Convolutional Networks: Advanced Topics

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

*Optional. Assumes the **CNN** note (Week 11); §1 also uses the matrix form of backprop from **NN_ADVANCED**. Nothing here is needed for the required problems of HW6. Section 5 supports optional Problem 7, and Section 6 goes deeper into Problem 6.*

1 Backprop Through a Convolution and a Pooling Layer

PyTorch computes these gradients for you, but they are worth seeing once: they show that a convolution's backward pass is again a convolution.

1-D, by hand. Input $x = (x_0, \dots, x_3)$, kernel $k = (k_0, k_1, k_2)$, output $y_i = \sum_a k_a x_{i+a}$ for $i = 0, 1$. Let $g_i = \partial L / \partial y_i$ arrive from above.

- **Kernel gradient.** k_a is used at every output position, so its gradient is a sum over positions: $\partial L / \partial k_a = \sum_i g_i \, x_{i+a}$ — the *correlation of the input with the output gradient*. Weight sharing in the forward pass becomes summing in the backward pass.
- **Input gradient.** x_j feeds every output whose window covers it: $\partial L / \partial x_j = \sum_i g_i \, k_{j-i}$ (terms with $j - i$ outside 0..2 are absent) — a *full* correlation (zero-padded) of g with the *flipped* kernel.

Worked example: numbers. $x = (1, 2, 3, 4)$, $k = (1, -1, 2)$: $y = (1 - 2 + 6,\ 2 - 3 + 8) = (5, 7)$. With $g = (1, 2)$: $\partial L / \partial k = (1 \cdot 1 + 2 \cdot 2,\ 1 \cdot 2 + 2 \cdot 3,\ 1 \cdot 3 + 2 \cdot 4) = (5, 8, 11)$, and $\partial L / \partial x = (1, 1, 0, 4)$, which is g padded with two zeros on each side, correlated with the flipped kernel $(2, -1, 1)$. Both agree with PyTorch's autograd.

The unfold view. With the patch matrix of the CNN note, $Y = WP$ is an ordinary matrix product, so the backward pass is the one from NN_ADVANCED: $\partial L / \partial W = G \, P^\top$ and $\partial L / \partial P = W^\top G$. The second has one entry per *copy* of a pixel; since each pixel was copied into several windows, its gradient is the sum over its copies ("fold", the inverse of unfold). On the 3×3 example with $K = \left[\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}\right]$ and G all ones: $\partial L / \partial W = (8, 12, 20, 24)$ (the sum of the four patches), and the centre pixel, covered by all four windows at four different offsets, gets $0 + 1 + 2 + 3 = 6$; the full input gradient is $\left[\begin{smallmatrix} 0 & 1 & 1 \\ 2 & 6 & 4 \\ 2 & 5 & 3 \end{smallmatrix}\right]$.

Pooling. Average pooling sends g/(window size) to every pixel of the window. Max pooling sends the whole g to the pixel that was the maximum and 0 to the others (on $\left[\begin{smallmatrix} 1 & 5 \\ 3 & 2 \end{smallmatrix}\right]$ the gradient is $\left[\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right]$): only the winner influenced the output. Max pooling has no parameters, so nothing is learned there; it only *routes* the gradient.

2 Receptive Fields: Why Small Kernels, Stacked

The **receptive field** of an output is the patch of the input image it depends on. One 3×3 layer sees 3×3; two stacked 3×3 layers see 5×5 (each output of layer 2 looks at 3×3 outputs of layer 1, each of which looks at 3×3 pixels); L layers see $(2L+1) \times (2L+1)$. Pooling and stride make the field grow much faster.

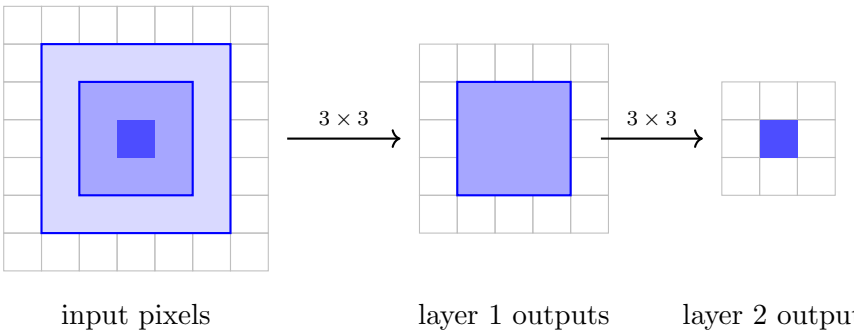


Figure 1: One output of layer 2 (dark) depends on a 3×3 block of layer 1, which depends on a 5×5 block of the input: two 3×3 layers have a 5×5 receptive field.

So why not one big kernel? Compare weights for c channels in and out: two 3×3 layers cost $2 \cdot 9c^2 = 18c^2$ for a 5×5 field, one 5×5 layer $25c^2$; three 3×3 layers cost $27c^2$ for a 7×7 field, one 7×7 layer $49c^2$. The stack is cheaper *and* has a nonlinearity between its layers, so it can represent more. This is VGG's design rule (Simonyan and Zisserman, 2015): blocks of 3×3 convolutions, then halve the image and double the channels. Every modern CNN, including HW6's small ResNet, follows it.

3 The Modern-CNN Family Tree

network	**year**	**what it introduced**
LeNet	1998	convolution + pooling + dense layers, trained by backprop (the CNN note)
AlexNet	2012	the same idea, much bigger, on GPUs and ImageNet; ReLU, dropout, augmentation. The start of the deep-learning era
NiN	2013	1×1 convolutions; global average pooling instead of large dense layers
VGG	2014	uniform blocks of 3×3 convolutions (§2)
GoogLeNet	2014	"Inception" blocks: parallel branches with 1×1, 3×3, 5×5 kernels and pooling, concatenated
Batch norm	2015	a layer that made deep networks train fast (the CNN note)
ResNet	2015	residual blocks; networks of 100+ layers (§5)
DenseNet	2017	each layer receives the *concatenation* of all earlier layers' outputs, instead of their sum
ConvNeXt	2022	a ResNet modernized with design choices borrowed from Transformers (larger kernels, layer norm, fewer activations)

Read the table as one story: make the network deeper, and find the trick that keeps it trainable at each step (ReLU, batch norm, residuals). d2l chapter 8 builds each of these in code.

4 Normalization Layers Compared

All four layers below compute the same thing — subtract a mean, divide by a standard deviation, then apply a learned γ, β per channel — and differ only in *which values share one mean and variance*. For a tensor (n, c, h, w):

layer	**statistics over**	**typical use**
batch norm	n, h, w (one channel, all examples)	CNNs with reasonable batch sizes
layer norm	c, h, w (one example, all channels)	Transformers, RNNs
instance norm	h, w (one example, one channel)	style transfer, image generation
group norm	h, w and a group of channels (one example)	CNNs trained with small batches

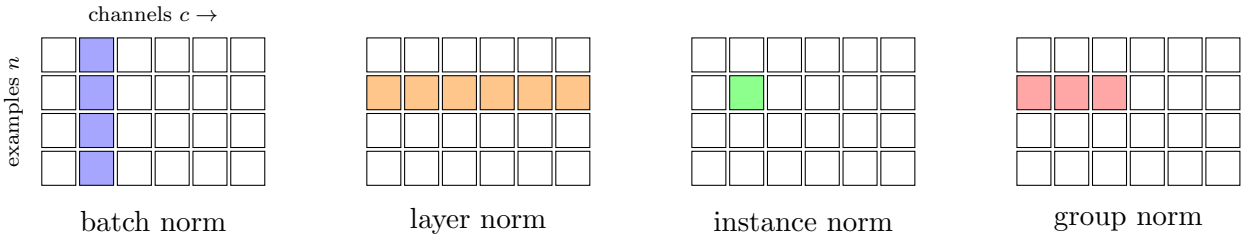


Figure 2: Each square is one channel's $h \times w$ map for one example; the shaded squares share one mean and variance.

Batch norm depends on the batch. With batch size 2, the batch mean of a channel is a noisy estimate, and every prediction in training depends on the other example in the batch. Layer, instance and group norm use one example at a time, so they behave the same in training and at test time and need no running averages. That is one reason Transformers use layer norm (the TRANSFORMER note, Week 14): their batches are padded sentences of different lengths, and the statistics of one token should not depend on other sentences.

Why does batch norm help? Told honestly. The original paper (Ioffe and Szegedy, 2015) argued that BN reduces *internal covariate shift*: the change in the distribution of each layer's inputs as earlier layers are updated. Later experiments (Santurkar et al., 2018) found that networks still train well when that shift is deliberately reintroduced after BN, and proposed instead that BN makes the loss surface *smoother*, so larger steps are safe. Both agree on what you observe in HW6 — faster training and tolerance of larger learning rates — and the complete explanation is still open.

5 Plain vs Residual Networks at Depth (Optional Problem 7)

The degradation problem. A deeper network can represent everything a shallower one can: copy the shallow network and make the extra layers compute the identity. So a deeper network's *training* error should never be higher. In practice, for plain (non-residual) networks, it is: He et al. (2016) report that on CIFAR-10 a 56-layer plain network has higher *training* error than a 20-layer one. This is not overfitting — overfitting would mean low training error and high test error — but an optimization failure: gradient descent cannot find the good solution that exists.

Why residuals fix it. Unroll a stack of residual blocks, $x_{l+1} = x_l + R_l(x_l)$ (ignoring the ReLU after the addition):

$$x_L = x_l + \sum_{k=l}^{L-1} R_k(x_k), \qquad \frac{\partial L}{\partial x_l} = \frac{\partial L}{\partial x_L}\Big(I + \frac{\partial}{\partial x_l}\sum_{k=l}^{L-1} R_k(x_k)\Big).$$

The gradient reaching block l always contains the term $\partial L/\partial x_L$ itself, delivered without passing through any layer: the "go around" of the instructor's handwritten page, in symbols. A plain stack has only the product of all the layer Jacobians, which can vanish. And "learn the identity" becomes "learn $R \approx 0$", which weight decay and small weights push toward anyway.

The pre-activation ResNet. The ReLU after the addition (HW6's layout) still sits on the main path. He et al.'s follow-up (2016) reorders each block to BN → ReLU → conv → BN → ReLU → conv, with the addition last, so the skip path from input to output is a pure sum. The same move appears in HW7 as *pre-norm* Transformers: normalize the input of each sub-layer, keep the residual path untouched.

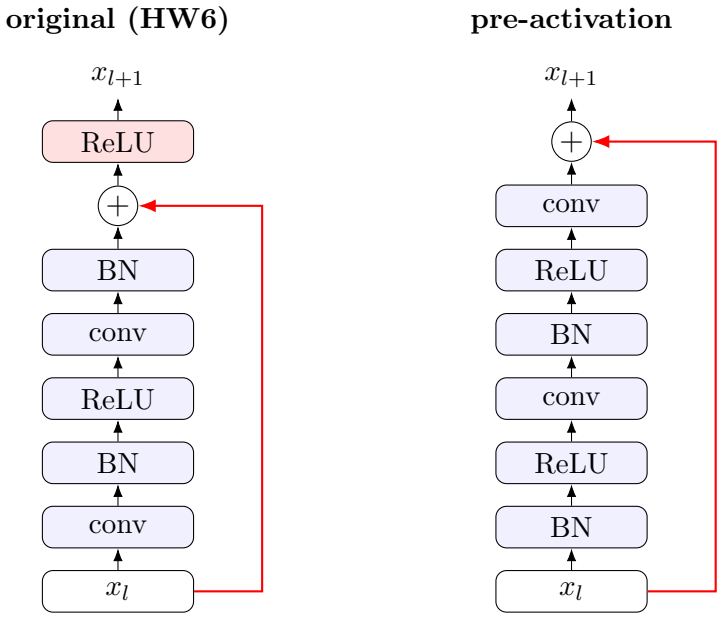


Figure 3: Left: HW6's block, with a ReLU after the addition (on the main path). Right: the pre-activation block, whose skip path from x_l to x_{l+1} is a pure sum.

What Problem 7 should show. Build a plain network and a residual network with the same widths and the same number of convolutions (20 or more), train both with the same budget, and plot the *training* loss. Expect the plain network to train worse — the degradation above — and the residual one to train at least as well as a shallower one (to judge that last claim you need the shallower network too: train one from the same family, e.g. the plain or residual net with half the blocks, under the same budget). Two pitfalls: compare training curves, not only test accuracy; and keep batch norm in both, so the comparison isolates the shortcuts.

6 Equivariance and Invariance, Stated Properly

Let T shift an image by one pixel. A layer f is **shift-equivariant** if $f(Tx) = Tf(x)$ and **shift-invariant** if $f(Tx) = f(x)$.

- **Convolution with stride 1** is equivariant, *except at the borders*: zero padding means pixels near the edge see zeros that move with the frame, not with the image.
- **Stride breaks equivariance.** Stride 2 keeps every second position. An edge response at position 2, $(0, 0, 1, 0, 0, 0, 0, 0)$, subsamples to $(0, 1, 0, 0)$; moved to position 3 it subsamples to $(0, 0, 0, 0)$ — the edge disappears. Pooling with stride 2 has the same problem in milder form (the CNN note's window-boundary example). Zhang (2019) shows that blurring before subsampling restores much of the lost robustness.
- **Global average pooling** over a feature map is invariant to shifts that keep the pattern inside the map (the edge at position 2 or 5 both average to 0.125). That is why HW6's ResNet, which ends with global average pooling, is less tied to positions than LeNet, which ends with dense layers over a $16 \times 5 \times 5$ map.
- **Rotation, scale, mirror images**: convolution is not equivariant to any of them. A network learns them only from examples (augmentation) or from special architectures that build them in (group-equivariant CNNs, Cohen and Welling, 2016). And only where they keep the label: an upside-down 6 is a 9.

7 Transfer Learning in More Detail

What the layers learn. Visualizations of trained CNNs (Zeiler and Fergus, 2014) show the first layer's kernels as edge and colour-blob detectors, the next layers as textures and simple shapes, and the last ones as object parts (faces, wheels). Early layers are general; late layers are specific to the training task. Hence the usual practice: replace the last layer, keep the rest.

How much to train. Between the linear probe (train only the head) and full fine-tuning (train everything) are intermediate choices: unfreeze only the last block; or train all layers with smaller learning rates for earlier layers (*discriminative learning rates*). With very little data, freezing more is safer; with more data, fine-tuning more wins.

Batch norm while fine-tuning. A pretrained network's BN layers carry ImageNet's running statistics. In a linear probe the network stays in evaluation mode, so those statistics are used as they are. When fine-tuning, the BN layers are in training mode and re-estimate the statistics from the new data's batches; with tiny batches it can be better to keep them frozen.

Resolution. ResNet-18 was trained on 224×224 images. In the HW6 build's pilot on the 5,000-image subset, the probe reached 0.548 / 0.729 / 0.746 test accuracy at 64 / 128 / 224 pixels and fine-tuning 0.819 / 0.892 / 0.893, with fine-tuning taking 14 / 40 / 118 seconds. HW6 uses 128 pixels: the pretrained features need objects at roughly the size they were trained on, and 128 is almost as good as 224 at a third of the cost.

8 From Convolution to the Vision Transformer

The Vision Transformer (Dosovitskiy et al., 2021) cuts a 224×224 image into 16×16 patches: $14 \times 14 = 196$ patches, each flattened to $16 \cdot 16 \cdot 3 = 768$ numbers and mapped by one linear layer to a vector. These 196

vectors are then treated exactly like the words of a sentence by a Transformer (Weeks 13–14). The patch step is the unfold of the CNN note with stride equal to the kernel size — equivalently, one convolution with a 16×16 kernel and stride 16. After that there is no more convolution: the Transformer's attention lets every patch look at every other patch directly, instead of growing a receptive field layer by layer. With enough data this works as well as a CNN; with little data the CNN's built-in locality and weight sharing still help.

9 Further Reading

- d2l, chapter 8 (modern CNNs) and 14.1–14.2 (augmentation, fine-tuning).
- C. Olah, "Understanding Convolutions" and "Conv Nets: A Modular Perspective" (colah.github.io).
- Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, 1998.
- A. Krizhevsky, I. Sutskever, G. Hinton, "ImageNet classification with deep convolutional neural networks," NeurIPS 2012.
- K. Simonyan, A. Zisserman, "Very deep convolutional networks for large-scale image recognition," ICLR 2015 (arXiv:1409.1556).
- S. Ioffe, C. Szegedy, "Batch normalization," ICML 2015 (arXiv:1502.03167).
- S. Santurkar, D. Tsipras, A. Ilyas, A. Madry, "How does batch normalization help optimization?," NeurIPS 2018 (arXiv:1805.11604).
- K. He, X. Zhang, S. Ren, J. Sun, "Deep residual learning for image recognition," CVPR 2016 (arXiv:1512.03385); and "Identity mappings in deep residual networks," ECCV 2016 (arXiv:1603.05027).
- Y. Wu, K. He, "Group normalization," ECCV 2018 (arXiv:1803.08494).
- M. Zeiler, R. Fergus, "Visualizing and understanding convolutional networks," ECCV 2014 (arXiv:1311.2901).
- R. Zhang, "Making convolutional networks shift-invariant again," ICML 2019 (arXiv:1904.11486).
- T. Cohen, M. Welling, "Group equivariant convolutional networks," ICML 2016 (arXiv:1602.07576).
- A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," ICLR 2021 (arXiv:2010.11929).
- Z. Liu et al., "A ConvNet for the 2020s," CVPR 2022 (arXiv:2201.03545).