Neural Networks for Data Science

Master's Degree in Data Science

Building deeper models

Lecturer: S. Scardapane



SAPIENZA
UNIVERSITÀ DI ROMA

Why depth is hard

Going deeper

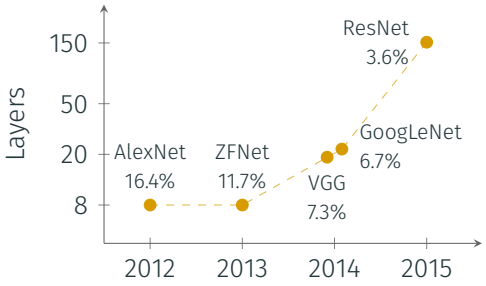
ImageNet (2009)
About 14 million images, labelled into some 22,000 categories of the WordNet hierarchy.

ILSVRC (2010-2017)
A yearly competition on a subset: 1,000 classes, 1.28 million training and 50,000 validation images.

The **top-5 error** is the fraction of test images whose true class is not among the five classes with the highest score.

Why five guesses? An image often contains several objects but has a single label, and some classes are very close: there are 120 dog breeds alone.

Deng et al. (2009), CVPR; Russakovsky et al. (2015), IJCV.



ILSVRC winners (and VGG): depth and top-5 error.

Every jump in depth needed a new trick to keep training working.

2012 ReLU, dropout, data augmentation.

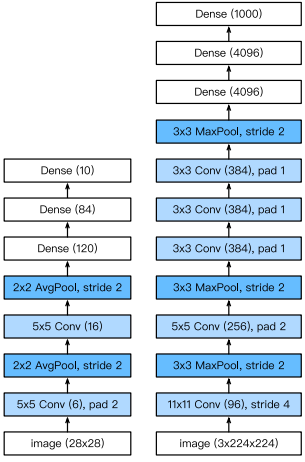
2014 Starting from a trained shallow network (VGG); extra classifiers halfway through the network (GoogLeNet).

2015 He initialization, **batch normalization**, gated shortcuts, **residual connections**.

2016–17 Stochastic depth, dense connections (DenseNet), squeeze-and-excitation (SENet, the last winner: 2.3%).

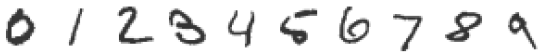
Not all tricks remain useful today, so we only cover the most important subset.

Szegedy et al. (2015), Table 2; Simonyan and Zisserman (2015); He et al. (2015); Ioffe and Szegedy (2015); Srivastava et al. (2015); He et al. (2016); Huang et al. (2016, 2017); Hu et al. (2018).



LeNet-5 (1998) and AlexNet (2012), simplified.

In 1998, **LeNet-5** already read handwritten digits (**MNIST**) with convolutions, pooling and a small classifier.

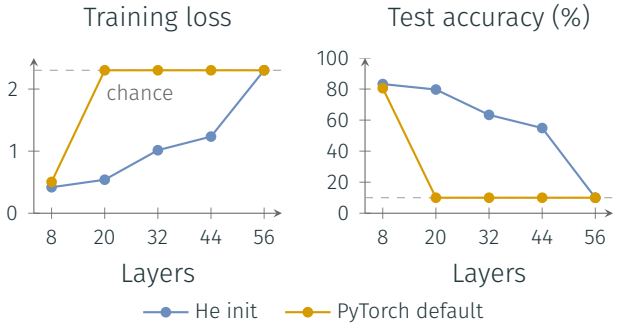


AlexNet is essentially a bigger LeNet! What changed in 2012 was GPUs, a large labelled dataset and a few training tricks.

LeCun et al. (1998), Proc. IEEE; Krizhevsky et al. (2012), NeurIPS. Figure: Dive into Deep Learning, Sec. 8.1 (CC BY-SA 4.0).

A plain CNN on **CIFAR-10** (60,000 small color images, 10 classes): blocks of two 3×3 convolutions with ReLU, global pooling, a linear classifier.

We change only the depth and the initialization, keeping the best of three learning rates.



Deeper plain networks are worse on the **training** set, so this is not overfitting: we simply cannot optimize them.

Krizhevsky (2009), *Learning multiple layers of features from tiny images.*

A good initialization (such as He's) only fixes the scale of the activations at the first step. Nothing keeps it there during training, and the gradient must cross every layer to reach the first ones.

We look at three possible solutions to improve training at scale:

Normalization
Keep the scale of the activations under control during training.

Residuals
Make the identity the default, so every block only adds a small change.

Regularization
Once the network trains, make it generalize.

Normalization

Normalization layers

Given a dataset $\mathbf{X} \sim (N, D)$, we usually standardize each column (feature) to zero mean and unit variance:

$$[\mathbf{X}']_{i,j} = \frac{[\mathbf{X}]_{i,j} - \mu_j}{\sigma_j}, \qquad \mu_j = \frac{1}{N}\sum_i [\mathbf{X}]_{i,j}, \quad \sigma_j^2 = \frac{1}{N}\sum_i \left([\mathbf{X}]_{i,j} - \mu_j\right)^2.$$

For least squares, the Hessian of the loss is $\frac{2}{N}\mathbf{X}^\top\mathbf{X}$. After standardization its diagonal is constant, so the problem is usually better **conditioned**: remember the elongated valley in the preliminaries.

Idea: do the same inside the network, on the output of every layer. The difficulty is that these outputs change at every step of training.

Ideally, we would standardize each unit with its mean and variance over the whole training set. But these change at every step, and recomputing them would need a full pass over the data (with gradients through all of it). Instead, we estimate them on the mini-batch. For outputs $\mathsf{Z} \sim (B, D)$:

$$\widetilde{\mu}_j = \frac{1}{B}\sum_b [\mathsf{Z}]_{b,j}\,, \quad \widetilde{\sigma}_j^2 = \frac{1}{B}\sum_b \left([\mathsf{Z}]_{b,j} - \widetilde{\mu}_j\right)^2, \quad [\mathrm{BN}(\mathsf{Z})]_{b,j} = \gamma_j \frac{[\mathsf{Z}]_{b,j} - \widetilde{\mu}_j}{\sqrt{\widetilde{\sigma}_j^2 + \varepsilon}} + \beta_j\,.$$

A **batch normalization** (BN) layer standardizes each column, then learns a new mean β_j and scale γ_j for it ($2D$ trainable parameters). It usually goes between a linear layer and its activation, whose bias becomes redundant.

Ioffe and Szegedy (2015), *Batch normalization: accelerating deep network training by reducing internal covariate shift*, ICML.

After a convolutional layer, the outputs on a mini-batch are $\mathsf{Z} \sim (B, H, W, C)$. Each channel is the same filter applied at every position, so we treat every position of every image as one more sample of that channel:

$$\widetilde{\mu}_c = \frac{1}{BHW} \sum_{b,i,j} [\mathsf{Z}]_{b,i,j,c}\,, \qquad \widetilde{\sigma}_c^2 = \frac{1}{BHW} \sum_{b,i,j} \left([\mathsf{Z}]_{b,i,j,c} - \widetilde{\mu}_c\right)^2.$$

Each channel is then standardized and rescaled as before, with one pair γ_c, β_c per channel ($2C$ parameters).

A channel plays the role of a feature: the mean is taken over the mini-batch *and* over all spatial positions.

In PyTorch, this is `nn.BatchNorm2d(C)` (with channels first).

During training, each output depends on the rest of its mini-batch. At inference we want a deterministic prediction, so we need fixed values μ_j and σ_j^2.

We could compute them on the training set after training. In practice, BN keeps a running average of the mini-batch statistics, at almost no cost:

$$\mu_j \leftarrow (1-m)\,\mu_j + m\,\widetilde{\mu}_j \qquad (m = 0.1 \text{ in PyTorch})$$

and similarly for σ_j^2, with the unbiased estimate $\frac{B}{B-1}\widetilde{\sigma}_j^2$. They replace the estimates at inference.

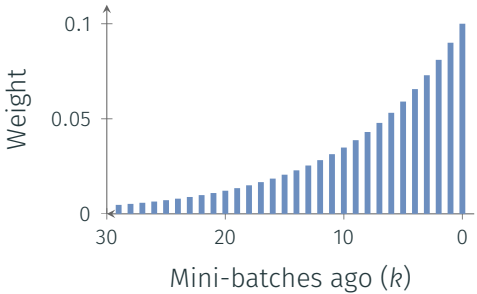
In PyTorch, `model.train()` and `model.eval()` switch between the two modes: forgetting `model.eval()` is a classic bug.

Unrolling the update over past mini-batches:

$$\mu_j = m \sum_{k \geq 0} (1-m)^k \, \widetilde{\mu}_j^{(t-k)} .$$

Older estimates count exponentially less: an **exponential moving average**. With $m = 0.1$, the last 10 mini-batches make up 65% of the value, the last 50 over 99%.

Example: starting from $\mu_j = 0$, if every mini-batch gives $\widetilde{\mu}_j = 1$, the running mean is 0.1 after one step, 0.65 after 10 and 0.99 after 50.



Weight $m(1-m)^k$ of each past estimate ($m = 0.1$).

BN usually sits between a linear layer and its activation: $\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b}))$. At inference, μ_j and σ_j^2 are fixed, so BN is just an elementwise affine map:

$$\mathrm{BN}(\mathbf{z}) = \mathbf{a} \odot \mathbf{z} + \mathbf{c}\,, \qquad a_j = \frac{\gamma_j}{\sqrt{\sigma_j^2 + \varepsilon}}\,, \quad c_j = \beta_j - a_j \mu_j\,.$$

Two affine maps in a row are one affine map, so we can fold BN into the layer:

$$\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathbf{W}'\mathbf{x} + \mathbf{b}'\,, \qquad \mathbf{W}' = \mathrm{diag}(\mathbf{a})\,\mathbf{W}\,, \quad \mathbf{b}' = \mathbf{a} \odot \mathbf{b} + \mathbf{c}\,.$$

After training, BN costs nothing at inference. The same works for a convolution, one output channel at a time.

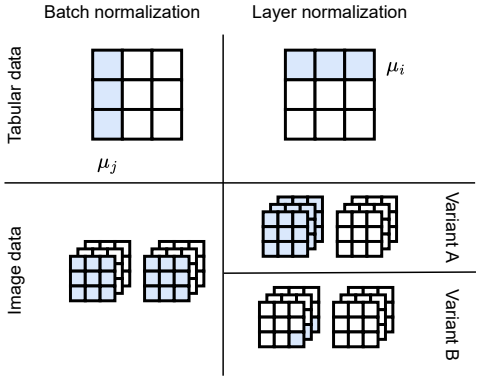
Tying each example to its mini-batch is costly: noisy estimates with small mini-batches, a poor fit for variable-length sequences or many devices, and different behavior in training and inference. **Layer normalization** (LN) computes the statistics of each example over its own features:

$$\widetilde{\mu}_b = \frac{1}{D}\sum_j [\mathsf{Z}]_{b,j}\,, \quad \widetilde{\sigma}_b^2 = \frac{1}{D}\sum_j \left([\mathsf{Z}]_{b,j} - \widetilde{\mu}_b\right)^2, \quad [\mathrm{LN}(\mathsf{Z})]_{b,j} = \gamma_j \frac{[\mathsf{Z}]_{b,j} - \widetilde{\mu}_b}{\sqrt{\widetilde{\sigma}_b^2 + \varepsilon}} + \beta_j\,.$$

	Statistics over	Parameters γ, β
BN	each feature, across the mini-batch	one pair per feature
LN	each example, across its features	one pair per feature

Nothing depends on other examples: the same computation in training and inference, even with $B = 1$. LN is the standard in transformers.

Ba et al. (2016), *Layer normalization*.



In blue: values sharing one mean and variance.

For images, LN has two common variants:

A: statistics over the whole image, all $H \times W \times C$ values (`nn.LayerNorm` learns one γ, β per value).

B: statistics over the C channels at each position, with $\gamma, \beta \in \mathbb{R}^C$, as in ConvNeXt. Transformers do the same on each token.

Wu and He (2018), *Group normalization*, ECCV.

Most recent LLMs skip the mean: $\mathbf{RMSNorm}(\mathbf{z}) = \boldsymbol{\gamma} \odot \mathbf{z} / \sqrt{\frac{1}{D}\sum_j z_j^2 + \varepsilon}$.

Why can we drop the mean? Centering is a linear map: $\mathbf{z} - \mathrm{mean}(\mathbf{z})\,\mathbf{1} = \mathbf{P}\mathbf{z}$, with $\mathbf{P} = \mathbf{I} - \frac{1}{D}\mathbf{1}\mathbf{1}^\top$. When LN follows a linear layer, $\mathbf{P}$ can be folded into it:

$$\mathrm{LN}(\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathrm{RMSNorm}(\mathbf{P}\mathbf{W}\mathbf{x} + \mathbf{P}\mathbf{b}) + \boldsymbol{\beta}\,.$$

So centering adds nothing that the previous layer cannot learn. The only nonlinear part of LN is the division by the norm, which RMSNorm keeps.

In practice, RMSNorm matches LN in accuracy and reduces running time by 7–64% across models.

Normalization

Why normalization works

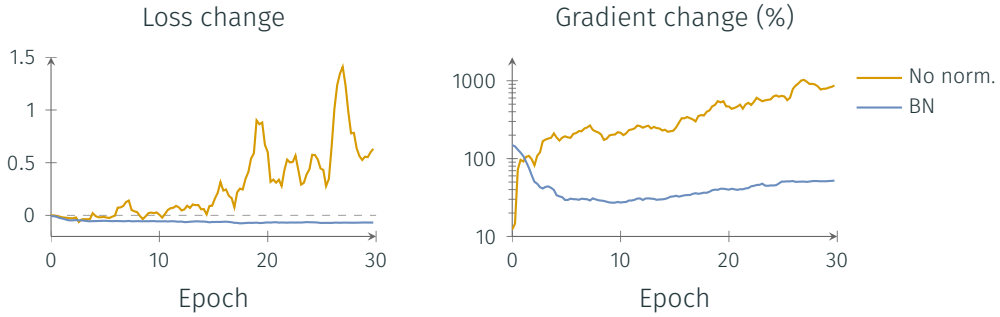
The original explanation was **internal covariate shift**: each layer sees inputs whose distribution keeps changing as the previous layers train, and BN stabilizes them.

Later experiments challenged this explanation: injecting noise after BN to shift the distributions again barely hurts training. A better-supported view is that BN makes the loss landscape **smoother**, so the gradient stays predictive over longer steps.

In practice, the most robust effect is that normalized networks train stably with much **larger learning rates**.

Santurkar et al. (2018), *How does batch normalization help optimization?*, NeurIPS; Bjorck et al. (2018), *Understanding batch normalization*, NeurIPS; Lipton and Steinhardt (2018), *Troubling trends in machine learning scholarship*.

Every 100 training steps ($\eta = 0.02$), we try a step of 0.05 along the gradient:



With BN, the gradient stays **predictive over longer steps**.

Measurement after Santurkar et al. (2018), *How does batch normalization help optimization?*, NeurIPS, Fig. 4.

A unit produces $\mathbf{z} = \mathbf{X}\mathbf{w}$, followed by BN. If we scale its weights by $a > 0$, the batch mean and standard deviation scale by a:

$$\mu(a\mathbf{z}) = a\mu(\mathbf{z}), \qquad \sigma(a\mathbf{z}) = a\sigma(\mathbf{z}).$$

We neglect ε and keep γ, β fixed. The scale cancels:

$$\mathrm{BN}(a\mathbf{z}) = \gamma \frac{a\mathbf{z} - a\mu(\mathbf{z})\mathbf{1}}{a\sigma(\mathbf{z})} + \beta\mathbf{1} = \mathrm{BN}(\mathbf{z}).$$

The predictions, and hence the data loss, do not change: $\mathcal{L}(a\mathbf{w}) = \mathcal{L}(\mathbf{w})$. Only the **direction** of these weights matters to the function.

For LN, the same holds only when we scale the *whole* preceding layer, weights and bias together: we come back to this at the end.

Take $\mathbf{g}(\mathbf{w}) = \nabla\mathcal{L}(\mathbf{w})$ and differentiate $\mathcal{L}(a\mathbf{w}) = \mathcal{L}(\mathbf{w})$:

1. In the scalar a, at $a = 1$:

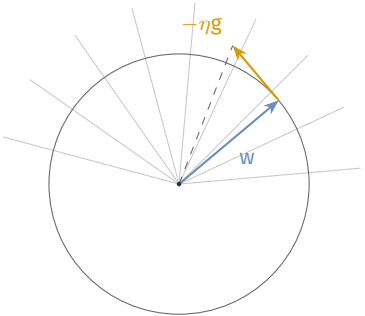
$$\frac{\mathrm{d}}{\mathrm{d}a}\mathcal{L}(a\mathbf{w})\bigg|_{a=1} = \mathbf{w}^\top \mathbf{g}(\mathbf{w}) = 0\,.$$

The gradient is **orthogonal to $\mathbf{w}$**, tangent to its sphere.

2. In the vector $\mathbf{w}$, with a fixed:

$$a\,\mathbf{g}(a\mathbf{w}) = \mathbf{g}(\mathbf{w}) \quad\Longrightarrow\quad \mathbf{g}(a\mathbf{w}) = \frac{\mathbf{g}(\mathbf{w})}{a}\,.$$

The chain rule supplies the factor a on the left. Longer weights get **smaller gradients**.



The loss is constant along each gray ray, so the gradient has no component along $\mathbf{w}$.

We separate length and direction: $\mathbf{w} = r\mathbf{u}$, with $r = \|\mathbf{w}\|$ and $\|\mathbf{u}\| = 1$. The previous identity gives $\mathbf{g}(r\mathbf{u}) = \mathbf{g}(\mathbf{u})/r$. A gradient-descent step is therefore

$$\mathbf{w}^+ = r\mathbf{u} - \frac{\eta}{r}\mathbf{g}(\mathbf{u}) = r\left(\mathbf{u} - \frac{\eta}{r^2}\mathbf{g}(\mathbf{u})\right) .$$

We divide by its new norm to recover the direction:

$$\mathbf{u}^+ = \frac{\mathbf{u} - \eta_{\text{eff}}\mathbf{g}(\mathbf{u})}{\|\mathbf{u} - \eta_{\text{eff}}\mathbf{g}(\mathbf{u})\|} , \qquad \eta_{\text{eff}} = \frac{\eta}{r^2} .$$

For a small step, $\mathbf{u}^+ - \mathbf{u} \approx -\eta_{\text{eff}}\mathbf{g}(\mathbf{u})$. Doubling r leaves the function unchanged, but divides this change in direction by four.

van Laarhoven (2017), *L2 regularization versus batch and weight normalization*. Gradient descent, without momentum.

Since $\mathbf{w} \perp \mathbf{g}$, plain gradient steps only grow r, so $\eta_{\text{eff}} = \eta/r^2$ keeps falling. **Weight decay**, $\mathbf{w}^+ = (1 - \eta\lambda)\mathbf{w} - \eta\mathbf{g}(\mathbf{w})$, pulls the other way:

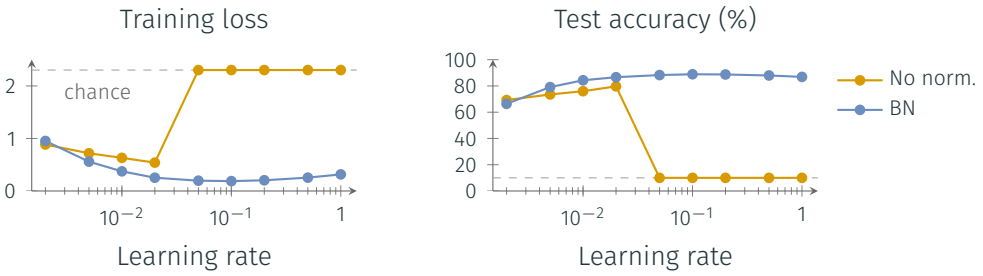
$$(r^+)^2 = \underbrace{(1-\eta\lambda)^2 r^2}_{\text{decay shrinks the norm}} + \underbrace{\eta^2 \|\mathbf{g}(\mathbf{w})\|^2}_{\text{gradient step grows it}}$$

When the two balance, each step turns $\mathbf{u}$ by about $\sqrt{2\eta\lambda}$.

Here decay does not change the function: it sets the **effective learning rate**, through $\eta\lambda$.

van Laarhoven (2017), *L2 regularization versus batch and weight normalization*; Li et al. (2020), *Reconciling modern deep learning with traditional optimization analyses: the intrinsic learning rate*, NeurIPS.

The same 20-layer CNN, with and without BN; we change only the learning rate:



Best accuracy: 79.7% without BN ($\eta = 0.02$), 88.9% with BN ($\eta = 0.1$).

A step too large grows $\|\mathbf{w}\|$ and lowers $\eta/\|\mathbf{w}\|^2$: BN tunes its own rate.

Arora et al. (2019), *Theoretical analysis of auto rate-tuning by batch normalization*, ICLR.

Standardizing each column suits inputs measured in different units. LN averages across the features of one example: why is this reasonable in hidden layers?

- **Hidden units are interchangeable.** They come from the same kind of linear map and start identically distributed (same reasoning as He initialization).
- **LN fixes the length of the representation**, as we see in the next slide.

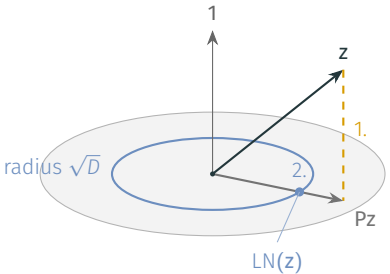
This assumes units contribute in similar ways. In CNNs, units near the image border behave differently: there, BN still works better.

Ba et al. (2016), *Layer normalization*, Secs. 5.1 and 6.7.

1. Centering. Given $\mathbf{z} \in \mathbb{R}^3$, centering with $\mathbf{z}_c = \mathbf{z} - \mu\mathbf{1}$ makes the vector orthogonal to $\mathbf{1}$ ($\mathbf{z}_c^\top \mathbf{1} = 0$) and is equivalent to projecting onto the subspace perpendicular to $\mathbf{1}$ (grey plane on the right).

2. Scaling. Since $\sigma^2 = \|\mathbf{z}_c\|^2/D$, dividing by σ makes a second projection onto the sphere of radius $\sqrt{D}$ (blue circle on the right).

Neglecting ε, LN is invariant to two transformations: adding a constant to every coordinate and multiplying the vector by a positive scalar: $\mathrm{LN}(a\mathbf{z} + c\mathbf{1}) = \mathrm{LN}(\mathbf{z})$ for $a > 0$.



For $D = 3$. **1.** Centering drops $\mathbf{z}$ onto the plane where entries sum to zero (shaded). **2.** Scaling moves it along its ray to the circle of radius $\sqrt{D}$ (blue).

	BN	LN
Normalizes	a feature, across the batch	an example, across its features
Ignores the scale of	each unit's weights	the whole layer's weights
Effective learning rate	one per unit, $\eta/\|\mathbf{w}\|^2$	one per layer, $\eta/\|\mathbf{W}\|^2$
At inference	running averages	same as in training

A predictive gradient and a self-tuning rate allow **larger learning rates**.

Still, with BN, 56 plain layers lose to 20 (81.6% vs 88.9%). Next: **residual connections**.

Ba et al. (2016), *Layer normalization*, Sec. 5.1 and Table 1; van Laarhoven (2017).

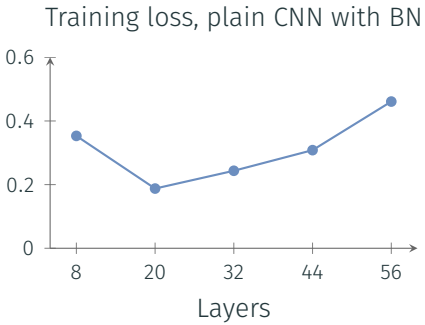
Residual connections

Residual blocks

Take a network $h(\mathbf{x})$ that works well, and add layers on top of it: $f(\mathbf{x}) = g(h(\mathbf{x}))$.

If the new layers learned the identity, $g(\mathbf{x}) \approx \mathbf{x}$, the deeper network would be at least as good as h.

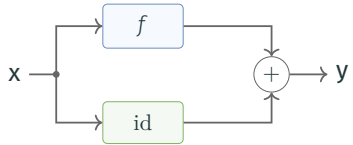
In practice, even with BN, the **training** loss grows with depth.



A stack of convolutions and ReLUs struggles to learn the identity: the deeper network contains a good solution that SGD does not find.

In a **residual network**, we add a **skip connection** around each block f (for example, two convolutions):

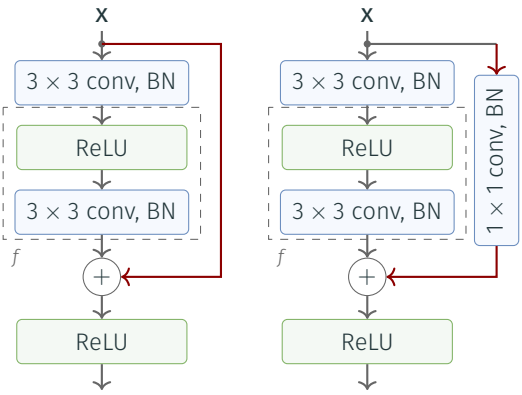
$$r(\mathbf{x}) = f(\mathbf{x}) + \mathbf{x}\,.$$



r is a **residual block**: f is the residual branch, $\mathbf{x}$ the skip path. With $f = 0$ the block is the identity, so every block only learns a **change** to its input.

If $\mathbf{x}$ and $f(\mathbf{x})$ have different shapes, we map $\mathbf{x}$ with a linear layer (a 1×1 convolution for images). The same block is in every modern architecture, transformers included.

He et al. (2016), *Deep residual learning for image recognition*, CVPR.



The branch f is two 3×3 convolutions with BN. The last ReLU comes **after** the sum: inside f, every block could only add a positive change.

When a stage halves the resolution and doubles the channels, the skip path becomes a 1×1 convolution with stride 2 (right).

A **ResNet** stacks these blocks in stages. Our CIFAR-10 network has three stages of n blocks (16, 32 and 64 channels), for $6n + 2$ layers.

He et al. (2016), *Deep residual learning for image recognition*, CVPR, Sec. 4.2; after Dive into Deep Learning, Sec. 8.6 (CC BY-SA 4.0).

Residual connections

Why residuals work

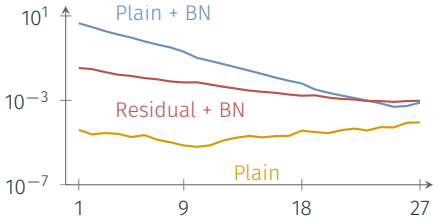
The Jacobian of a residual block is

$$\partial r(\mathbf{x}) = \partial f(\mathbf{x}) + \mathbf{I}\,.$$

In the backward pass, the VJP adds the incoming gradient back unchanged:

$$\mathbf{v}^\top \partial r(\mathbf{x}) = \underbrace{\mathbf{v}^\top \partial f(\mathbf{x})}_{\text{original VJP}} + \mathbf{v}\,.$$

Gradient at initialization, 56 layers



With BN alone, the gradient grows 6,000 times towards the input.

The gradient reaches the first blocks through the skip path, whatever happens in the branches.

With He initialization and no BN, a branch keeps the scale of its input, so every block roughly doubles the variance:

$$\mathrm{Var}(\mathbf{h}_{l+1}) \approx \mathrm{Var}(\mathbf{h}_l) + \mathrm{Var}(f(\mathbf{h}_l)) \approx 2\,\mathrm{Var}(\mathbf{h}_l)\,.$$

After 27 blocks, a factor of $2^{27} \approx 10^8$: our 56-layer network never trains. Two fixes keep each branch small:

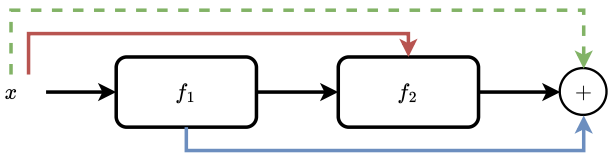
BN in the branch
The branch has unit variance while the skip path grows, so block l adds only about $1/l$ of the variance.

Start from the identity
Zero the last layer of each branch, or use $\mathbf{x} + \alpha f(\mathbf{x})$ with a learned $\alpha = 0$ at the start.

Zeroing the last convolution is enough: 88.1% without BN ($\eta = 0.02$), 91.4% with BN ($\eta = 0.1$).

De and Smith (2020), *Batch normalization biases residual blocks towards the identity function in deep networks*, NeurIPS; Goyal et al. (2017); Zhang et al. (2019), Fixup; Bachlechner et al. (2021), ReZero.

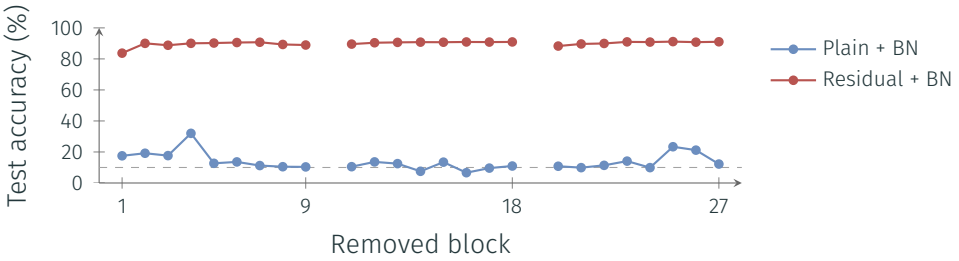
Two stacked residual blocks create four *paths* from input to output:



With L blocks there are 2^L paths, of every length from 0 to L. The network behaves like an **ensemble** of shallower networks, which makes it robust. This view becomes very important in transformers.

Veit et al. (2016), *Residual networks behave like ensembles of relatively shallow networks*, NeurIPS.

After training, we delete one block at a time and measure the test accuracy (blocks that change the shape cannot be removed):



Without one block, the residual network loses under 1 point (median); the plain network falls to 7–32%.

Measurement after Veit et al. (2016), NeurIPS, Fig. 3.

With T residual blocks, the hidden state evolves as

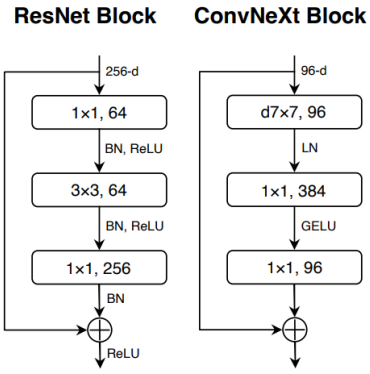
$$\mathbf{h}_t = \mathbf{h}_{t-1} + f(\mathbf{h}_{t-1}), \qquad t = 1, \dots, T,$$

with $\mathbf{h}_0 = \mathbf{x}$. In the limit $T \to \infty$, each block only makes an infinitesimal step. We cannot have infinitely many layers, but we can condition a single block on t:

$$\frac{\mathrm{d}\mathbf{h}(t)}{\mathrm{d}t} = f(\mathbf{h}(t), t).$$

An ODE solver computes the output, and we call these models neural ODEs.

Chen et al. (2018), *Neural ordinary differential equations*, NeurIPS.

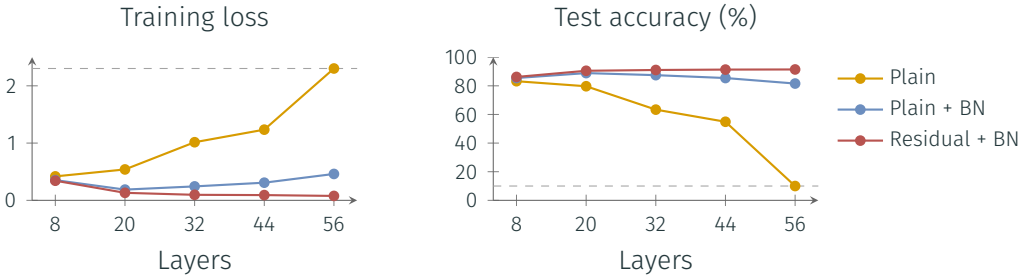


Deep ResNets use a **bottleneck** block (left): a 1×1 convolution reduces the channels, the 3×3 works on fewer of them, and a second 1×1 restores them.

ConvNeXt (right) updates the ResNet with design choices from transformers: a depthwise 7×7 convolution, then an MLP on each position, with LN and GELU. Its largest V2 model reaches 88.9% top-1 on ImageNet.

With enough data and compute, CNNs match vision transformers: **NFNets**, residual networks without normalization, reach 90.4%.

Liu et al. (2022), *A ConvNet for the 2020s*, CVPR, Fig. 4; Woo et al. (2023), *ConvNeXt V2*, CVPR; Smith et al. (2023), *ConvNets match vision transformers at scale.*



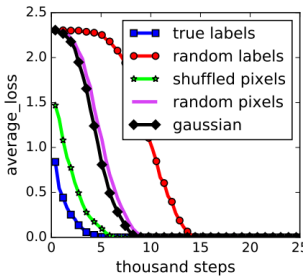
With residual connections, depth helps again: from 86.2% (8 layers) to 91.4% (56 layers).

The networks now train. Next: making them generalize, with **regularization**.

Regularization

Overfitting

A model **overfits** when its performance keeps improving on the training set while it gets worse on a separate validation set.



Deep networks are strangely resilient to classical overfitting, yet they can **memorize** anything: trained long enough, a CNN fits CIFAR-10 perfectly even with random labels (red) or random pixels.

So the architecture alone does not explain why they generalize with real labels.

Zhang et al. (2017), *Understanding deep learning requires rethinking generalization*, ICLR, Fig. 1a.

In optimization, we would stop when the loss stops changing, $|\mathcal{L}(\mathbf{w}_t) - \mathcal{L}(\mathbf{w}_{t-1})| \leq \varepsilon$. This is rarely done: large networks are limited by compute, small ones by overfitting. With **early stopping** we watch a validation set instead:

1. we keep a portion of the data as the **validation set**;
2. after each epoch, we compute the validation loss (or accuracy);
3. if it has not improved for a number of epochs (the **patience**), we stop and keep the best checkpoint.

Early stopping is common with small datasets, and shows the difference between pure optimization and learning.

LLMs see most of their data once: the compute budget decides when to stop.

Regularization

Penalties and parameterizations

Large weights make the network less *smooth*: small changes in the input give sharp changes in the output. We met two views of weight decay, $\mathbf{w} \leftarrow (1-\eta\lambda)\mathbf{w} - \eta\nabla\mathcal{L}(\mathbf{w})$:

A penalty (MLP lecture)
$\lambda\|\mathbf{w}\|^2$ prefers smaller weights: with $\lambda = 0$ nothing changes, with λ too large all weights go to zero.

A learning rate (normalization)
Before BN or LN, the function ignores $\|\mathbf{w}\|$: decay only sets $\eta/\|\mathbf{w}\|^2$.

In a modern network both are at work. With Adam, we use **AdamW**, which applies the decay directly instead of through the gradient. Biases and normalization parameters are usually not decayed.

Loshchilov and Hutter (2019), *Decoupled weight decay regularization*, ICLR.

Many other penalties exist. Replacing the squared norm with the sum of absolute values,

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) + C \sum_j |w_j| ,$$

leads to **sparse** solutions, with many weights exactly zero.

Penalizing whole groups of weights (all the outgoing weights of a unit) removes entire units, and so shrinks the network.

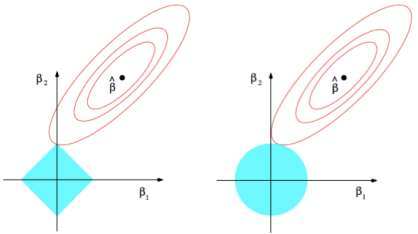


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions $|\beta_1| + |\beta_2| \le t$ and $\beta_1^2 + \beta_2^2 \le t^2$, respectively, while the red ellipses are the contours of the least squares error function.*

The corners of the ℓ_1 ball (left) lie on the axes, where some weights are zero.

Hastie et al., *The Elements of Statistical Learning*, Fig. 3.11; Scardapane et al. (2017), *Group sparse regularization for deep neural networks*, Neurocomputing.

The ℓ_1 norm $\|\mathbf{w}\|_1 = \sum_j |w_j|$ is hard to optimize with SGD, because it is not differentiable at 0. We can *reparameterize* $\mathbf{w} = \mathbf{a} \odot \mathbf{b}$ and use weight decay on the new vectors:

$$\mathcal{L}(\mathbf{w}) + C\|\mathbf{w}\|_1 \quad \rightarrow \quad \mathcal{L}(\mathbf{a} \odot \mathbf{b}) + \frac{C}{2}\left(\|\mathbf{a}\|^2 + \|\mathbf{b}\|^2\right).$$

The minima are the same (since $\frac{1}{2}(a^2 + b^2) \geq |ab|$), but the new problem is smooth and much easier for SGD.

How we **parameterize** the same function can change training a lot, as we saw with normalization.

Ziyin and Wang (2022), *Sparsity by redundancy: solving L_1 with a simple reparametrization*, arXiv:2210.01212.

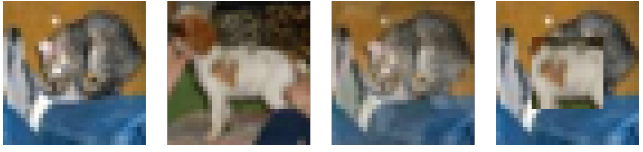
Regularization

More data augmentation

We can also augment by **combining** two examples $(\mathbf{x}_1, \mathbf{y}_1)$ and $(\mathbf{x}_2, \mathbf{y}_2)$, with one-hot labels. **MixUp** takes a convex combination with a random $\lambda \in [0, 1]$:

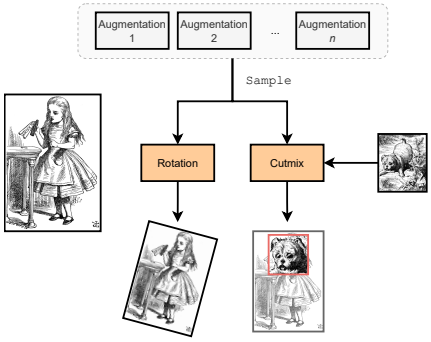
$$\mathbf{x} = \lambda \mathbf{x}_1 + (1 - \lambda)\mathbf{x}_2 \,, \qquad \mathbf{y} = \lambda \mathbf{y}_1 + (1 - \lambda)\mathbf{y}_2 \,.$$

MixUp images look unnatural. **CutMix** pastes a random patch of $\mathbf{x}_2$ instead, $\mathbf{x} = (1 - \mathbf{M}) \odot \mathbf{x}_1 + \mathbf{M} \odot \mathbf{x}_2$, with λ the fraction of the image left to $\mathbf{x}_1$.



cat dog MixUp: 70% cat CutMix: 75% cat

Zhang et al. (2018), *mixup: beyond empirical risk minimization*, ICLR; Yun et al. (2019), *CutMix*, ICCV.



In practice, we combine many transformations, and for each mini-batch we **sample** which ones to apply.

RandAugment keeps this simple: from a fixed list (rotations, shears, color changes, ...) it picks *N* transformations at random, all with the same magnitude *M*. Only *N* and *M* need tuning.

Designing good augmentations is still an active research field.

Cubuk et al. (2020), *RandAugment: practical automated data augmentation with a reduced search space*, CVPR Workshops.

Augmentation can be part of the data pipeline, or a layer of the model:

```
# In the data pipeline
from torchvision.transforms import v2
augment = v2.Compose([
    v2.RandomCrop(32, padding=4),
    v2.RandomHorizontalFlip(),
])
train_data = CIFAR10(..., transform=augment)
```

```
# As a layer of the model
class Augment(nn.Module):
    def forward(self, x):
        if self.training:
            x = random_shift(x)
        return x

model = nn.Sequential(Augment(), cnn)
```

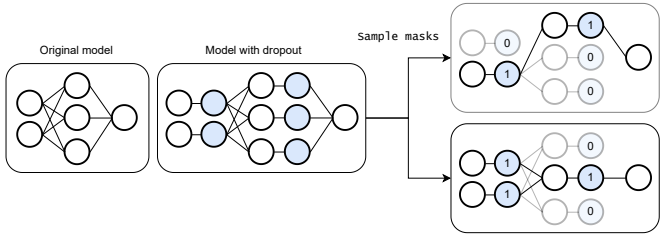
As a layer, augmentation behaves differently in training and inference, like BN: `model.train()` and `model.eval()` switch it on and off.

Regularization

Dropout

Why does augmentation help? Small **perturbations** of the input make the network more robust: training with noise is provably a form of regularization.

Dropout brings the same idea inside the network: we perturb the hidden layers by randomly *dropping* some of their units.



Each mask selects a sub-network: we train many of them, with shared weights.

Bishop (1995), *Training with noise is equivalent to Tikhonov regularization*, Neural Computation.

Take the output $\mathbf{H} \sim (B, D)$ of a layer with D units on a mini-batch of B examples. During training, dropout replaces it with

$$\widetilde{\mathbf{H}} = \mathbf{H} \odot \mathbf{M}\,,$$

where every entry of $\mathbf{M} \sim (B, D)$ is drawn independently: 0 with probability p, 1 with probability $1 - p$.

If $[\mathbf{M}]_{b,j} = 0$, unit j is dropped for example b. Every example and every step sees a different sub-network.

Dropout can also be applied to the input of the network.

Srivastava et al. (2014), *Dropout: a simple way to prevent neural networks from overfitting*, JMLR.

At inference, we want a deterministic output. The simplest choice is the expected value of the dropout *layer*, which we can compute in closed form:

$$\mathbb{E}[\widetilde{\mathsf{H}}] = p \cdot \mathbf{0} + (1-p)\,\mathsf{H} = (1-p)\,\mathsf{H}\,.$$

It is more convenient to rescale during training instead (**inverted dropout**):

$$\widetilde{\mathsf{H}} = \frac{\mathsf{H} \odot \mathsf{M}}{1-p} \quad \Longrightarrow \quad \mathbb{E}[\widetilde{\mathsf{H}}] = \mathsf{H}\,.$$

With inverted dropout, at inference we simply remove the layer. This is what `nn.Dropout(p)` does, switched by `model.eval()`.

With dropout, the output $f(\mathbf{x}; \mathbf{M})$ is a random variable that depends on the mask. A more principled prediction is the expected value over the masks of the whole *network*:

$$\hat{\mathbf{y}} = \mathbb{E}_{p(\mathbf{M})}\left[f(\mathbf{x}; \mathbf{M})\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}; \mathbf{M}_i)\,, \qquad \mathbf{M}_i \sim p(\mathbf{M})\,.$$

We keep dropout active at inference and average N forward passes: **Monte Carlo dropout**.

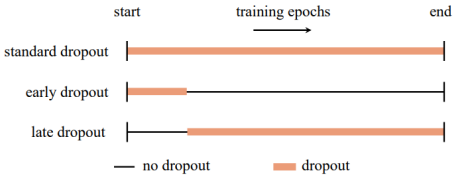
> The spread of the N predictions gives an estimate of the model's **uncertainty**, at the cost of N forward passes.

Gal and Ghahramani (2016), *Dropout as a Bayesian approximation: representing model uncertainty in deep learning*, ICML.

- In a convolutional layer, dropout removes single values of single channels, which neighboring pixels easily replace. **Spatial dropout** drops entire channels; **cutout** drops a patch of the image.
- **DropConnect** drops single weights instead of units: $\widetilde{\mathbf{h}} = \phi\left((\mathbf{W} \odot \mathbf{M})\,\mathbf{x}\right)$.
- **Stochastic depth** drops entire residual branches, $r(\mathbf{x}) = \mathbf{x} + m f(\mathbf{x})$ with a random $m \in \{0, 1\}$. A residual network barely notices a missing block, as we saw.

Plain dropout is less common in CNNs, and more common in transformers and recurrent networks.

Tompson et al. (2015), CVPR; DeVries and Taylor (2017); Wan et al. (2013), ICML; Huang et al. (2016), *Deep networks with stochastic depth*, ECCV.

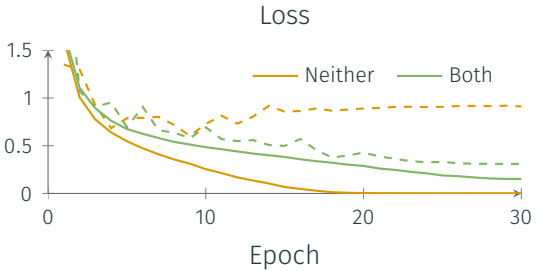


Dropout can be switched on or off during training. **Early dropout** reduces the variance of SGD in the first epochs and helps models that *underfit*. **Late dropout** fights overfitting.

Large LLMs see each example about once, so they rarely overfit: PaLM is pretrained without dropout, and uses 0.1 only for fine-tuning. Vision transformers usually prefer stochastic depth.

Liu et al. (2023), *Dropout reduces underfitting*, ICML; Chowdhery et al. (2023), *PaLM*, JMLR, Sec. 5; Touvron et al. (2021), DeiT, ICML.

Our 20-layer ResNet, with and without augmentation and weight decay:



Solid: training loss; dashed: test loss.

	Test acc.
Neither	82.6%
Weight decay	84.4%
Augmentation	88.2%
Both	89.5%

Without regularization, the training loss goes to zero while the test loss grows. Augmentation matters most; the two together gain 7 points.

Normalization
BN and LN keep the activations in scale, and allow larger learning rates.

Residuals
The identity is the default: the gradient reaches every block.

Regularization
Augmentation, weight decay and dropout make the network generalize.

Together, they train our 56-layer network to 91.4% test accuracy on CIFAR-10. The same three ingredients are in every transformer.

Reading material: Chapter 9 in the book.