CSE 559A: Computer Vision

Fall 2020: T-R: 11:30-12:50pm @ Zoom

Instructor: Ayan Chakrabarti (ayan@wustl.edu).
Course Staff: Adith Boloor, Patrick Williams

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

Dec 1, 2020
• Please look at proposal feedback if you haven’t yet.
# 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)

- Defines computation graph, rather than perform operations.
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
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)

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()
# Global forward

def Forward():
    for c in ops: c.forward()

...  

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)

...
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!
From edf.py

```python
# 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.
Let us say the forward function computes an output array as some function of an input array:

\[ f : \{x[l, m, n]\} \rightarrow \{y[i, j, k]\} \]

When that layer’s backward function is called, `self.grad` will be available, defined as:

\[ \text{self.grad} = \nabla_y L \quad (\nabla_y L)[i, j, k] = \frac{\partial L}{\partial y[i, j, k]} \]

The backward function then has to update the gradient of its input(s):

\[ \text{self.x.grad} = \nabla_x L \quad (\nabla_x L)[l, m, n] = \frac{\partial L}{\partial x[l, m, n]} \]

From the chain rule, each element of this gradient is given by:

\[ \frac{\partial L}{\partial x[l, m, n]} + = \sum_i \sum_j \sum_k \frac{\partial y[i, j, k]}{\partial x[l, m, n]} \times \frac{\partial L}{\partial y[i, j, k]} \]

This is an addition because that same input could have been used by other ops.
The ba’kwar’ “un’tion adds to the “ra’ients o” its inputs. Those inputs ‘oul’ɉve been use’ by other ops.

Computes “ra’ients o” its inputs base’ on the value o” its own “ra’ients (i.e., o” its output).

Assumes that by the time ba’kwar’ is ‘alle’ on it, sel”.ra’ will exist an’ be “inal. 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)

• 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 ?
Assumes that by the time \texttt{backwards} is called on it, \texttt{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).
values

inp
lab

ops

params

W1
B1
W2
B2
AUTOGRAD
AUTOGRAD

values
inp
lab

params
W1
B1
W2
B2

ops
AUTOGRAD

values
inp
lab

params
W1
B1
W2
B2

ops
• 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!
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.
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.
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
# 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.
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
# 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)
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.
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?

\[
\nabla_x = \nabla_z \text{ if } x \geq y, 0 \text{ otherwise}
\]
\[
\nabla_y = \nabla_z \text{ if } y > x, 0 \text{ otherwise}
\]

- So we back-propagate to the input we picked.
- But don’t back-propagate the “condition”
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.
  \[
  \nabla_x = \nabla_z \text{ if } p \geq q, \quad 0 \text{ otherwise} \\
  \nabla_y = \nabla_z \text{ if } q > p, \quad 0 \text{ otherwise}
  \]

• 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.
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!
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.
CORE SEMANTIC TASKS IN VISION

- Image “Classification”
- Object Detection
- Semantic Segmentation
**Image Classification:** Given an image, classify it as being one of N categories
Image Classification: Given an image, classify it as being one of N categories

Scene Type
**Image Classification:** Given an image, classify it as being one of N categories

**Scene Type**

Aquarium
**Image Classification**: Given an image, classify it as being one of $N$ categories

*Scene Type*

*Art Gallery*
**Image Classification:** Given an image, classify it as being one of N categories

*Scene Type*

*Army Base*
**CORE SEMANTIC TASKS IN VISION**

*Image Classification*: Given an image, classify it as being one of \( N \) categories

Objects in Scenes

| arille | mushroom | cherry | Madagascar cat |
**Image Classification:** Given an image, classify it as being one of N categories

Objects in Scenes

But there might be more than one 'obvious' answer for an image
Detection: Check if an object exists in an image, and if so, draw a box around it.
**Detection:** Check if an object exists in an image, and if so, draw a box around it.
**Detection**: Check if an object exists in an image, and if so, draw a box around it.

Output: List of $[x_1, x_2, y_1, y_2]$ co-ordinates + class labels.
CORE SEMANTIC TASKS IN VISION

Detection: Check if an object exists in an image, and if so, draw a box around it.

But a box is too coarse ...
**Semantic Segmentation:** Label the class of each pixel.
Semantic Segmentation: Label the class of each pixel.
CORE SEMANTIC TASKS IN VISION

Semantic Segmentation: Label the class of each pixel.

Instance Segmentation: Separate pixels of the same class into instances.
Human Pose Estimation

Generally, detect parts or keypoints of objects.
Face Recognition

Determine identity by matching input image to individual in database.

N-way classification
N = no of people in the database
Face / Person Re-identification
Face / Person Re-identification

Determine that two images are of the same person
CORE SEMANTIC TASKS IN VISION

Face / Person Re-identification

Determine that two images are of the same person
Recognize Expressions from Faces
CORE SEMANTIC TASKS IN VISION

Action / Activity Recognition from Videos
Action / Activity Recognition from Videos
New Semantic Tasks: Many at the Intersection of Vision and Language
New Semantic Tasks: Many at the Intersection of Vision and Language

The man at bat readies to swing at the pitch while the umpire looks on.

A large bus sitting next to a very tall building.

Image Captioning (from COCO challenge)
New Semantic Tasks: Many at the Intersection of Vision and Language

Visual Question Answering (visualqa.org)