

CSE 559A: Computer Vision



Fall 2018: T-R: 11:30-1pm @ Lopata 101

Instructor: Ayan Chakrabarti (ayan@wustl.edu).

Course Staff: Zhihao Xia, Charlie Wu, Han Liu

<http://www.cse.wustl.edu/~ayan/courses/cse559a/>

November 6, 2018

GENERAL

- Look at Proposal Feedback
- PSET 4 Due Next Tuesday
- Recitation will be this Friday (Lopata 103, 10:30-Noon)

1

2

AUTOGRAD

Building Our Own Deep Learning Framework

- Computation Graph that encodes symbolic relationship between input to output (and loss)
- Nodes in this graph are
 - Values: Values are set from training data
 - Params: Values are initialized and updated using SGD
 - Operations: Values are computed as functions of their input
- Forward computation to compute loss
- Backward computation to compute gradients for every node

AUTOGRAD

Code from pset5/mnist.py

```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()

W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp,W1)
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```

3

4

BRIEF DETOUR

- What are RELUs ?

Element-wise non-linear activations

$$\text{RELU}(x) = \max(0, x)$$

- Previous activations would be sigmoid-like: $\sigma(x) = \exp(x)/(1 + \exp(x))$
 - Great when you want a probability, bad when you want to learn by gradient descent.
 - For both high and low-values of x , $\partial\sigma(x)/\partial x = 0$
 - So if you weren't careful, you would end up with high-magnitude activations, and the network stops learning.

Gradient descent is **very fragile** !

BRIEF DETOUR

- What are RELUs ?

$$\text{RELU}(x) = \max(0, x)$$

What is $\partial \text{RELU}(x) / \partial x$?

0 if $x < 0$, 1 otherwise.

- So your gradients are passed un-changed if the input is positive.
- But completely attenuated if the input is negative
- So there's still the possibility of your optimization getting stuck, if you reach a point where all inputs to the RELU are negative.
- Will talk about initialization, etc. later.

5

AUTOGRAD

Code from pset5/mnist.py

```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()

W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp,W1)
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```

AUTOGRAD

Code from pset5/edf.py

```
ops = []; params = []; values = []
...
class Param:
    def __init__(self):
        params.append(self)
...
class Value:
    def __init__(self):
        values.append(self)
...
class matmul:
    def __init__(self,x,y):
        ops.append(self)
        self.x = x
        self.y = y
```

When you construct an object, it just does book-keeping !

7

6

8

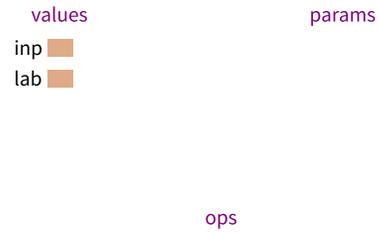
AUTOGRAD

```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()

W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp,W1)
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```



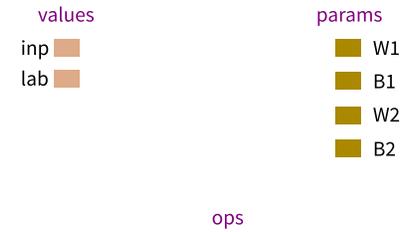
AUTOGRAD

```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()

W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp,W1)
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```



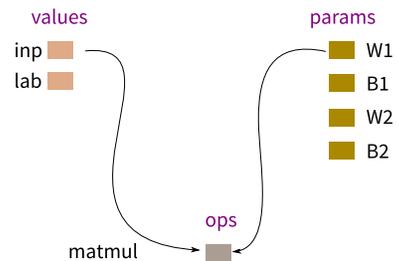
AUTOGRAD

```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()

W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp,W1)
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```



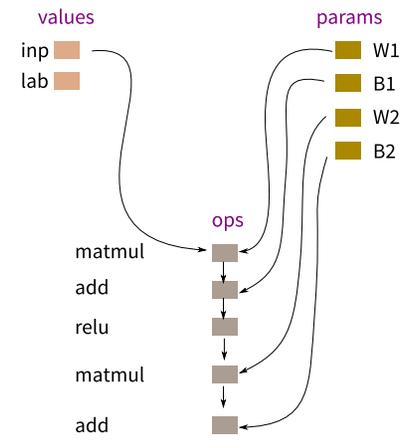
AUTOGRAD

```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()

W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp,W1)
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```



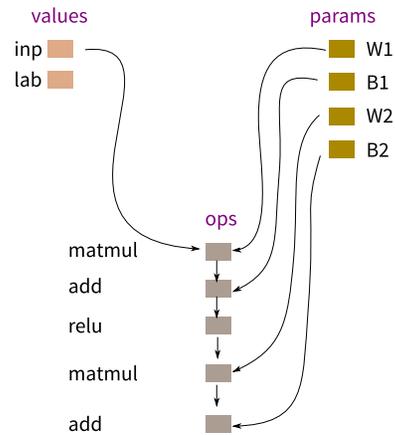
AUTOGRAD

```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()

W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp,W1)
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```



At this point, we've just defined the graph:
no actual computation has happened !

AUTOGRAD

```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()
W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()
y = edf.matmul(inp,W1)
y = edf.add(y,B1)
y = edf.RELU(y)
y = edf.matmul(y,W2)
y = edf.add(y,B2)

loss = edf.smaxloss(y,lab)
loss = edf.mean(loss)

acc = edf.accuracy(y,lab)
acc = edf.mean(acc)
```

- mean will be across a batch.
- accuracy is actual accuracy of hard predictions (not differentiable).

19

20

AUTOGRAD

Now let's train this thing !

Beginning of training:

```
nHidden = 1024; K = 10
W1.set(xavier((28*28,nHidden)))
B1.set(np.zeros((nHidden)))
W2.set(xavier((nHidden,K)))
B2.set(np.zeros((K)))
```

Initialize weights randomly.

In each iteration of training:

```
for iters in range(...):
    . . .
    inp.set(train_im[idx[b:b+BSZ],:])
    lab.set(train_lb[idx[b:b+BSZ]])
    . . .
```

Load data into the 'values' or inputs.

What is this set function anyway ?

AUTOGRAD

set is the **only** function that the classes param and value have.

```
class Value:
    def __init__(self):
        values.append(self)

    def set(self,value):
        self.top = np.float32(value).copy()

class Param:
    def __init__(self):
        params.append(self)

    def set(self,value):
        self.top = np.float32(value).copy()
```

Sets a member called "top" to be an array that holds these values.

21

22

AUTOGRAD

```
W1.set(xavier((28*28,nHidden)))
...
B2.set(np.zeros((K)))

for iters in range(...):
    inp.set(train_im[idx[b:b+BSZ],:])
    lab.set(train_lb[idx[b:b+BSZ]])
```

- Note that we are loading our input data in batches as matrices.
- inp is BSZ x N
- Then we're doing a matmul with W1 which is N x nHidden.
- Output will be BSZ x nHidden
- Essentially, we're replacing vector matrix multiply for a single sample with matrix-matrix multiply for a batch of samples.

23

AUTOGRAD

```
W1.set(xavier((28*28,nHidden)))
...
B2.set(np.zeros((K)))

for iters in range(...):
    inp.set(train_im[idx[b:b+BSZ],:])
    lab.set(train_lb[idx[b:b+BSZ]])

    edf.Forward()
    print(loss.top,acc.top)
```

And this will work. It will print the loss and accuracy values for the set inputs, given the current value of the parameters.

What is this magical function forward ?

24

AUTOGRAD

From edf.py

```
# Global forward
def Forward():
    for c in ops: c.forward()
```

But the operation classes have their own forward function.

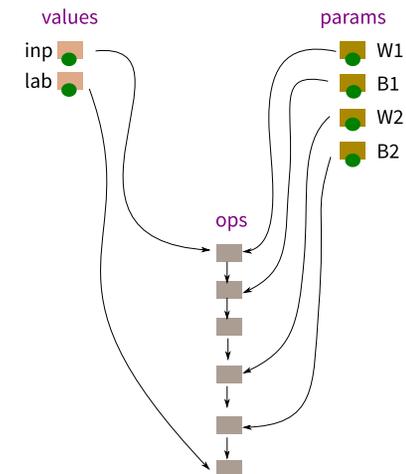
```
class matmul:
    def __init__(self,x,y):
        ops.append(self)
        self.x = x
        self.y = y

    def forward(self):
        self.top = np.matmul(self.x.top,self.y.top)

    ...
```

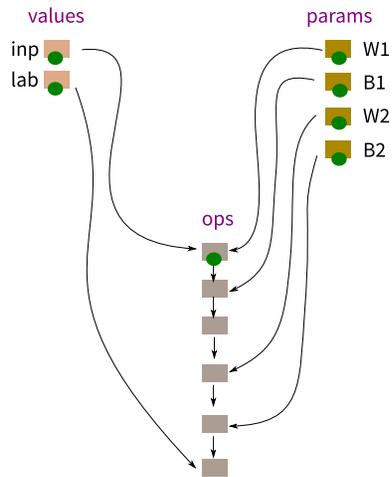
25

AUTOGRAD



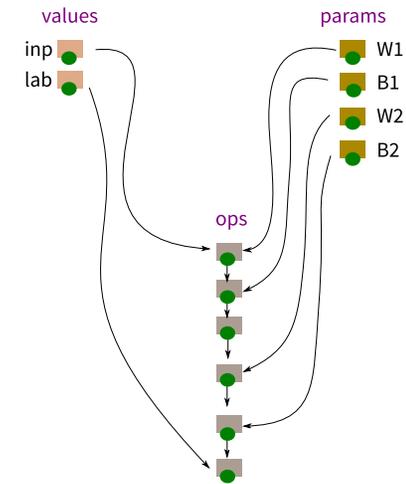
28

AUTOGRAD



29

AUTOGRAD



34

AUTOGRAD

- So the forward pass computes the loss.
- But we want to learn the parameters.

```
for iters in range(...):  
    . . .  
    inp.set(train_im[idx[b:b+BSZ],:])  
    lab.set(train_lb[idx[b:b+BSZ]])  
  
    edf.Forward()  
    print(loss.top, acc.top)  
    edf.Backward(loss)  
    edf.SGD(lr)
```

- The SGD function is pretty simple

```
def SGD(lr):  
    for p in params:  
        p.top = p.top - lr*p.grad
```

Requires `p.grad` (gradients with respect to loss) to be present.

That's what backward does!

35

AUTOGRAD

From `edf.py`

```
# Global backward  
def Backward(loss):  
    for c in ops:  
        c.grad = np.zeros_like(c.top)  
    for c in params:  
        c.grad = np.zeros_like(c.top)  
  
    loss.grad = np.ones_like(loss.top)  
    for c in ops[::-1]: c.backward()
```

- Called with an `op` object as the loss.
- Initializes all gradients to zero.
- `x.grad` is defined as the gradient of the loss wrt to `x.top`
- And so, initializes `loss.grad` to all ones.
- Each `op` has a `backward` function. Calls this in reverse order.

36

AUTOGRAD

```
# Matrix multiply (fully-connected layer)
class matmul:
    def __init__(self,x,y):
        ops.append(self)
        self.x = x
        self.y = y

    def forward(self):
        self.top = np.matmul(self.x.top,self.y.top)

    def backward(self):
        if self.x in ops or self.x in params:
            self.x.grad = self.x.grad +
                np.matmul(self.y.top,self.grad.T).T
        if self.y in ops or self.y in params:
            self.y.grad = self.y.grad +
                np.matmul(self.x.top.T,self.grad)
```

- The backward function **adds** to the gradients of its inputs. Those inputs could've been used by other ops.
- Computes gradients of its inputs based on the value of its own gradients (i.e., of its output).
- Assumes that by the time backward is called on it, self.grad will exist and be final. Why ?

37

AUTOGRAD

```
# Matrix multiply (fully-connected layer)
class matmul:
    def __init__(self,x,y):
        ops.append(self)
        self.x = x
        self.y = y

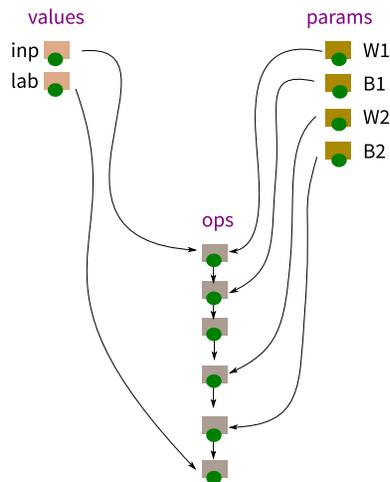
    def forward(self):
        self.top = np.matmul(self.x.top,self.y.top)

    def backward(self):
        if self.x in ops or self.x in params:
            self.x.grad = self.x.grad +
                np.matmul(self.y.top,self.grad.T).T
        if self.y in ops or self.y in params:
            self.y.grad = self.y.grad +
                np.matmul(self.x.top.T,self.grad)
```

- Assumes that by the time backward is called on it, self.grad will exist and be final. Because anything that could add to its gradient would have been called before it.
- Only computes grads for params and ops (not values).

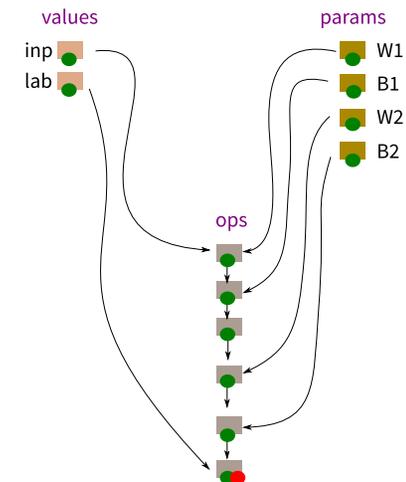
38

AUTOGRAD



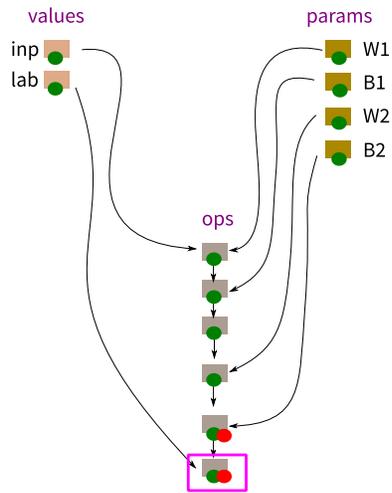
39

AUTOGRAD



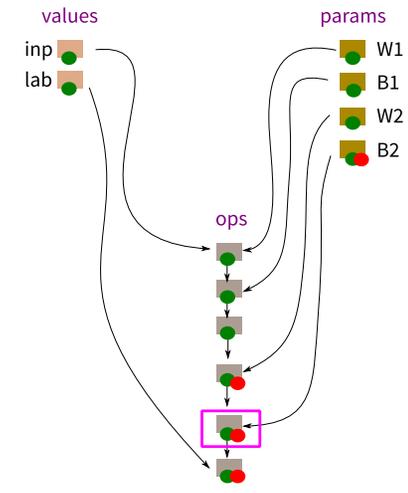
40

AUTOGRAD



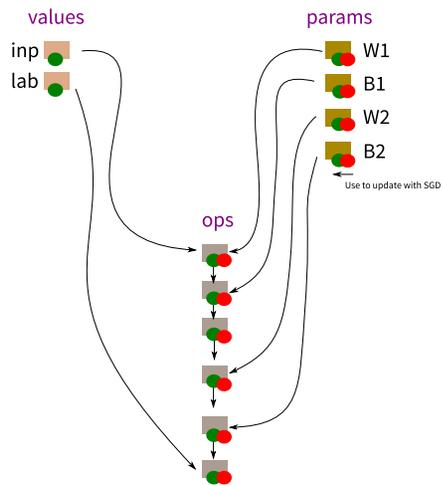
41

AUTOGRAD



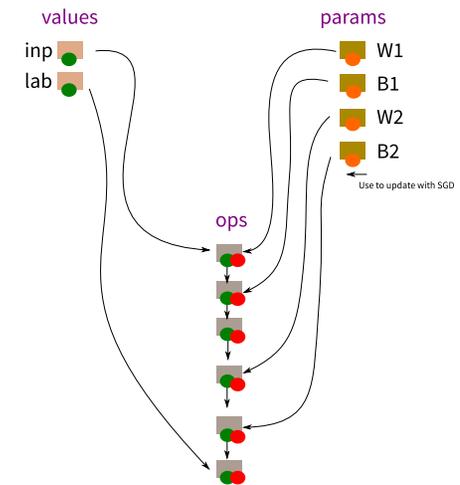
42

AUTOGRAD



47

AUTOGRAD



48

AUTOGRAD

- Allows us to **specify** a complex estimation function as a composition of simpler functions.
- Need to just provide an 'implementation' for each function so that we know how to:
 - Compute outputs given inputs
 - Compute contribution to gradients of inputs given gradient of output
- Given this model, can allow us to compute gradients of all parameters automatically!

SPATIAL OPERATIONS

```
y = edf.matmul(inp,W1)
. . .
for iters in range(...):
. . .
    inp.set(train_im[idx[b:b+BSZ],:])
    lab.set(train_lb[idx[b:b+BSZ]])
. . .
```

- So far, even though our inputs were images, we were treating them as vectors.
- inp is a matrix of size $B \times N$, where B is the batch-size, and N was the number of pixels (28^2 for MNIST).
- So each sample is represented by a row vector of size N .
- This means that parameters of the first layer $W1 = N \times H$, where H is the dimensionality of the encoding.
- That's a lot of weights, especially as you go to "real" images.
- We can't solve this by reducing H . Because then we would be assuming that there is a "linear" function that can reduce dimensionality and keep information required for classification.
- Need to do this slowly.

49

SPATIAL OPERATIONS

Instead, consider our input to be what it is, an image!

- Work with 4-D arrays of size $B \times H \times W \times C$ instead of matrices.
- Use convolutional layers, which produce multi-channel output images from multi-channel input images, except that each output pixel only depends on a small number of input pixels in its neighborhood.
 - And this dependence is translation invariant.

$$g[b, y, x, c_2] = \sum_{k_y} \sum_{k_x} \sum_{c_1} f[b, y + k_y, x + k_x, c_1] k[k_y, k_x, c_1, c_2]$$

(Note that this is actually 'correlation' not convolution)

In the problem set, you'll have to implement forward and backward.

SPATIAL OPERATIONS

```
y = edf.conv2(inp,K1)
y = edf.down2(y); y = edf.down2(y);
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.flatten(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```

50

51

52

SPATIAL OPERATIONS

```
# Downsample by 2
class down2:
    def __init__(self,x):
        ops.append(self)
        self.x = x

    def forward(self):
        self.top = self.x.top[:,::2,::2,:]

    def backward(self):
        if self.x in ops or self.x in params:
            grd = np.zeros_like(self.x.top)
            grd[:,::2,::2,:] = self.grad
            self.x.grad = self.x.grad + grd
```

This is actually a huge waste of computation ! We're computing values and then throwing them away.

SPATIAL OPERATIONS

```
y = edf.conv2(inp,K1)
y = edf.down2(y); y = edf.down2(y);
y = edf.add(y,B1)
y = edf.RELU(y)

y = edf.flatten(y)

y = edf.matmul(y,W2)
y = edf.add(y,B2) # This is our final prediction
```

53

54

SPATIAL OPERATIONS

```
# Flatten (conv to fc)
class flatten:
    def __init__(self,x):
        ops.append(self)
        self.x = x

    def forward(self):
        self.top = np.reshape(self.x.top, [self.x.top.shape[0], -1])

    def backward(self):
        if self.x in ops or self.x in params:
            self.x.grad = self.x.grad
            + np.reshape(self.grad, self.x.top.shape)
```

55

OTHER OPERATIONS

View 1

- Neural network is a collection of neurons that mimic the brain

View 2

- Neural networks are programs with differentiable operations acting on inputs and trainable parameters.
- Variety of "basic" operations possible: as long as they're (at least partially) differentiable
- Express more complex operations in terms of these basic operations
- Design your network by writing a program
 - That you think can express the required computation to solve a problem (would work if you just have the right parameters).
 - Has a healthy gradient flow so that it can learn those parameters.

56

OTHER OPERATIONS

Conditional Operations

- Consider edf.max which implements $z = \max(x, y)$
- Its forward function is very simple:
 - `self.top = np.maximum(self.x.top, self.y.top)`
- What about backward ?

$$\begin{aligned}\nabla_x &= \nabla_z \text{ if } x \geq y, 0 \text{ otherwise} \\ \nabla_y &= \nabla_z \text{ if } y > x, 0 \text{ otherwise}\end{aligned}$$

- So we back-propagate to the input we picked.
- But don't back-propagate the "condition"

57

EAGER EXECUTION

- Conditionals are tricky
 - You are often executing both execution paths of the condition and picking one.
 - Can get really hairy when the condition is to setup a loop (recurrent networks)
- Essentially, your computation graph must contain all possible execution paths, with the condition "masking" which values end up in your final solution.
- This can be inefficient.
- What if you added only the ops that are required based on the condition ?

But we don't know the values of the conditions when we construct the graph!

59

OTHER OPERATIONS

Conditional Operations

- Now consider edf.where which implements $z = (p \geq q)x + (q > p)y$
 - Selects from two inputs based on relative value of two other inputs.

$$\begin{aligned}\nabla_x &= \nabla_z \text{ if } p \geq q, 0 \text{ otherwise} \\ \nabla_y &= \nabla_z \text{ if } q > p, 0 \text{ otherwise}\end{aligned}$$

- No gradients generated for the condition

$$\nabla_p = \nabla_q = 0$$

- Can still be useful
 - But if you hope to learn parts of the network generating p and q , they must have gradients coming from elsewhere.

58

EAGER EXECUTION

But we don't know the values of the conditions when we construct the graph!

Solution: Eager Execution (PyTorch, now in Tensorflow)

- Construct a separate graph for each iteration of training
- Params are initialized before training. Inputs are called with their values in each iteration.
- Object constructor for each operation node also calls forward
- You can therefore check the value of top as you are constructing the graph to decide which nodes to add.
- Once you have the final output, just call backward
 - Will compute gradients for all parameters (and operations) used in that iteration.
- Afterwards, delete the graph and start anew for the next iteration.

60