BACK-PROPAGATION REVISITED

- Chain Rule: for scalar variables

\[ y = f(x) \quad L = g(y) \quad \frac{\partial L}{\partial x} = \frac{\partial y}{\partial x} \times \frac{\partial L}{\partial y} \]

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

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

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

\[
\begin{align*}
\text{self}.grad &= \nabla_y L \\
\text{self}.x.grad &= \nabla_x L
\end{align*}
\]

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

- 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., \( \partial y[i,j,k]/\partial x[l,m,n] \).

- 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).
Work out the expressions for:

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

- Consider the matmul layer
  \[ z = xy \]
- \( x \) is an \( L \times M \) matrix, \( y \) is an \( M \times N \) matrix, and so \( z \) is an \( L \times N \) matrix.

Work out the expressions for:

- \[ \frac{\partial L}{\partial z[l, m, n]} \]
- \[ \frac{\partial L}{\partial x[l, m, n]} \]

in terms of the incoming gradient (\( \partial L/\partial z[l, m] \)), and the inputs to the layer \( x[l, m] \) and \( y[m, n] \).

**BACK-PROPAGATION REVISITED**

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?
  \[
  (\nabla_J L)[b, y, x, c] = \frac{1}{K^2} \sum_{y'=0}^{\min(K-1, y)} \sum_{x'=0}^{\min(K-1, x)} (\nabla_I L)[b, y - y', x - x', c]
  \]

Realize for a given \( x[l, m] \), values of \( z[l, m] \) for the same \( l \) but \( \partial L/\partial n \) will depend on that \( x \) (have non-zero partials).

Both these equations represent matrix multiplications between \( V_I L \) and \( y \) or \( x \). That is how they are implemented in edf.py.
**BACK-PROPAGATION REVISITED**

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.

**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.
ILLUMINATION ESTIMATION

- 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.

- Indoor Illumination: More complicated, possibly multiple light sources.
  - 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.