CSE 559A: Computer Vision

Fall 2019: T-R: 11:30-12:50pm @ Hillman 60

Instructor: Ayan Chakrabarti (ayan@wustl.edu).
Course Staff: Arushee Agrawal, Annie Lee, Jiahao Li, Xiaochen Zhou

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

Nov 26, 2019
BACK-PROPAGATION REVISITED

- Chain Rule: for scalar variables

\[ y = f(x) \quad \text{and} \quad L = g(y) \]

\[ \frac{\partial L}{\partial x} = \frac{\partial y}{\partial x} \times \frac{\partial L}{\partial y} \]

If \( f(\cdot) \) were a layer with scalar inputs and scalar outputs, then

- \((\partial L/\partial y)\) would be \texttt{self.grad}
- You would be computing \((\partial L/\partial x)\) for \texttt{self.x.grad}.
- But our layers have inputs and outputs that are arrays or tensors.

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

\[
\begin{align*}
\text{self.grad} &= \nabla_y L \\
(\nabla_y L)[i, j, k] &= \frac{\partial L}{\partial y[i, j, k]} \\
\text{self.x.grad} &= \nabla_x L \\
(\nabla_x L)[l, m, n] &= \frac{\partial L}{\partial x[l, m, n]}
\end{align*}
\]
BACK-PROPAGATION REVISITED

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

\[
\begin{align*}
\text{self.grad} &= \nabla_y L \quad (\nabla_y L)[i, j, k] = \frac{\partial L}{\partial y[i, j, k]} \\
\text{self.x.grad} &= \nabla_x L \quad (\nabla_x L)[l, m, n] = \frac{\partial L}{\partial x[l, m, n]} \\
\end{align*}
\]

\[
\frac{\partial L}{\partial x[l, m, n]} + = ?
\]

- The \( + = \) is because you might have other layers backpropagating to \( x \).

Write expressions in terms of the incoming gradients \( \partial L/\partial y[i, j, k] \), and all the partial derivatives of the function \( f \), i.e., \( \partial y[i, j, k]/\partial x[l, m, n] \).
\[ f : \{x[l, m, n]\} \rightarrow \{y[i, j, k]\} \]

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

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

for \(l, m, n\):

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

- But each element of the gradient for \(x\) will contain contributions from all elements of the gradient for \(y\).
- Of course, don't actually want to use for loops (or as few for loops as possible).
Consider the matmul layer

\[ z = x \cdot y \]

\[ z[l, n] = \sum_m x[l, m]y[m, n] \]

Work out the expressions for:

- \( \frac{\partial L}{\partial x[l,m]} \)
- \( \frac{\partial L}{\partial y[m,n]} \)

in terms of the incoming gradient \( (\partial L/\partial z[l,n]) \), and the inputs to the layer \( x[l,m] \) and \( y[m,n] \).
\[
\frac{\partial L}{\partial x[l, m, n]} + \sum_i \sum_j \sum_k \frac{\partial y[i, j, k]}{\partial x[l, m, n]} \cdot \frac{\partial L}{\partial y[i, j, k]}
\]

\[
z[l, n] = \sum_m x[l, m]y[m, n]
\]

Realize for a given \(x[l, m]\), values of \(z[l, n]\) for the same \(l\)

but all \(n\) will depend on that \(x\) (have non-zero partials).

\[
\frac{\partial L}{x[l, m]} = \sum_n \frac{\partial z[l, n]}{\partial x[l, m]} \frac{\partial L}{\partial z[l, n]} = \sum_n y[m, n] \frac{\partial L}{\partial z[l, n]}
\]

\[
\frac{\partial L}{y[m, n]} = \sum_l \frac{\partial z[l, n]}{\partial y[m, n]} \frac{\partial L}{\partial z[l, n]} = \sum_l x[l, m] \frac{\partial L}{\partial z[l, n]}
\]

Both these equations represent matrix multiplications between \(\nabla_z L\) and \(y\) or \(x\). That is how they are implemented in \texttt{edf.py}. 

Of course, convolutional layers are more complex. So let's take a simpler version: average-pooling.

- Average pooling by size $K$:

Input $I$ is a $B \times H \times W \times C$, output $J$ is $B \times (H-K+1) \times (W-K+1) \times C$.

$$J[b, y, x, c] = \frac{1}{K^2} \sum_{y' = 0}^{K-1} \sum_{x' = 0}^{K-1} I[b, y + y', x + x', c]$$

- Remember, much simpler than convolution. No learned kernel, and pooling is independent per channel.
- What about back-propagation?

$$(\nabla_I L)[b, y, x, c] + = \frac{1}{K^2} \sum_{y' = 0}^{\min(K-1,y)} \sum_{x' = 0}^{\min(K-1,x)} (\nabla_J L)[b - y', x - x', c]$$
class avg_pool:
    def __init__(self, x, sz):
        ops.append(self)
        self.x = x
        self.sz = sz

    def forward(self):
        B, H, W, C = self.x.top.shape
        top = np.zeros([B, H-self.sz+1, W-self.sz+1, C])
        for i in range(self.sz):
            for j in range(self.sz):
                xcrop = self.x.top[:, i:(H-self.sz+1+i), j:(W-self.sz+1+j), :]
                top = top + xcrop
        self.top = top / np.float32(self.sz*self.sz)

\[
J[b, y, x, c] = \frac{1}{K^2} \sum_{y' = 0}^{K-1} \sum_{x' = 0}^{K-1} I[b, y + y', x + x', c]
\]

\[
(\nabla_I L)[b, y, x, c] += \frac{1}{K^2} \sum_{y' = 0}^{\min(K-1, y)} \sum_{x' = 0}^{\min(K-1, x)} (\nabla_J L)[b, y - y', x - x', c]
\]
Notice how in backward, we repeat an equivalent version of the loop we did for forward.

Does the same summation: but once you’ve figured out how to parallelize forward, you apply the same idea for backward.

Each statement in backward is adding gradients caused due to corresponding statement in forward.
Avg Pool of size $K$

$$J[b, y, x, c] = \frac{1}{K^2} \sum_{y'=0}^{K-1} \sum_{x'=0}^{K-1} I[b, y + y', x + x', c]$$

Avg Pool of size $K$ and stride $s$

$$J[b, y, x, c] = \frac{1}{K^2} \sum_{y'=0}^{K-1} \sum_{x'=0}^{K-1} I[b, s y + y', s x + x', c]$$
class avg_pool_with_stride:
    def __init__(self, x, sz, stride):
        ops.append(self)
        self.x = x
        self.sz = sz
        self.stride = stride

    def forward(self):
        B, H, W, C = self.x.top.shape
        nH, nW = (H - self.sz) // self.stride + 1, (W - self.sz) // self.stride + 1
        top = np.zeros([B, nH, nW, C])
        for i in range(self.sz):
            for j in range(self.sz):
                xcrop = self.x.top[:, i:(H - self.sz + i):self.stride, j:(W - self.sz + j):self.stride, :]
                top = top + xcrop

        self.top = top / np.float32(self.sz * self.sz)

    def backward(self):
        if self.x in ops or self.x in params:
            B, H, W, C = self.x.top.shape
            xgrad = np.zeros([B, H, W, C])
            for i in range(self.sz):
                for j in range(self.sz):

            self.x.grad = self.x.grad + xgrad
NEURAL NETWORKS FOR PHYSICAL TASKS

- Last time: talked about networks for stereo, monocular depth, and estimating shading / albedo decomposition from an image.
- Shading is a function of both shape and illumination.
Remember in the most general case, object shading is based on an "environment map".

For AR and image-editing applications, we sometimes want to estimate this map to figure out how new objects will appear.

So let's train a neural network that, given an image, predicts this map.

But how do we get ground truth training data?

To the rescue: 360 degree panorama cameras, and datasets collected using such cameras

- SUN360: Xiao et al., CVPR 2012.
ILLUMINATION ESTIMATION

- Outdoor Illumination: Predict position of sun, and relative strength of sun and sky.
ILLUMINATION ESTIMATION

- Outdoor Illumination: Predict position of sun, and relative strength of sun and sky.
ILLUMINATION ESTIMATION

- Outdoor Illumination: Predict position of sun, and relative strength of sun and sky.
ILLUMINATION ESTIMATION

- Outdoor Illumination: Predict position of sun, and relative strength of sun and sky.
ILLUMINATION ESTIMATION

- Indoor Illumination: More complicated, possibly multiple light sources.
- But similar idea: Gardner et al., Learning to Predict Indoor Illumination from a Single Image
ILLUMINATION ESTIMATION

- But similar idea: Gardner et al., Learning to Predict Indoor Illumination from a Single Image
STYLE TRANSFER

- Make one image have the "style" or texture quality of another.
- Gatys et al., CVPR 2016:
  - Don't train a network for this
  - Instead take an existing network and look at the properties of its activations
  - Values of higher layers represent "content": Try to preserve them
  - Covariances of other layers represent style: Try to match them with other image
Set this up as an optimization problem, and minimize with SGD+Backprop from a random init.