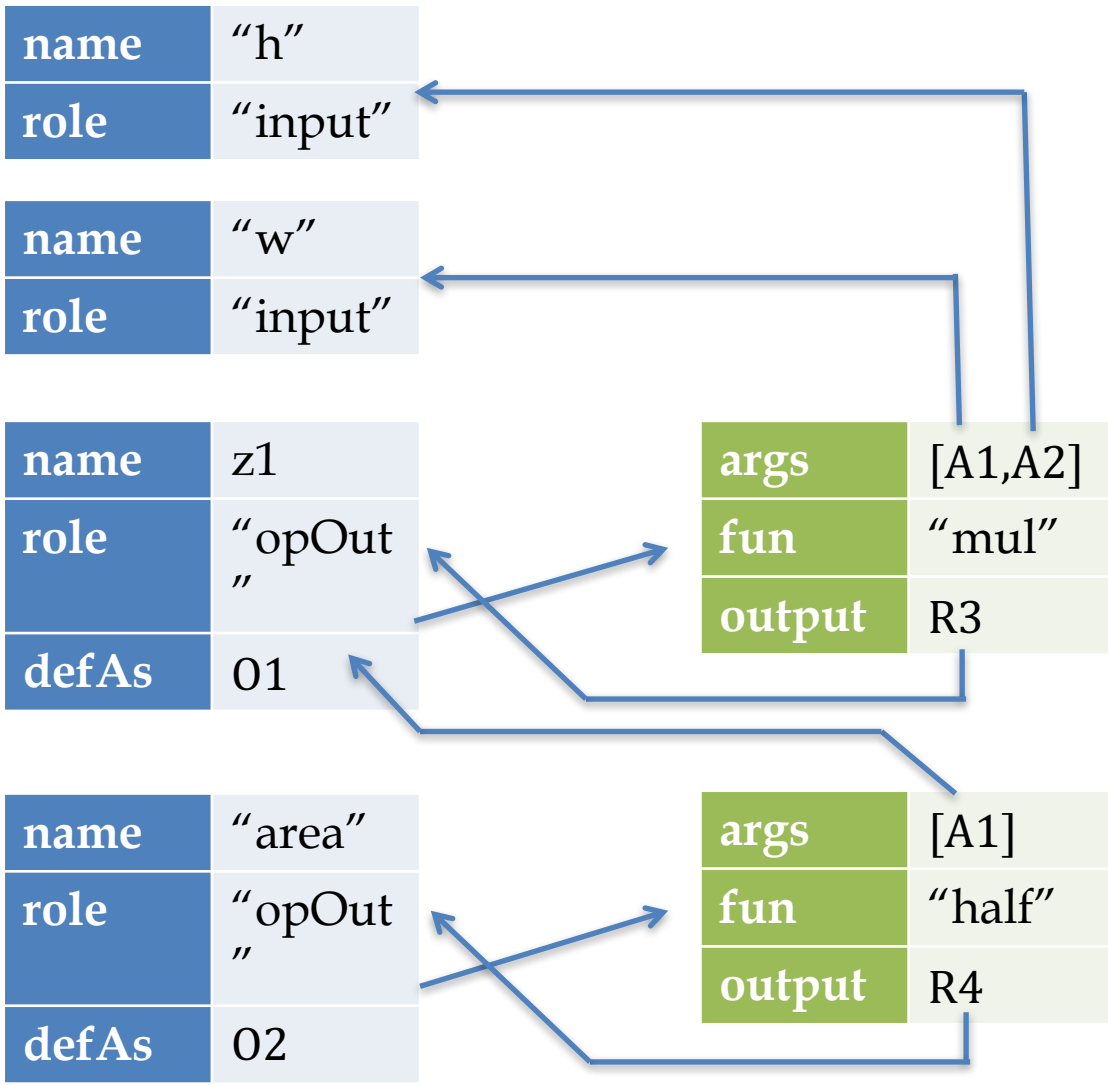
so is "mul" -- - h*w ➔ mul(h,w)

each instance of an operator points to registers that are arguments and outputs

Python code to create a Wengert list

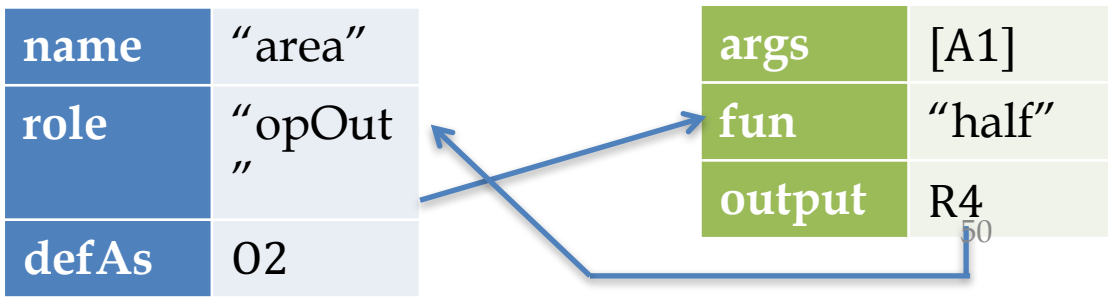
h, w are registers;
mul() generates an operator

```
class f(XManFunctions):
    @staticmethod
    def half(a):
        ...
class Triangle(XMan):
    h = f.input()
    w = f.input()
    area = f.half(h*w)
```



Python code to create a Wengert list

```
class XManFunctions(object):
    @staticmethod
    def input(default=None):
        return Register(role='input',default=default)
    ...
    @staticmethod
    def mul(a,b):
        return XManFunctions.registerDefinedByOperator('mul',a,b)
    ...
    @staticmethod
    def registerDefinedByOperator(fun,*args):
        reg = Register(role='operationOutput')
        op = Operation(fun,*args)
        reg.definedAs = op
        op.outputReg = reg
        return reg
```



AUTODIFF IN EVEN MORE DETAIL

More detail: xman.py

```
class Operation(object):
    """ An operation encodes a single step of a differentiable
    computation, eg y=f(x1,...,xk). It contains a function, arguments,
    and a pointer to the register that is defined as the output of
    this operation.
    """

    def __init__(self,fun,*args):
        self.fun = fun
        self.args = args
        self.outputReg = None

    def asStringTuple(self):
        """ Return a nested tuple of encoding the operation y=f(x1,...xk) of
        the form (y,f,(x1,...,xk)) where y,f, and x1...xk are strings.
        """
        dstName = self.outputReg.name if (self.outputReg and self.outputReg.name) else "???"
        argNames = map(lambda reg:reg.name or "???", self.args)
        return (dstName,self.fun,argNames)

    def __str__(self):
        """ Human readable representation """
        (dstName,fun,argNames) = self.asStringTuple()
        return dstName + " = f." + fun + "(" + ",".join(argNames) + ")"
```

```python
class Register(object):
    """ Registers are like variables - they are used as the inputs to and
    outputs of Operations.  The 'name' of each register should be unique,
    as it will be used as a key in storing outputs, and
    """
    _validRoles = set("input param operationOutput".split())

    def __init__(self,role=None,default=None):
        assert role in Register._validRoles
        self.role = role
        self.name = None
        self.definedAs = None
        self.default = default

    def inputsTo(self):
        """ Trace back through the definition of this register, if it exists,
        to find a list of all other registers that this register
        depends on.  This method is needed to find the
        operationSequence that is needed to construct the value of a
        register, and also to assign names to otherwise unnamed
        registers.
        """
        if self.definedAs:
            assert isinstance(self.definedAs,Operation)
            return self.definedAs.args
        else:
            return []
    # operator overloading
    def __add__(self,other):
        return XManFunctions.add(self,other)
    def __sub__(self,other):
        return XManFunctions.subtract(self,other)
    def __mul__(self,other):
        return XManFunctions.mul(self,other)
```

```python
class XManFunctions(object):
    """ Encapsulates the static methods that are used in a subclass of
    XMan.  Each of these generates an OperationOutput register that is
    definedBy an Operation, with the operations outputReg field
    pointing back to the register.

    You will usually subclass this so you can
    add your own functions, and give the subclass
    a short name

    """

    @staticmethod
    def input(default=None):
        return Register(role='input',default=default)
    @staticmethod
    def param(default=None):
        return Register(role='param',default=default)

    @staticmethod
    def add(a,b):
        return XManFunctions.registerDefinedByOperator('add',a,b)
    @staticmethod
    def subtract(a,b):

    ...


    @staticmethod
    def registerDefinedByOperator(fun,*args):

        ...
```

name	"area"
role	"opOut"
defAs	02

args	[A1]
fun	"half"
output	R4

```python
class XMan(object):

    def __init__(self):
        self._nextTmp = 1
        self._setupComplete = False
        self._registers = {}

    def setup(self):
        """ Must call this before you do any other operations with an
        expression manager """
        # use available python variable names for register names
        for regName,reg in self.namedRegisterItems():
            if not reg.name:
                reg.name = regName
                self._registers[regName] = reg
        # give names to any other registers used in subexpressions
        def _recursivelyLabelUnnamedRegisters(reg):
            if not reg.name:
                reg.name = 'z%d' % self._nextTmp
                self._nextTmp += 1
                self._registers[reg.name] = reg
            for child in reg.inputsTo():
                _recursivelyLabelUnnamedRegisters(child)
        for regName,reg in self.namedRegisterItems():
            _recursivelyLabelUnnamedRegisters(reg)
        self._setupComplete = True
        # convenient return value so we can say net = FooNet().setup()
        return self

 def namedRegisterItems(self):
     """ Returns a list of all pairs (name,registerObject) where some
     python class/instance variable with the given name is bound to
     a Register object.  These are sorted by name to make tests
     easier.
     """
     keys = sorted(self.__dict__.keys() + self.__class__.__dict__.keys())
     vals = [self.__dict__.get(k) or self.__class__.__dict__.get(k) for k in keys]
     return filter(lambda (reg,regObj):isinstance(regObj,Register), zip(keys,vals))
```

```python
class XMan(object):

    ...

    def operationSequence(self,reg):
        """ Traverse the expression tree to find the sequence of operations
        needed to compute the values associated with this register.
        """
        assert self._setupComplete, 'operationSequence() called before setup()'
        # pre-order traversal of the expression tree
        previouslyDefined = set()
        buf = []
        for child in reg.inputsTo():
            if child.name not in previouslyDefined:
                buf += self.operationSequence(child)
                previouslyDefined.add(child.name)
        if reg.definedAs and (not reg.name in previouslyDefined):
            buf.append(reg.definedAs.asStringTuple())
        return buf
```

Using xman.py

```
from xman import *

# functions I'll use for this problem

class f(XManFunctions):
    @staticmethod
    def half(a):
        return XManFunctions.registerDefinedByOperator('half',a)
    @staticmethod
    def square(a):
        return XManFunctions.registerDefinedByOperator('square',a)
    @staticmethod
    def alias(a):
        """ This will just make a copy of a register that
        has a different name."""
        return XManFunctions.registerDefinedByOperator('alias',a)
```

Using xman.py

```
EVAL_FUNS = {
    'add':      lambda x1,x2: x1+x2,
    'subtract': lambda x1,x2: x1-x2,
    'mul':      lambda x1,x2: x1*x2,
    'half':     lambda x: 0.5*x,
    'square':   lambda x: x*x,
    'alias':    lambda x: x,
    }

BP_FUNS = {
    'add':      [lambda delta,out,x1,x2: delta,    lambda delta,out,x1,x2: delta],
    'subtract': [lambda delta,out,x1,x2: delta,    lambda delta,out,x1,x2: -delta],
    'mul':      [lambda delta,out,x1,x2: delta*x2, lambda delta,out,x1,x2: delta*x1],
    'half':     [lambda delta,out,x: delta*0.5],
    'square':   [lambda delta,out,x: delta*2*x],
    'alias':    [lambda delta,out,x: delta],
    }
```

Using xman.py

```
class Autograd(object):
    """ Automatically compute partial derivatives.
    """

    def __init__(self,xman):
        self.xman = xman

    def eval(self,opseq,valueDict):
        """ Evaluate the function specified by the wengart list (aka operation
        sequence). Here valueDict is a dict holding the values of any
        inputs/parameters that are needed (indexed by register name,
        which is a string).  Returns the augmented valueDict.
        """
        for (dstName,funName,inputNames) in opseq:
            inputValues = map(lambda a:valueDict[a], inputNames)
            fun = EVAL_FUNS[funName]
            result = fun(*inputValues)
            valueDict[dstName] = result
        return valueDict
```

Using xman.py

```
class Autograd(object):
    """ Automatically compute partial derivatives.
    """


    def bprop(self,opseq,valueDict,**deltaDict):
        """ For each intermediate register g used in computing the function f
        computed by the operation sequence, find df/dg.  Here
        valueDict is a dict holding the values of any
        inputs/parameters that are needed for the gradient (indexed by
        register name), as populated by eval.
        """
        for (dstName,funName,inputNames) in reversed(opseq):
            delta = deltaDict[dstName]
            # values is extended to include the next-level delta and
            # the function output, and these will be passed as
            # arguments
            values = [delta] + map(lambda a:valueDict[a], [dstName]+list(inputNames))
            for i in range(len(inputNames)):
                result = (BP_FUNS[funName][i])(*values)
                # increment a running sum of all the delta's that are
                # pushed back to the i-th parameter, initializing to
                # zero if needed.
                self._incrementBy(deltaDict, inputNames[i], result)
        return deltaDict
```

Using xman.py – like Guinea Pig

```
def House():
    """ Expression manager for a toy task that has parameters and a loss function.
    First you compute the area of a simple shape, a 'house',
    which is a triangle on top of a rectangle.
    """

    # define some macros
    def roofHeight(wallHeight):
        return f.half(wallHeight)
    def triangleArea(h,w):
        return f.half(h*w)
    def rectArea(h,w):
        return h*w

    #create the instance
    x = XMan()

    # height and width of rectangle are inputs
    x.h = f.param(default=30.0)
    x.w = f.param(default=20.0)
    # so is the target height and the target area, these inputs have
    # defaults
    x.targetArea = f.input(default=0.0)
    x.targetHeight = f.input(default=8.0)
    x.heightFactor = f.input(default=100.0)

    # compute area of the house
    x.area = rectArea(x.h,x.w) + triangleArea(roofHeight(x.h), x.w)

    # loss to optimize is weighted sum of square loss of area relative
    # to the targetArea, plus same for height
    x.loss = f.square(x.area - x.targetArea) + f.square(x.h - x.targetHeight) * x.heightFactor

    return x
```

```python
# build your dream house with gradient descent

h = House().setup()
autograd = Autograd(h)
rate = 0.001
epochs = 20

# this fills in default values for inputs with defaults, like
# targetHeight and heightFactor
initDict = h.inputDict(h=5,w=10,targetArea=200)

# form wengart list to compute the loss
opseq = h.operationSequence(h.loss)

for i in range(epochs):


    # find gradient of loss wrt parameters
    gradientDict = autograd.bprop(opseq,valueDict,loss=1.0)

    # update the parameters appropriately
    for rname in gradientDict:
        if h.isParam(rname):
            initDict[rname] = initDict[rname] - rate*gradientDict[rname]
```

An Xman instance and a loop

```
def Skyscraper(numFloors):
    """ Another toy task - optimize area of a stack of several rectangles
    """

    x = XMan()
    # height and width of rectangle are inputs
    x.h = f.param(default=30.0)
    x.w = f.param(default=20.0)
    x.targetArea = f.input(default=0.0)
    x.targetHeight = f.input(default=8.0)
    x.heightFactor = f.input(default=100.0)

    # compute area of the skyscraper
    x.zero = f.input(default=0.0)

    # here areaRegister popints to a different
    # register in each iteration of the loop
    areaRegister = x.zero
    for i in range(numFloors):
        floorRegister = (x.h * x.w)
        floorRegister.name = 'floor_%d' % (i+1)
        areaRegister = areaRegister + floorRegister
    # when the loop finishes, we give it the register a name, in this
    # case by having an instance variable point to the register.
    # we could also execute: areaRegister.name = 'area'
    x.area = areaRegister

    x.loss = f.square(x.area - x.targetArea) + f.square(x.h - x.targetHeight) * x.heightFactor
    return x
```