Deep Learning from First Principles

Lesson 7 - Convolution backpropagation

Andrea Giardina
contact@andreagiardina.com
https://www.linkedin.com/in/agiardina

Why Backprop for Conv2d Matters

- In previous lectures, we implemented **Conv2d forward** using:
  - `im2col` to unfold patches of the input
  - **GEMM** (matrix multiplication) to compute convolutions efficiently
- Now: how do we compute gradients so that Conv2d can be trained?
- Key idea: if the forward is a matrix multiplication, the backward is the gradient of a matrix multiplication.
- This turns convolution backpropagation into a problem we already understand.

What Is the Upstream Gradient?

- During backpropagation, each layer receives a gradient from the layer above:
  $$\frac{\partial L}{\partial Y}$$
  where Y is the output of the current layer.
- This quantity is often called the **upstream gradient**:
  - it comes from the "next" layer in the computational graph,
  - it expresses how the loss changes with respect to the layer's output.
- The role of the current layer is to transform this upstream gradient into:
  1. gradients with respect to its parameters,
  2. a new gradient to send "downstream" to the previous layer.
- Every layer follows the same logic; only the local derivative changes.

Example: How the Upstream Gradient Is Formed

- Consider two consecutive layers:

$$X \xrightarrow{\text{Layer 1}} Y \xrightarrow{\text{Layer 2}} Z$$

  and a final loss $L = \mathcal{L}(Z)$.

- Layer 2 computes the first gradient:

$$\frac{\partial L}{\partial Y} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial Y}$$

- This quantity becomes the **upstream gradient** for Layer 1.
- Layer 1 then applies the chain rule:

$$\frac{\partial L}{\partial X} = \underbrace{\frac{\partial L}{\partial Y}}_{\text{upstream}} \cdot \frac{\partial Y}{\partial X}$$

- The process continues layer by layer until the first layer.

Recap: Forward Pass via `im2col` + GEMM

- For an input X of shape (C_{in}, H, W):

$$X_{\text{col}} = \texttt{im2col}(X)$$

- For a weight tensor W of shape $(C_{\text{out}}, C_{\text{in}}, k_H, k_W)$:

$$W_{\text{row}} = W.reshape(C_{\text{out}}, C_{\text{in}} k_H k_W)$$

- Forward output:

$$Y = W_{\text{row}} \cdot X_{\text{col}}$$

- This is just a matrix multiplication.
- The cache we store for backprop:

$$X_{\text{col}}, \; W_{\text{row}}, \; \text{original shape of } X$$

Backprop: What We Need

Given the upstream gradient:

$$\frac{\partial L}{\partial Y}$$

we want to compute:

1. $\frac{\partial L}{\partial W}$ gradient w.r.t. weights
2. $\frac{\partial L}{\partial X}$ gradient to pass to the previous layer

These correspond exactly to the derivatives of a linear layer.

Gradient w.r.t. the Weights: $\partial L / \partial W$

- Using GEMM notation:

$$Y = W_{\text{row}} \cdot X_{\text{col}}$$

- Then:

$$\frac{\partial L}{\partial W_{\text{row}}} = \frac{\partial L}{\partial Y} \cdot X_{\text{col}}^{\top}$$

- After computing this matrix gradient, reshape back to:

$$(C_{\text{out}}, C_{\text{in}}, k_H, k_W)$$

- Thus Conv2d weight backprop is again just matrix multiplication.

Gradient of the Row-Weight Matrix

Explanation:

- Each output element is

$$Y_j = \sum_{i=1}^{K} W_i \, X_{i,j}.$$

- Using the chain rule:

$$\frac{\partial L}{\partial W_i} = \sum_{j=1}^{N} \frac{\partial L}{\partial Y_j} \, X_{i,j}.$$

- Putting all components together yields the compact matrix form:

$$\frac{\partial L}{\partial W_{\text{row}}} = \frac{\partial L}{\partial Y} \, X_{\text{col}}^{\top}.$$

Gradient w.r.t. the Input: $\partial L / \partial X$

- Starting from:

$$Y = W_{\text{row}} X_{\text{col}}$$

- Input gradient in column space:

$$\frac{\partial L}{\partial X_{\text{col}}} = W_{\text{row}}^{\top} \frac{\partial L}{\partial Y}$$

- But X_{col} is not the original X!
- We must fold back the patches into the original spatial arrangement.
- This is done with:

$$\texttt{col2im}\left(\frac{\partial L}{\partial X_{\text{col}}}\right)$$

Role of `col2im`

- `col2im` performs the inverse of `im2col`.
- It redistributes patch gradients across the appropriate spatial locations.
- Overlapping patches accumulate gradient contributions.
- This is the key difficulty of Conv2d backward:
  - **unfolding is easy**
  - **folding back is tricky** due to overlapping regions

Backprop Summary

- Conv2d forward reduces to:

$$\text{im2col} + \text{GEMM}$$

- Consequently, Conv2d backward reduces to:

$$\text{GEMM backward} + \text{col2im}$$

- All gradients can be derived using standard matrix calculus.
- Next: implement these operations step by step in NumPy.

From `im2col` to `col2im`

- We already know `im2col`:
  - Takes an input tensor X and extracts all sliding patches
  - Rearranges them into a 2D matrix X_{col}
- `col2im` is the conceptual inverse of `im2col`:
  - Takes a 2D matrix of patches
  - Folds them back into a tensor with spatial dimensions
- **Key point:** patches overlap, so values must be *accumulated*, not just copied.

Shapes: What `col2im` Should Do

- Suppose `im2col` was called on an input of shape:

$$X \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W}$$

- With kernel size (k_H, k_W) and some (possibly implicit) stride/padding.
- Then `im2col` produced:

$$X_{\text{col}} \in \mathbb{R}^{(N \cdot C_{\text{in}} \cdot k_H \cdot k_W) \times L}$$

  where L is the number of sliding windows (e.g. $H_{\text{out}} \cdot W_{\text{out}}$).
- `col2im` must reconstruct something with the original spatial size:

$$\texttt{col2im}(X_{\text{col}}) \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W}$$

`col2im` as the Inverse of Unfolding

- Conceptually, `im2col` does:
  - Take each sliding window (patch) of X
  - Flatten it into a column
  - Stack all columns into a matrix
- `col2im` must reverse this:
  1. Take each column of X_{col}
  2. Reshape it into a patch of shape $(C_{\text{in}}, k_H, k_W)$
  3. Place this patch back into the correct spatial location
- When different patches cover the same pixel, their contributions are **summed**.

The Challenge: Overlapping Patches

- With stride $= 1$ and no padding:
  - Most pixels belong to multiple convolution windows
  - Therefore, they appear multiple times in `im2col`'s columns
- In `col2im`:
  - Each occurrence contributes to the same spatial location
  - The final value is the **sum** of all overlapping contributions
- This is crucial for backpropagation:
  - A single input pixel influences many outputs
  - Its gradient is the sum of all these influences

The Challenge: Overlapping Patches

- With stride $= 1$ and no padding:
  - Most pixels belong to multiple convolution windows
  - Therefore, they appear multiple times in `im2col`'s columns
- In `col2im`:
  - Each occurrence contributes to the same spatial location
  - The final value is the **sum** of all overlapping contributions
- This is crucial for backpropagation:
  - A single input pixel influences many outputs
  - Its gradient is the sum of all these influences

`col2im` in the Context of Backprop

- During Conv2d forward with `im2col`:

$$Y = W_{\text{row}} \cdot X_{\text{col}}$$

- During backprop we get:

$$\frac{\partial L}{\partial X_{\text{col}}} = W_{\text{row}}^{\top} \frac{\partial L}{\partial Y}$$

- But the previous layer expects a gradient with the *same shape as* X:

$$\frac{\partial L}{\partial X} \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W}$$

- `col2im` is exactly the operation that maps:

$$\frac{\partial L}{\partial X_{\text{col}}} \longrightarrow \frac{\partial L}{\partial X}$$

respecting shape and overlaps.

Intuition: `col2im` as Gradient Aggregator

- Think of `im2col` as:

$$X \mapsto X_{\text{col}}$$

  which **duplicates** some entries of X (because of overlaps).
- The Jacobian of this mapping has multiple 1's in positions corresponding to each duplicated element.
- In backprop, applying the transpose Jacobian corresponds to:
  - Taking all contributions from X_{col}
  - Summing them into the corresponding positions of X
- `col2im` is the concrete implementation of this aggregation.

What col2im Must Take into Account

- To reconstruct the tensor correctly, `col2im` needs:
  - The original input shape: N, C_{in}, H, W
  - The kernel size: (k_H, k_W)
  - The stride and padding used in `im2col`
- These parameters determine:
  - Where each patch should be placed
  - How many times each location is covered
- In backprop, using *inconsistent* parameters between `im2col` and `col2im` will produce wrong shapes or wrong gradients.

Lab time: Implement col2im

We have already implemented the im2col function:

```
def im2col(x, kH, kW, stride=1):
    # x: input numpy array di shape (C, H, W)
    C, H, W = x.shape
    H_out = (H - kH) // stride + 1
    W_out = (W - kW) // stride + 1

    X_col = np.empty((C * kH * kW, H_out * W_out), dtype=x.dtype)

    col = 0
    for i in range(0, H - kH + 1, stride):       # top-left row of patch
        for j in range(0, W - kW + 1, stride):  # top-left col of patch
            patch = x[:, i:i + kH, j:j + kW]    # patch with shape (C, kH, kW)
            X_col[:, col] = patch.reshape(-1)
            col += 1

    return X_col
```

- Implement the inverse operation col2im.
- It must reconstruct the original tensor from the column matrix.
- Overlapping regions must be summed.
- Use the same kernel size and stride used in im2col.

Hints for Implementing `col2im`

- Think of `im2col` as a function that *unfolds* the input into columns.
- `col2im` must perform the inverse operation: *fold* each column back into its spatial location.
- Each column corresponds to one sliding window of size (C, k_H, k_W).
- You will need to reshape each column back into this window shape.
- Determine the correct top-left coordinate of each window using the kernel size and stride.
- Accumulate values into the output tensor—some regions appear multiple times.
- At the end, the reconstructed tensor must have shape (C, H, W), matching the original input.

My trivial implementation

```python
def col2im(X_col, C, H, W, kH, kW, stride=1):
    x = np.zeros((C, H, W), dtype=X_col.dtype)
    col = 0
    for i in range(0, H - kH + 1, stride):
        for j in range(0, W - kW + 1, stride):
            patch = X_col[:, col].reshape(C, kH, kW)
            x[:, i:i + kH, j:j + kW] += patch
            col += 1
    return x
```

Extending `Conv2d` with Backpropagation

- Our current `Conv2d` class only defines:
  - a constructor storing hyperparameters and weights
  - a `forward(x)` method that:
    - applies padding
    - calls `im2col` for each image
    - uses GEMM to compute the output
- To train this layer, we also need a `backward` method.
- **Goal of backward**: given the upstream gradient

$$\frac{\partial L}{\partial Y}$$

compute:

1. gradients w.r.t. parameters (here: weights, later bias)
2. the gradient w.r.t. the input X.

What Should `Conv2d.backward` Do?

- Input to `backward`:

$$dY = \frac{\partial L}{\partial Y} \in \mathbb{R}^{N \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}}$$

- It must compute:
  1. $dW = \frac{\partial L}{\partial W}$ with shape
$$(C_{\text{out}}, C_{\text{in}}, k_H, k_W)$$
  2. $dX = \frac{\partial L}{\partial X}$ with shape
$$(N, C_{\text{in}}, H, W)$$

- `backward` should:
  - **store** dW inside the module (e.g. `self.grad_weight`)
  - **return** dX to the previous layer.

What Do We Need to Cache in `forward`?

- To implement the backward pass, the layer must remember:
  - the original input shape: (N, C_{in}, H, W)
  - the padding and stride used
  - the kernel size (k_H, k_W)
  - the **padded input** shape: $(H_{\text{pad}}, W_{\text{pad}})$
  - the **column representation** used in forward:

    $$X_{\text{col}} \in \mathbb{R}^{(C_{\text{in}} k_H k_W) \times (H_{\text{out}} W_{\text{out}})}$$

    for each image in the batch
  - the reshaped weights:

    $$W_{\text{col}} = W.\text{reshape}(C_{\text{out}}, C_{\text{in}} k_H k_W)$$

- This information forms the **cache** used by `backward`.

Backward: Gradients w.r.t. Weights

- For each image n in the batch we used:

$$Y_{\text{col}}^{(n)} = W_{\text{col}} X_{\text{col}}^{(n)}$$

- During backward, we receive $dY^{(n)}$ with shape

$$(C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$$

  and we reshape it into:

$$dY_{\text{col}}^{(n)} \in \mathbb{R}^{C_{\text{out}} \times (H_{\text{out}} W_{\text{out}})}$$

- Using matrix calculus:

$$dW_{\text{col}}^{(n)} = dY_{\text{col}}^{(n)} \cdot \left(X_{\text{col}}^{(n)}\right)^{\top}$$

- The final gradient w.r.t. the weights sums over the batch:

$$dW_{\text{col}} = \sum_{n=1}^{N} dW_{\text{col}}^{(n)}$$

  and then we reshape back to $(C_{\text{out}}, C_{\text{in}}, k_H, k_W)$.

Backward: Gradients w.r.t. the Input Columns

- For each image n:

$$Y_{\text{col}}^{(n)} = W_{\text{col}} X_{\text{col}}^{(n)}$$

- The gradient w.r.t. the column input is:

$$dX_{\text{col}}^{(n)} = W_{\text{col}}^{\top} \, dY_{\text{col}}^{(n)}$$

- Here:

$$dX_{\text{col}}^{(n)} \in \mathbb{R}^{(C_{\text{in}} k_H k_W) \times (H_{\text{out}} W_{\text{out}})}$$

- These are gradients in the same unfolded space as `im2col`.
- We still need to map them back to the spatial tensor $(C_{\text{in}}, H_{\text{pad}}, W_{\text{pad}})$.

Backward: Using `col2im` and Removing Padding

- For each image, we now have $dX_{\text{col}}^{(n)}$.
- We use `col2im` to fold these gradients back:

$$dX_{\text{pad}}^{(n)} = \texttt{col2im}\left(dX_{\text{col}}^{(n)}, C_{\text{in}}, H_{\text{pad}}, W_{\text{pad}}, k_H, k_W, \text{stride}\right)$$

- This reconstructs the gradient w.r.t. the *padded* input.
- Finally, Conv2d removes the padding:

$$dX^{(n)} = \text{crop}(dX_{\text{pad}}^{(n)}, \text{padding})$$

giving:

$$dX \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W}$$

which is returned by `backward`.

Summary: How `Conv2d` Changes with Backprop

- **Forward**:
  - pads the input
  - calls `im2col` for each image
  - applies GEMM with reshaped weights
  - stores all necessary tensors/shapes in a cache
- **Backward**:
  - receives dY (upstream gradient)
  - reshapes dY into column form dY_{col}
  - computes dW by GEMM and accumulates over the batch
  - computes dX_{col} using $W_{\text{col}}^{\top}$
  - applies `col2im` to get dX_{pad}
  - crops the padding to get dX and returns it
- Conceptually: Conv2d backward is just

$$\text{GEMM backward} + \texttt{col2im} + \text{crop padding.}$$

Lab time: Add Backpropagation to Conv2d

We already have a `Conv2d` class with:

- constructor: stores `in_channels`, `out_channels`, `kernel_size`, `stride`, `padding`
- `forward(x)`:
  - applies spatial padding to the input
  - calls `im2col` on each image in the batch
  - performs a matrix multiplication with the reshaped weights

Task: modify this class to support backpropagation:

- add a `backward(dY)` method
- make sure the layer:
  1. stores all information needed in the forward pass (cache)
  2. computes and stores the gradient w.r.t. the weights
  3. returns the gradient w.r.t. the input X

Lab time: Add Backpropagation to `Conv2d`

Implement `backward(self, dY)` assuming:

$$dY = \frac{\partial L}{\partial Y} \in \mathbb{R}^{N \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}}$$

In `forward`, store in a cache:

- the original input shape (N, C_{in}, H, W)
- the padded spatial size $(H_{\text{pad}}, W_{\text{pad}})$
- the kernel size (k_H, k_W), stride, padding
- the reshaped weights W_{col}
- for each image, the corresponding X_{col}

Lab time: Add Backpropagation to `Conv2d`

In `backward`:

- reshape each $dY[n]$ into a matrix

$$dY_{\text{col}}^{(n)} \in \mathbb{R}^{C_{\text{out}} \times (H_{\text{out}} W_{\text{out}})}$$

- compute the gradient w.r.t. weights in matrix form:

$$dW_{\text{col}}^{(n)} = dY_{\text{col}}^{(n)} \cdot \left(X_{\text{col}}^{(n)}\right)^{\top}$$

  accumulate over the batch and reshape to

$$(C_{\text{out}}, C_{\text{in}}, k_H, k_W)$$

- compute the gradient w.r.t. the column input:

$$dX_{\text{col}}^{(n)} = W_{\text{col}}^{\top} dY_{\text{col}}^{(n)}$$

- apply `col2im` to obtain $dX_{\text{pad}}^{(n)}$ with shape $(C_{\text{in}}, H_{\text{pad}}, W_{\text{pad}})$
- remove the padding to get $dX^{(n)}$ with shape (C_{in}, H, W)

Stack all $dX^{(n)}$ to form

$$dX \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W}$$

and return it.

My trivial implementation

```
class Conv2d:
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0)
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        kH, kW = self.kernel_size
        self.weight = np.random.randn(out_channels, in_channels, kH, kW) * 0.01
        self.grad_weight = np.zeros_like(self.weight) #Added
```

My trivial implementation

```python
def forward(self, x):
    """
    x: (N, C_in, H, W)
    return: (N, C_out, H_out, W_out)
    """
    N, C_in, H, W = x.shape
    kH, kW = self.kernel_size
    p = self.padding
    s = self.stride

    # To store original image shape
    self.x_shape = x.shape

    #We keep original input for the backward
    if p > 0:
        x_padded = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')
    else:
        x_padded = x

    _, _, H_pad, W_pad = x_padded.shape
    H_out = (H_pad - kH) // s + 1
    W_out = (W_pad - kW) // s + 1
```

My trivial implementation

```
#we save padded dimensions
self.H_pad = H_pad
self.W_pad = W_pad
self.H_out = H_out
self.W_out = W_out

W_col = self.weight.reshape(self.out_channels, -1)

#We store the weights in W_col format
self.W_col = W_col

out = np.empty((N, self.out_channels, H_out, W_out), dtype=x.dtype)

#Buffer to save all X_col
self.X_cols = []

for n in range(N):
    X_col = im2col(x_padded[n], kH, kW, stride=s)
    #X_col added to the buffer
    self.X_cols.append(X_col)
    Y_col = W_col @ X_col
    out[n] = Y_col.reshape(self.out_channels, H_out, W_out)

return out
```

My trivial implementation

```python
def backward(self, dY):
    N, C_out, H_out, W_out = dY.shape
    kH, kW = self.kernel_size
    C_in = self.in_channels
    s = self.stride
    p = self.padding

    H_pad = self.H_pad
    W_pad = self.W_pad
    W_col = self.W_col

    dW_col = np.zeros_like(W_col)
    dX = np.empty(self.x_shape, dtype=dY.dtype)
```

My trivial implementation

```
for n in range(N):
    dY_col = dY[n].reshape(C_out, H_out * W_out)
    X_col = self.X_cols[n]

    dW_col += dY_col @ X_col.T
    dX_col = W_col.T @ dY_col

    dX_padded = col2im(dX_col, C_in, H_pad, W_pad, kH, kW, stride=s)

    if p > 0:
        dX[n] = dX_padded[:, p:-p, p:-p]
    else:
        dX[n] = dX_padded

self.grad_weight += dW_col.reshape(self.weight.shape)
return dX
```

Lab time: Train Your `Conv2d` to Learn a Filter

Now that your `Conv2d` layer supports backpropagation, test it by training it on a simple synthetic task.

Goal: make a single convolution layer learn a fixed 3×3 filter.

Task specification

- Generate a batch of small random images (e.g. $N = 16$, $1 \times 8 \times 8$).
- Choose a known 3×3 filter, such as:
  - a blur (Gaussian-like) kernel
  - or a sharpening kernel
  - or a Sobel edge detector
- Produce the target outputs by applying this fixed filter manually (without using your `Conv2d`).
- Create a network consisting of a **single** `Conv2d` layer (1 input channel, 1 output channel, kernel size 3, padding 1).
- Train it with MSE loss so that:

$$\text{Conv2d}(x) \approx \text{FilteredTarget}(x)$$

Expected outcome: if the backward pass is correct, the loss will decrease and the learned kernel will become close to the target filter.

My trivial implementation

```python
def train_conv_to_learn_blur():
    np.random.seed(0)

    N = 16
    C = 1
    H = W = 8

    x = np.random.randn(N, C, H, W).astype(np.float64)

    target = conv2d_reference(x, BLUR_KERNEL)

    conv = Conv2d(
        in_channels=1,
        out_channels=1,
        kernel_size=3,
        stride=1,
        padding=1
    )
    conv.weight = np.random.randn(1, 1, 3, 3).astype(np.float64) * 0.1

    lr = 0.5
```

My trivial implementation

```
    for epoch in range(50):
        y = conv.forward(x)
        loss = mse_loss(y, target)
        dY = mse_grad(y, target)

        dX = conv.backward(dY)

        conv.weight -= lr * conv.grad_weight
        print(f"epoch {epoch:02d}  loss = {loss:.6f}")

    print("Final loss:", loss)
    print("Learned kernel:")
    print(conv.weight[0, 0])

    print("Reference blur kernel:")
    print(BLUR_KERNEL)

train_conv_to_learn_blur()
```

Thanks!