CSE 559A: Computer Vision

Fall 2017: T-R: 11:30-1pm @ Lopata 101

Instructor: Ayan Chakrabarti (ayan@wustl.edu).
Staff: Abby Stylianou (abby@wustl.edu), Jarett Gross (jarett@wustl.edu)

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

Nov 16, 2017
• PSET 5 Posted.
• One day extension (now due on the Friday two weeks from now)
• You get to implement SLIC, and your own conv layer!
  ▪ and work on the "world famous" MNIST data set.
• Has extra credit question---but prioritize your projects.
• Look through the entire code!
• Regular Office Hours Tomorrow.
• Recitation Monday (9/20) Evening at 5:30pm
  ▪ In Jolley 420
• Wed Morning Office Hours as usual
# 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
When you construct an object, it just does book-keeping!

```python
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
```
# 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
# Inputs and parameters

```python
inp = edf.Value()
lab = edf.Value()
```

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

# Model
```python
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
```
# 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
# 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)
\[ y = \text{edf.add}(y, B1) \]
\[ y = \text{edf.RELU}(y) \]
\[ y = \text{edf.matmul}(y, W2) \]
\[ y = \text{edf.add}(y, B2) \] # This is our final prediction
# 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
# 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)
\textbf{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
# 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!
# 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)
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?

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

But this can saturate too!

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

0 if \( x < 0 \), 1 otherwise.
BRIEF DETOUR

- What are RELUs?

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

But this can saturate too!

If all your \(x\)'s are negative.

- Will talk about batch-norm later, but we try to make sure all activations are zero-mean.
- Normalize inputs to be zero mean unit variance.
- Initialize each layer's weights randomly, such that if its inputs are zero mean unit variance, so are its outputs in expectation.
- And then, learn very very slowly.
What is soft-max?

We are now doing multi-class prediction. Instead of binary, our answers are one of $K$ classes.

Binary case: produce a single number and checking if it is $> 0$ or $< 0$

$K$ Class Case: produce a vector of $K$ numbers, and do $\arg \max$. We say $k$ is the right answer, if the $k$th element of this vector is the highest. (soft-max is a soft version of $\arg \max$, not max)

Binary Case: Predicted log-odds

Multi-class Case: Predict un-normalized log probabilities
BRIEF DETOUR

- What is soft-max?

\[ y = f(x) \in \mathbb{R}^K \]

- Predict class \( k \) if \( y^k \geq y^{k'} \forall k' \)

Here \( y^k \) refers to the \( k \)th element of \( y \), not an exponent

\[
P(\text{class} = k) = p^k = \frac{\exp(y^k)}{\sum_{k'} \exp(y^{k'})}
\]

- Cross Entropy Loss

\[
L(C, y) = -\log p^C = \log(\sum_{k'} \exp(y^{k'}) - \log \exp y^C
\]

for true \( C \in \{0, 2, \ldots K - 1\} \)
BRIEF DETOUR

- What is soft-max?

\[ y = f(x) \in \mathbb{R}^K \]

- Predict class \( k \) if \( y^k \geq y^{k'} \; \forall k' \)

\[ P(\text{class} = k) = p^k = \frac{\exp(y^k)}{\sum_{k'} \exp(y^{k'})} \]

- Cross Entropy Loss

\[ L(C, y) = -\log p^C = \log \left( \sum_{k'} \exp y^{k'} \right) - y^C \]

What is \( \frac{\partial L}{\partial y^k} \)? For \( k = c \) and \( k \neq c \)?

\[ \frac{\partial L}{\partial y^k} = p^k \text{ if } k \neq C \]

\[ \frac{\partial L}{\partial y^C} = p^C - 1 \]
What is soʿ-max?

- Predict class \( k \) if \( y^k \geq y^{k'} \) \( \forall k' \)

\[
y = f(x) \in \mathbb{R}^K
\]

\[
L(C, y) = -\log p^C = \log(\sum_{k'} \exp y^{k'}) - y^C
\]

\[
\frac{\partial L}{\partial y^k} = p^k \text{ if } k \neq C
\]

\[
\frac{\partial L}{\partial y^C} = p^C - 1
\]

Similar form to binary case: \( p^k - p_{GT}^k \)!

Soft-max is not an element-wise non-linearity. True class \( C \) creates gradients for all classes.

You'll want to implement soft-max + cross-entropy usually as a joint "layer" for numerical stability and efficiency.
# 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).
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?
set is the **only** function that the classes param and value have.

```python
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.
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?
From edf.py

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

But the operation classes have their own forward function.

```python
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)

    ...
```
AUTOGRAD

values
inp
lab

ops

params
W1
B1
W2
B2
AUTOGRAD
So the forward pass computes the loss.
But we want to learn the parameters.

```python
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

```python
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!
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.
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.

Assumes that by the time `backward` is called on it, `self.grad` will exist and be final. Why?

```python
# 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.
- Assumes that by the time `backward` is called on it, `self.grad` will exist and be final. Why?
# 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).
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!
CONVOLUTIONAL NETWORKS

- 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 $W_1 = N \times H$, 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.

```python
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]])
    ...
```
CONVOLUTIONAL NETWORKS

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.

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

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

In the problem set, you'll have to implement forward and backward.
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
This is actually a huge waste of computation! We're computing values and then throwing them away.
# 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)
CONVOLUTIONAL NETWORKS

Still to come

- Actual full network models and tasks.
- Regularization: Data augmentation, early stopping, drop out
- Other optimization techniques: batchnorm, Adam, etc.
- Handling Vanishing Gradients: Residual Networks!