

Padding for Backpropagation

Lewis Levin and ChatGPT

March 22, 2005

Resolution of Padding for Backpropagation

Prompting sequence for Deep Search

Lewis:

during backpropagation from one convolution layer to another convolution layer below, there are difficulties with the size of arrays being incompatible.

ChatGPT:

When performing backpropagation through convolutional layers in a neural network, size mismatches can arise due to various reasons related to the way convolutions and gradients are computed. Here are some key points to check:

1. **Stride and Padding Mismatch** • If your forward pass used padding, ensure that the backward pass accounts for it correctly. If padding was added, the gradient also needs to be distributed correctly over the padded input. • If stride > 1, the gradient map will be smaller than the original feature map, so it must be properly resized when propagating backward.
2. **Gradient Flow in Transposed Convolutions** • When backpropagating through a convolutional layer, the gradient with respect to the input (L/X) is computed using a transposed convolution (sometimes called deconvolution). • Ensure that the kernel size, padding, and stride settings align properly when computing gradients.
3. **Shape Mismatch Between Input and Gradient** • If your convolution operation reduces spatial dimensions (H, W), the gradient must be upsampled before propagating further. • If using a framework like PyTorch or TensorFlow, check how they compute the shape of gradients by running `.shape` on tensors.
4. **Check Kernel Size and Dilation** • If your convolution uses dilation, the receptive field of each neuron increases. The backward pass must correctly distribute gradients over the expanded receptive field.
5. **Batch Size Consistency** • Ensure that the batch size remains the same across forward and backward passes. This can be an issue when manually slicing tensors.
6. **Framework-Specific Issues** • If using a deep learning framework (e.g., PyTorch, TensorFlow, JAX), check whether: • You're using `torch.nn.Conv2d` vs. `torch.nn.ConvTranspose2d` correctly. • TensorFlow's `conv2d_backprop_input` function is used correctly with expected `input_shape`.

Would you like me to help debug a specific case? If so, sharing code or tensor shapes would be helpful!

Lewis:

The biggest difficulty:

Using no padding

The current layer loss with respect to the input is a convolution with a double reversed kernel. This will be smaller in the `img_x` and `img_y` dimensions than the size of the convolution of the layer below. But it must match that lower layer convolution. This can be done by introducing zero padding to match the lower layers image plane dimensions even though padding wasn't originally used. This won't create a problem with the calculated loss.

Question 1: can this arbitrary zero padding be added at the bottom and right sides of the loss array? Or has one row at top and bottom and 1 column at right and left?

Once calculated this initial pass of the loss calculation must be element-wise multiplied times the gradient of the activation function of the original linear combination output from feedforward (sometimes called z when referring to fully connected layers). This array and the output of the gradient function (for the layer's activation) is also smaller than the convolution of the layer below, which it must match.

Question 2: Can the “ z ” output from the feedforward pass be zero padded before calculating the gradient of activation? If so, can the padding be 2 extra rows at the bottom and 2 columns at the right? Or should the padding be 1 row each at top and bottom and one column each at the right and left sides?

These questions are never adequately addressed by greek letter notation of the process and not in any pseudo code simplifications.

Chatgpt:

You're dealing with zero-padding adjustments during backpropagation in convolutional layers where no padding was originally used. To ensure a precise response: 1. Are you working within a specific deep learning framework (e.g., PyTorch, TensorFlow, NumPy custom implementation)? 2. What is the kernel size and stride for your convolutional layers? This affects how the output dimensions shrink. 3. Do you need a general theoretical answer, or are you looking for practical implementation details (e.g., Python code examples)?

Understanding these details will help ensure an accurate response to your padding questions.

Lewis:

I am writing the actual code in Julia.

the 3rd layer (assuming the input is layer 1) is convolution with a result of $24 \times 24 \times 8 \times N$. the 2nd layer is convolution with result of $26 \times 26 \times 4 \times N$ the input is an image tensor of dimensions $28 \times 28 \times 1 \times N$

I need practical implementation details because this is the problem never addressed by hand wave theoretical discussions.

ChatGPT:

Thanks for providing the details. I will research the practical implementation of handling zero-padding in backpropagation for convolutional layers in Julia, ensuring that your calculations properly align with the lower layer's convolution dimensions. I'll include specific guidance on where padding should be added when resizing the loss tensor and “ z ” from the feedforward pass.

I'll let you know when I have the results.

Resulting Report

Thanks for providing the details. I will research the practical implementation of handling zero-padding in backpropagation for convolutional layers in Julia, ensuring that your calculations properly align with the lower layer's convolution dimensions. I'll include specific guidance on where padding should be added when resizing the loss tensor and “ z ” from the feedforward pass.

I'll let you know when I have the results.

Handling Zero-Padding in Backpropagation (Julia Implementation)

When implementing convolution backpropagation from scratch (e.g. in Julia), careful handling of **zero-padding** is crucial to align gradient dimensions. In your scenario (no padding in forward pass), each convolution reduced the spatial size ($28 \rightarrow 26 \rightarrow 24$). We'll address each question in turn with practical details:

1. Padding dL/dX (Input Gradient) in Backpropagation

Situation: The 3rd layer’s output gradient dL/dY has size $24 \times 24 \times 8$ (for N images), but the 2nd layer’s output (which was the 3rd layer’s input) is $26 \times 26 \times 4$. To compute dL/dX for the 2nd layer, we convolve dL/dY with the **flipped (180° rotated) kernel**. Without padding, this convolution yields a 22×22 result (shrinking again), which is too small. We need to expand (pad) the gradient so that the resulting dL/dX matches the $26 \times 26 \times 4$ volume.

How to pad: Use **symmetric zero-padding on all sides** of the dL/dY gradient. In a valid forward conv (no padding), each output pixel skipped one input row/col at the top and bottom (for a 3×3 filter, one on each side). Thus, in backprop we add one row of zeros at the top & bottom and one column of zeros at the left & right of the gradient $()$ (Convolutional Neural Networks · Knet.jl). This “full” convolution approach ensures the input gradient has the correct size (convolutional neural network - Padding and stride in backpropagation of a conv net - Cross Validated). Padding only on the bottom and right (asymmetric) would mis-align the gradient – it would effectively shift the convolution window, causing incorrect correspondence between output gradients and input pixels. Each input pixel’s gradient should accumulate evenly from all the outputs that originally touched it, which requires padding **both** ends of each dimension (Backpropagation in a Convolutional Neural Network).

Julia implementation: You can manually create a larger zero array and copy the smaller gradient into the center. For example, to pad a $24 \times 24 \times 8$ gradient to $26 \times 26 \times 8$ (one pixel border of zeros):

```
# dL_dY is the upstream gradient from layer 3 with shape (24, 24, 8, N)
dL_dY = rand{Float32, 24, 24, 8, N}

# Prepare a zero-padded array for dL_dY with a 1-pixel border on each side
dL_dY_padded = zeros{Float32, 26, 26, 8, N}
dL_dY_padded[2:end-1, 2:end-1, :, :] = dL_dY # insert the 24x24 data into the center

# Now dL_dY_padded is 26x26x8. Convoluting this with the flipped 3x3 kernel
# will produce a 26x26x4 output (dL/dX for layer 2) as desired.
```

Alternatively, if you have the kernel W from layer 3 (of shape $3 \times 3 \times 4 \times 8$ for $\text{in}=4, \text{out}=8$), you could use Julia’s conv utilities to do this in one step by specifying a padding. For example, using **NNlib.jl** (which Flux uses under the hood):

```
using NNlib: conv

rotW = reverse(W, dims=(1,2)) # flip kernel spatially (3x3)
rotW = permutedims(rotW, (1,2,4,3)) # swap input/output channels => shape 3x3x8x4
dL_dX = conv(dL_dY, rotW; stride=(1,1), pad=(1,1)) # full convolution, pad=(1,1) adds 1px border
```

This yields dL_dX with shape $26 \times 26 \times 4 \times N$ directly. Under the hood, this zero-pads dL_dY on all sides by 1 (Convolutional Neural Networks · Knet.jl). The key is that **pad=(1,1)** pads both top-bottom and left-right. (Padding bottom/right only would use **pad=(0,1)** etc., which is **not** what we want for backprop.)

Why symmetric padding? In forward valid convolution, the first output pixel corresponds to the top-left 3×3 region of the input, and the last output pixel to the bottom-right 3×3 region. Thus the input’s top row and leftmost column each influence only one output, as do the bottom row and rightmost column. During backward pass, those border input pixels should each get gradients only from one output position. By padding all sides equally, the convolution in backprop “slides” over the gradient in the same way the forward filter slid over the input (Backpropagation in a Convolutional Neural Network). For a 3×3 filter with no forward padding, one-zero border on each side is needed so that every input position (including corners) receives its correct gradient. If you padded only on the bottom and right, the top and left input pixels would incorrectly miss out on gradient (and the bottom-right would get too much). In summary: **always mirror the forward padding pattern in backprop** (no padding forward → pad equally on all sides backward).

2. Padding the Pre-Activation Output (“z”) for Activation Gradient

Situation: Let “z” be the pre-activation output of layer 2 (the $26 \times 26 \times 4$ volume before applying nonlinearity). In forward propagation with valid conv, z has no outputs for the would-be border positions (since the conv didn’t produce a 28×28 output). However, in backprop we now have a 26×26 gradient (after padding, from step 1) for the activation output. To apply the activation’s derivative, we need z and the gradient to have the same shape.

Yes – you **can zero-pad** the stored “z” array to $26 \times 26 \times 4$ so that it aligns with the 26×26 gradient. The padding should be done **symmetrically** just like above (1 row on top & bottom, 1 column on left & right) if you’re padding a 24×24 array to 26×26 . This way each activation value still lines up with its corresponding gradient element. Adding 2 rows to the bottom and 2 columns to the right (with none on top/left) would misalign values; for example, the original top-left activation would suddenly be paired with a gradient that actually belongs to what was the second input row/col. By distributing padding evenly, you ensure the spatial correspondence remains correct.

After padding, the newly introduced border values of “z” are zeros – which makes sense because those positions had no output in the forward pass (they represent the activation input for a conv window that never existed). When you compute the activation function’s gradient, those zeros will yield a neutral effect. For instance, if using ReLU, $f'(z)=1$ for $z>0$ and 0 for $z<0$; at $z=0$ you can take 0 or 1 – either way, the gradient at those padded locations will end up zero because the upstream gradient there is zero (we padded it in step 1). In other words, padding z with 0s does not introduce any erroneous gradient; it just extends the array for alignment, and any activation derivative at padded positions will multiply a zero upstream gradient (Backpropagation in a Convolutional Neural Network).

Julia implementation: Similar to above, create a padded array for z and copy the values:

```
# z is the conv pre-activation (linear output) of shape 24×24×4×N (for example)
z = rand{Float32, 24, 24, 4, N}

# Pad z to 26×26×4×N symmetrically
z_padded = zeros{Float32, 26, 26, 4, N}
z_padded[2:end-1, 2:end-1, :, :] = z

# Now compute activation gradient. For ReLU as an example:
relu_grad = float.(z_padded .> 0) # indicator (1 where z>0, 0 where z<=0)

# Suppose dL_da is the gradient w.r.t. the activated output (after activation) from the next layer,
# which should be 26×26×4×N after the padding done in step 1.
dL_da = rand{Float32, 26, 26, 4, N} # (just an example gradient array)

# Elementwise multiply to get dL/dz (gradient w.r.t pre-activation)
dL_dz = dL_da .* relu_grad
```

For other activation functions, compute the derivative similarly (e.g., sigmoid: $f'(z) = (z) \cdot (1 - (z))$) on the padded z. The key is that `z_padded` and the incoming gradient have matching 26×26 shape, so the elementwise multiplication yields a $26 \times 26 \times 4 \times N$ `dL_dz`. This `dL_dz` can then be used in the convolution backprop for layer 2’s weights or passed further down.

Summary: In backpropagation through an unpadded convolution, add zero padding **evenly to all sides** of the gradient or pre-activation maps whenever you need to **increase the spatial dimensions** to match the previous layer. This preserves the alignment of values. In practice, this means padding by `kernel_size-1` total (split evenly top/bottom, left/right) for `stride=1`. For a 3×3 filter, pad 1 on each side (`()`). Doing so is equivalent to a “full” convolution in mathematics (convolutional neural network - Padding and stride in backpropagation of a conv net - Cross Validated), ensuring the gradient covers every input position. By contrast, asymmetric padding (only bottom/right) would shift the convolution window and yield incorrect gradients.

Both the **loss-gradient convolution** and the **activation-gradient** steps can use this approach. As a result, your backprop implementation will produce gradients with the correct 26×26 (and ultimately 28×28 for the input layer) shapes. In Julia, you can implement this by manually padding arrays as shown above or by using convolution functions with a padding parameter. The main impact on gradient computation is correctness – with proper padding, each input and filter weight gets its due contribution from the loss, matching the forward pass geometry (Backpropagation in a Convolutional Neural Network).

References:

1. Backpropagation through convolution requires a “full” convolution (zero-padding the upstream gradient) to recover the input’s size (convolutional neural network - Padding and stride in backpropagation of a conv net - Cross Validated).

2. Julia/NNlib's conv API uses a `pad` parameter to handle padding; `pad=(1,1)` pads both sides for a 3×3 kernel (preserving size) (Convolutional Neural Networks · Knet.jl). This mirrors the idea that to preserve size, pad by $(W-1)/2$ on each side (for W odd).
3. The input-gradient is essentially a convolution of the output-gradient with the flipped kernel **with zeros padded around the gradient** (Convolutional Neural Networks · Knet.jl) (Backpropagation in a Convolutional Neural Network). As one tutorial notes, "we need to zero-pad the output [gradient], since around the edges, the indices ... may be negative (i.e. those outputs don't exist)" (Backpropagation in a Convolutional Neural Network). Padding the gradient on *all* sides by the appropriate amount (equal to the forward padding or filter radius) is the correct strategy ().

Sources Citations deeplearning.cs.cmu.edu Backward Pass example: • To visualize the pattern more clearly, we pad the gradient tensor with zeros at the top and bottom as well as to the left and right. • The number of zeros padded on either side is equal to the stride denizyuret.github.io Convolutional Neural Networks · Knet.jl Here is the gradient for the input: Faviconstats.stackexchange.com convolutional neural network - Padding and stride in backpropagation of a conv net - Cross Validated As far as I know you would need to perform a "full" convolution during the backpropagation step. So the gradients from the $l+1$ layer will be a $(7, 7)$ tensor. The "full" convolution with the rotated filter $(2, 2)$ will result in a $(8, 8)$ tensor. Removing the original padding as added to your input $(4, 4)$ would again result in a $(4, 4)$ tensor. There would be no need to calculate or guess any further padding/stride while doing the backprop convolution. Faviconmukulrathi.com Backpropagation in a Convolutional Neural Network $X(m \times i) * i, j, c \text{ } J = * k \text{ } a \text{ } b \text{ } Z(m \text{ } i) * i - a, j - b, k \text{ } J * W \text{ } _ \text{ } a, b, c$ denizyuret.github.io Convolutional Neural Networks · Knet.jl operator. If `padding=k`, x will be assumed padded with k zeros on the left and right before the convolution, e.g. `padding=1` means treat x as $[0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 0]$. The default padding is 0. For inputs in D -dimensions we can specify padding with a D -tuple, e.g. `padding=(1,2)` for 2D, or a single number, e.g. `padding=1` which is shorthand for `padding=(1,1)`. The result will have $Y = X + 2P - W + 1$ elements where P is the padding size. Therefore to preserve the size of x when $W=3$ we should use `padding=1`.

All Sources: <https://stats.stackexchange.com/> <https://discourse.julialang.org/> <https://denizyuret.github.io/> <http://johnwlambert.github.io/> <https://juliaimages.github.io/> <http://people.csail.mit.edu/> <https://github.com/> <https://deeplearning.cs.cmu.edu/> <https://mukulrathi.com/>

Chatgpt accessing sources and reading:

Zero-Padding in Backpropagation for Convolutional Layers in Julia

Zero-padding output gradients during backpropagation efficiently aligns shapes with next-layer dimensions, addressing gradient shape mismatches and aligning computations accurately. Padding gradients during backpropagation ensures they match convolution dimensions. For instance, zero-padding gradients to $26 \times 26 \times 4$ after a $24 \times 24 \times 8$ output keeps the convolution process accurate. Padding gradients symmetrically ensures they match convolution dimensions, avoiding shape mismatches and maintaining the intended output size. This approach fosters a smooth and precise backward propagation process. Symmetrically padding pre-activation outputs ensures proper alignment with convolution dimensions, maintaining accurate backward propagation and gradient computation. This method effectively compensates for any dimensional discrepancies. Searched for Julia convolution backpropagation with zero padding ChatGPT I'm checking the CrossValidated result on padding and stride in backpropagation of a conv net. This looks relevant for understanding the concepts.

Read stats.stackexchange.com

ChatGPT I'm scrolling through lines 173 to 185 in the CV document, as the answer might be hidden in a collapsed block.

Read more from stats.stackexchange.com

Searched for Julia convolution backprop with specific dimensions Searched for Julia convolution backprop with zero padding ChatGPT I'm considering different approaches to backpropagation and padding strategies to meet specific criteria. Searched for zero-padding in loss array ChatGPT I'm scanning Julia discourse to see if it's useful for convolution and `juliadiff`. This makes me think it might help with understanding Julia specific operations.

Read discourse.julialang.org

ChatGPT It's interesting to see that the emphasis is on a numeric scheme for convolution, diverging from the traditional approach in Convolutional Neural Networks (CNNs). Searched for double reversed kernel convolution backprop in Julia ChatGPT I'm digging into convolution with the same output size, particularly focusing on reversing kernels and border differences.

Read more from discourse.julialang.org

ChatGPT I'm thinking through symmetrical padding, especially for CNN backprop with valid convolution. This leads me to unusual padding options, focusing on exact Julia implementation. Analyzing the necessity and symmetry of padding "z" involves considering forward convolution losses. Symmetrical approaches align gradients with convolution dimensions. Padding choices are shaped by specific requirements. OK, let me see: symmetrical padding is preferred for aligning gradients with convolution dimensions. Julia's `padarray` or `cat` functions can efficiently implement this approach. Searched for symmetrical zero padding in Julia ChatGPT I'm pulling together details on NNlib convolution with `pad` and flipped kernels, emphasizing practical Julia implementation and symmetrical padding for accurate gradient alignment.

Read denizyuret.github.io

ChatGPT OK, let me see: padding in backpropagation helps keep output sizes consistent by adding zeros. This method is the same as in convolution computations.

Read more from denizyuret.github.io

ChatGPT Taking a closer look at a specific convolution with zero padding, which might imply symmetry in this context.

Read johnwlambert.github.io

ChatGPT Interestingly enough, symmetrical padding is used in the Knet documentation, where padding is set to $P=W-1$ to maintain input size. This method applies to both forward and backward convolution. Searched for zero padding in backward convolution ChatGPT Ensuring symmetrical zero-padding for gradients aligns dimensions during backward convolution, with suggestions on using Julia's `NNlib.conv` or `padarray`.

Read discourse.julialang.org

ChatGPT Noticing that `Pad(1,1)` likely refers to symmetric padding, with repeated edge values instead of zeros, and considering this default behavior.

Read juliaimages.github.io

Read discourse.julialang.org

Read people.csail.mit.edu

ChatGPT I'm thinking through how padding affects convolution output. Symmetrical padding seems best for consistency, but reliability still needs more tweaks.

Read discourse.julialang.org

Read github.com

ChatGPT The user is confirming that the error message is not related to the current question. Analyzed with python ChatGPT Working through convolution techniques to accurately align the output gradient with the reversed filter size, ensuring the output size matches the input size. Analyzed with python ChatGPT I'm rethinking the 'valid' correlation on padded gradients, aiming for a complete xgrad of length 5, not just a partial sequence. Analyzed with python ChatGPT OK, let me see. Padding gradients symmetrically in Julia ensures each input element gets contributions from appropriate output gradients, aligning with forward convolution, avoiding partial gradients. I'm detailing padding methods in Julia, including symmetry and manual padding, as well as computing activation gradients like ReLU with element-wise operations. Searched for backprop convolution output size D2L Read denizyuret.github.io

ChatGPT I'm gathering details from Goodfellow 2010, focusing on Knet and crossval; not heading in that direction at the moment. Read deeplearning.cs.cmu.edu

ChatGPT Padding the gradient tensor symmetrically by stride equals stride, such as padding by 1 when stride=1, appears accurate. This suggests a method suitable for earlier questions. Searched for zero padding pre-activation in backprop convolution Read mukulrathi.com

ChatGPT OK, let me see. For backpropagation in convolutional layers using zero padding to fix spatial dimensions, symmetric padding across all edges ensures the gradient map aligns perfectly with the original input features. Wrapping up with symmetric zero padding and some handy code. This ensures the convolution and activation gradients align, maintaining precision in backpropagation. Research completed in 7m · 16 sources