Architecture Components Practicum

Convolutions, residual connections, layer normalization, weight initialization

CSCI 567 — Machine Learning • Discussion 07 • Oct 23, 2026

Outline

Lecture 07 covered the deep-learning arc conceptually: neural networks as a linear model with a learned feature map, backprop, why deep-net optimization is hard, momentum/Adam, inductive bias.

This session is hands-on: the four **architecture components** that lecture deliberately left to discussion — specific, engineering-level building blocks that address that same difficulty directly.

1. **Convolutions** — why not just flatten the image?
2. **Residual connections** — deeper should never be worse
3. **Layer normalization** — keep activations well-scaled
4. **Weight initialization** — start well-scaled in the first place

1. Convolutions

Why not just flatten the image?

The cost of flattening an image

A $256 \times 256 \times 3$ image, flattened: input dimension 196,608.

a single fully-connected layer to 1000 hidden units:

$$196{,}608 \times 1000 = 196{,}608{,}000$$

$$\approx 1.97 \times 10^8 \text{ parameters — in one layer}$$

A convolutional layer instead

64 filters, each $3 \times 3 \times 3$ (matching the 3 input channels):

$$64 \times (3 \times 3 \times 3) + 64 \text{ (biases)} = 1{,}792 \text{ parameters}$$

$\approx 110{,}000\times$ fewer parameters — and flattening destroys spatial/neighbor structure entirely, which a convolution never does.

Convolution mechanics: the sliding-window sum

Filter size K, stride S, single channel:

$$x^{(1)}_{n,m} = \sum_{k,l=0}^{K-1} f_{k,l} \cdot x^{(0)}_{nS+k,\, mS+l}$$

Slide the same small filter f across the input, one dot product per output position.

Three properties that fall out of this formula

Local connectivity: each output depends only on a small $K \times K$ input neighborhood, not the whole image

Translation equivariance: shift the input $\Rightarrow$ shift the output by the same amount — the same pattern is detected wherever it appears

Weight sharing: the *same* filter f is reused at every position — this is what actually delivers the parameter savings above, not local connectivity alone

Worked example: a convolution by hand

$$X = \begin{pmatrix} 1 & 2 & 0 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \end{pmatrix}, \qquad F = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

F is a vertical-edge filter. Stride 1, no padding.

output size: $(4 - 3 + 1) \times (4 - 3 + 1) = 2 \times 2$

Compute the top-left output entry

$X_{0:3,0:3} \odot F$, summed row by row:

$$\underbrace{1 \cdot 1 + 2 \cdot 0 + 0 \cdot (-1)}_{=1} + \underbrace{0 \cdot 1 + 1 \cdot 0 + 2 \cdot (-1)}_{=-2} + \underbrace{1 \cdot 1 + 0 \cdot 0 + 1 \cdot (-1)}_{=0}$$

$$= 1 - 2 + 0 = -1$$

top-left output entry $= -1$

The full output, verified

$$\begin{pmatrix} -1 & -1 \\ 0 & -2 \end{pmatrix}$$

Every other entry follows the identical pattern: slide the same 3×3 filter, one weight sharing the load across all four output positions instead of four independent 3×3 filters.

Padding

Valid (no padding): output shrinks,
as in the worked example above

Same (zero-pad the border):
output size matches the input

Multi-channel filters, and pooling

Multi-channel, multi-filter: a filter spans *all* input channels at once (e.g. $3 \times 3 \times 3$ for an RGB image); using F different filters produces F output channels

Pooling (max or average): downsamples a feature map — **no learnable parameters at all**, just a fixed aggregation over a small window

Convolution $\Leftrightarrow$ fully-connected equivalence

any convolutional layer is exactly expressible as a heavily constrained — block-structured, weight-shared — fully-connected layer (set most entries of W to zero, tie the rest together to a single filter, repeated at every position)

A CNN is a fully-connected network *plus* a strong architectural prior: "nearby pixels interact, and the same pattern matters everywhere it appears" — Lecture 07's **inductive bias** theme, concretely, before a single gradient step or training example exists.

Backprop aside: weight sharing sums gradients

Many edges share one weight; treat each as independent during backprop, then *sum* the gradients of every edge sharing that weight.

a direct consequence of the multivariable chain rule:
$g(x, y) = f(x, y, x) \Rightarrow \partial g/\partial x = \partial f/\partial x + \partial f/\partial z$

2. Residual Connections

Deeper should never be worse

In principle: a deeper network can always match a shallower one

extra layers can simply learn the **identity** function, reproducing the shallower network exactly

So a deeper network should never do *worse* than a shallower one it contains.

But empirically...

He, Zhang, Ren, and Sun (2016) show plain deep CNNs are *harder to optimize* as they get deeper — training loss can get *worse* with more layers, even though it never has to

This is squarely Lecture 07's vanishing-gradients story: more layers, more multiplicative terms in the backward recursion.

The fix: skip connections

Instead of asking a block of layers to learn the whole mapping $Y = F(X)$, ask it to learn only the *residual* on top of the input, and add the input back:

$$Y = R(X) + X$$

Now the identity function is *trivial* to represent — just set $R \equiv 0$ — rather than something a stack of nonlinear layers has to discover by luck.

Why this helps the backward pass

the "$+X$" term gives the backward pass a direct, additive path — a gradient can flow through it without passing through every intermediate layer's multiplicative chain

Directly countering Lecture 07's vanishing-gradient attenuation.

Used everywhere now

skip connections appear in "almost all modern neural networks, including CNNs and transformers" (He et al., 2016)

Lecture 09's Transformers reuse this exact mechanism.

3. Layer Normalization

Keep activations well-scaled

The shared recipe: normalize, then let the network undo it

1. Normalize using some statistics (mean, variance) to zero-mean, unit-variance.
2. Apply a *learnable* scale γ and shift β, so the network can recover the original scale if that turns out to be what it needs.

batch norm and layer norm differ only in *which* statistics:
batch norm uses statistics across the batch (per feature);
layer norm uses statistics across the features (per example)

Layer normalization: the formula

For one example's activation vector $\mathbf{x}^{(l)} \in \mathbb{R}^K$, with $\hat{\mu}, \hat{\sigma}^2$ its own mean and variance *across its K features*:

$$\mathrm{LN}(\mathbf{x}^{(l)}) = \gamma \odot \frac{\mathbf{x}^{(l)} - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \varepsilon}} + \beta$$

Why layer norm, specifically

no dependence on other examples in the batch — same formula at training *and* inference time, unlike batch norm (which needs running batch statistics at test time)

This is why layer norm, not batch norm, is the default for transformers and other sequence models (Ba, Kiros, and Hinton, 2016) — a direct forward-pointer to Lecture 09.

4. Weight Initialization

Start well-scaled in the first place

Why initialization matters this much

Lecture 07 named "proper initialization" as one of three remedies for vanishing/exploding gradients.

He, Zhang, Ren, and Sun (2015) show empirically that even a **factor-of-2** difference in initialization scale can be the difference between a deep ReLU network training at all and not training at all

He initialization: the goal

Choose weights $W \sim \mathcal{N}(0, \sigma^2 I_K)$ (fan-in K) so that:

if the pre-activation $\mathbf{z}^{(l)}$ has unit variance per entry, the *next* layer's pre-activation does too — neither shrinking nor growing as signal passes through the network

The ReLU correction

For a ReLU layer, $x_i = \max(0, z_i)$ zeroes out roughly half its inputs on average:

$$\mathbb{E}[x_i^2] = \tfrac{1}{2}\mathbb{E}[z_i^2] = \tfrac{1}{2}$$

(unit-variance z, zero-mean).

Solving for σ

With K independent inputs and independent weight entries:

$$\mathrm{Var}(z_j') = \sum_{i=1}^{K} \mathrm{Var}(W_{ij})\,\mathbb{E}[x_i^2] = K \cdot \sigma^2 \cdot \tfrac{1}{2}$$

setting $\mathrm{Var}(z_j') = 1$: $\quad \sigma = \sqrt{2/K}$

Worked example

Fan-in $K = 512$:

$$\sigma = \sqrt{2/512} = \sqrt{1/256} = 0.0625 \text{ exactly}$$

Xavier (Glorot) initialization

He initialization is ReLU-specific. Glorot and Bengio (2010) derive the earlier, activation-symmetric version (tanh/linear, fan-in *and* fan-out $n_{\text{in}}, n_{\text{out}}$, no "half the units die" correction):

$$W_{i,j} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}},\ \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right)$$

or, equivalently, $\mathcal{N}\left(0,\ \frac{2}{n_{\text{in}}+n_{\text{out}}}\right)$

Rule of thumb: **Xavier for tanh/sigmoid, He for ReLU.**

Tying it back to Lecture 07

Lecture 07 named it	This practicum built it
Vanishing/exploding gradients, remedy 2	Xavier/He weight initialization (derived)
Vanishing/exploding gradients, remedy 3	Residual connections ($Y = R(X) + X$)
Non-uniform smoothness / ill-conditioning	Layer normalization (keeps activations well-scaled)
Inductive bias	Convolution = FC + a strong architectural prior

Coming up

Coming up: Lecture 08 — Ensemble Methods. Lecture 09 — Self-Supervised Learning and Transformers, reuses residual connections and layer normalization directly, exactly as built here.

References

- He, Zhang, Ren, Sun. *Deep Residual Learning for Image Recognition.* CVPR 2016.
- He, Zhang, Ren, Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* ICCV 2015.
- Glorot, Bengio. *Understanding the Difficulty of Training Deep Feedforward Neural Networks.* AISTATS 2010.
- Ba, Kiros, Hinton. *Layer Normalization.* arXiv:1607.06450, 2016.

Convolutions and residual connections adapted from EPFL's `ML_course`; weight initialization and layer normalization drawn from the same source plus independently verified citations. Full notes:

`references/sources/epfl-l07-discussion-architecture-components.md`,

`references/sources/lecture07-additional-verified-facts.md`