Convolutional Neural Networks



Antonio Carta

2026-10-19

University of Pisa

antonio.carta@unipi.it

Goal of the Day

- Shift from static to learned feature extractors based on DNNs
- Discuss CNN design choices

Limitations of Static Feature Extractors

hand-crafted filters (Sobel, Gabor, SIFT, ...): fixed, chosen by a human, identical for every image/domain.

- we can create a large filter bank tuned for different orientations, scales, or patterns
- we don't have a simple way to compose and model hierarchical representations (i.e. more complex and semantic features)
- all our tools up to now are linear: we need non-linearities to model more complex features
  - the main disadvantage is that we will lose interpretability

Deep Neural Networks

Deep Neural Networks

- DNN: a composition of many layers $f_\theta(x) = f_L(f_{L-1}(...f_1(x)))$
  - parameters θ
  - input x (e.g. a (W, H, C) image)
  - layers alternate linear and non-linear operations
- trained via SGD: $\theta_t = \theta_{t-1} - \alpha \nabla_\theta L(f_\theta(x))$
  - L is the loss function (e.g. crossentropy for classification, MSE for regression)
  - α the learning rate

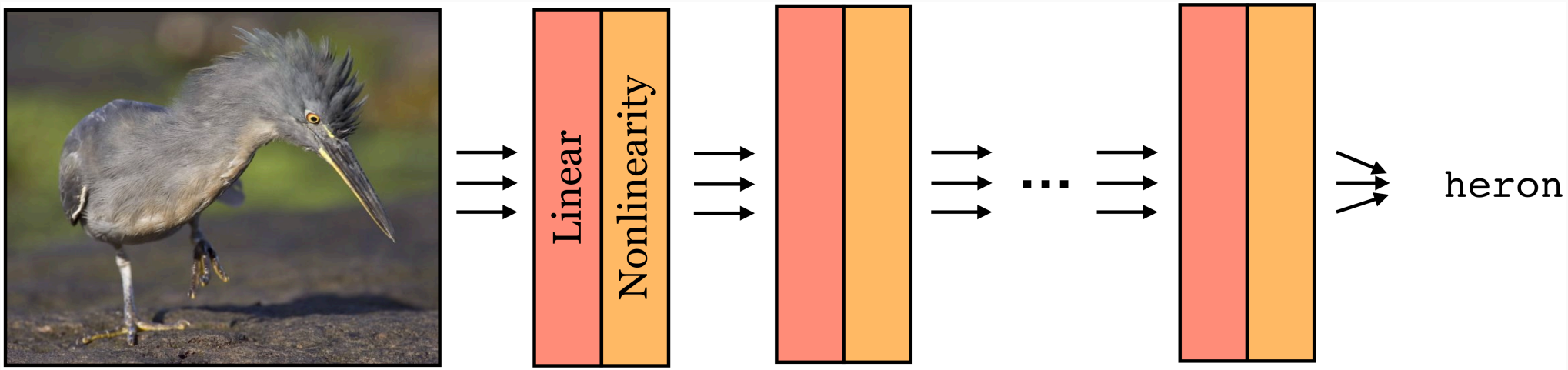


Figure 1: source: visionbook

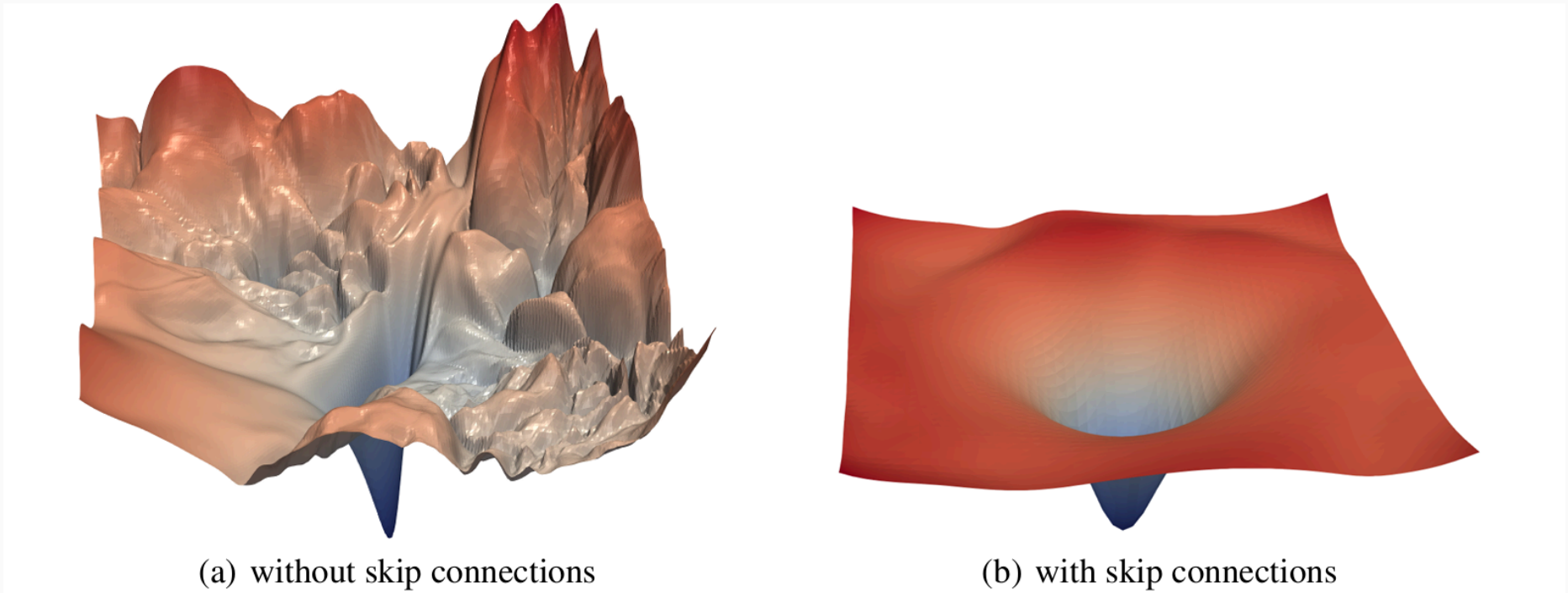
Recap: Simplified DNN training loop

- for each mini-batch
  - apply augmentations to the input
  - forward pass (1): compute the outputs of the DNN
  - forward pass (2): compute the loss
  - backward pass: compute the gradient
  - apply the update with SGD

repeat until convergence / exit condition

Optimization in DNN

- The loss is non-convex due to nonlinearities: we do not have any formal convergence guarantess
  - SGD will find a local minima
  - in practice, in good DNN architectures local minima are good enough



(a) without skip connections

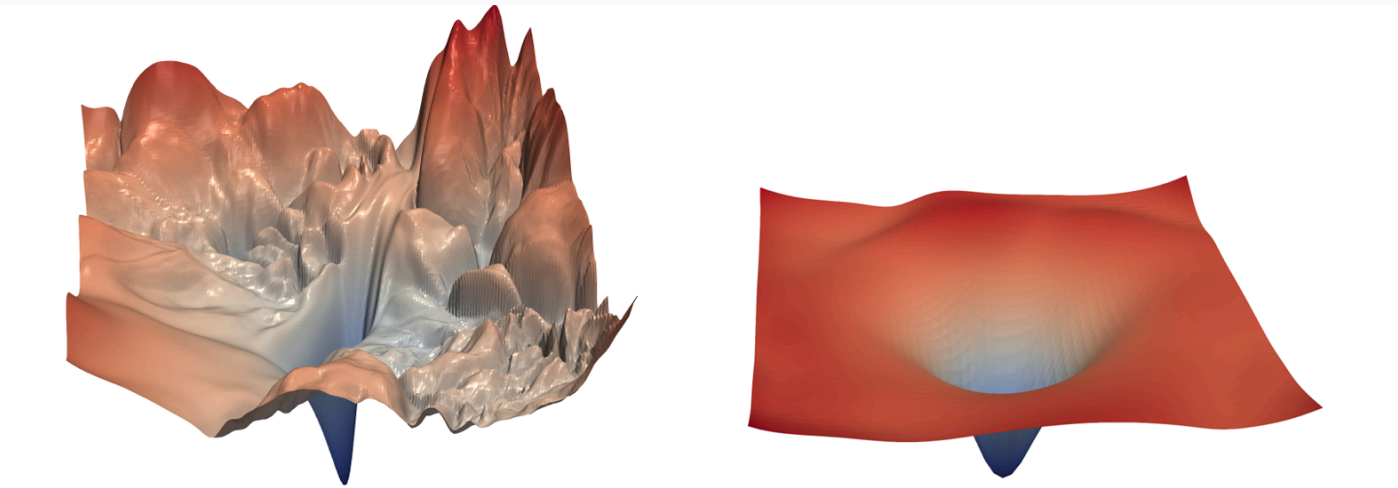
(b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

Figure 2: source: Li, Hao, et al. 'Visualizing the loss landscape of neural nets.' NIPS 2018.

Optimization in DNN (2)

- flatness/sharpness of the curvature (hessian) determine how hard the optimization task is
  - Modern optimizers (e.g. Adam or well-tuned SGD with momentum) and several architectural tricks are designed to improve DNN convergence
    - figure: residual (skip) connections induce a flatter loss landscape
- some hyperparameters (e.g. learning rate) need careful tuning
- general rule: never change default architecture or hyperparameters without understanding and measuring the effect of your change



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

Figure 3: source: Li, Hao, et al. 'Visualizing the loss landscape of neural nets.' NIPS 2018.

Design Choices in DNN Architecture

- **forward pass**: the induced representations / inductive bias of the layer
  - examples: CNNs are LSI operations, a strong prior for images. Dense layer have a very weak inductive prior.
- **backward pass**: avoid optimization issues such as vanishing/exploding gradients
  - examples: proper layer initializations that induce normalized activations; residual connections
- **computational cost**: often based on GPU-like architectures (SIMD-like, massively parallel hardware)
  - memory cost
  - GPU-to-CPU memory bandwidth or GPU-to-GPU in multi-GPU scenarios
  - computations amenable to parallel execution
  - intuitively: the goal is to keep the GPU at 100% usage during training and inference without any memory/CPU/bandwidth bottlenecks
  - in embedded vision systems we may also have realtime and latency constraints

DNN Input Transformations

For a non-linear learning problem, we want to find a representation that maps inputs to a linearly separable representation.

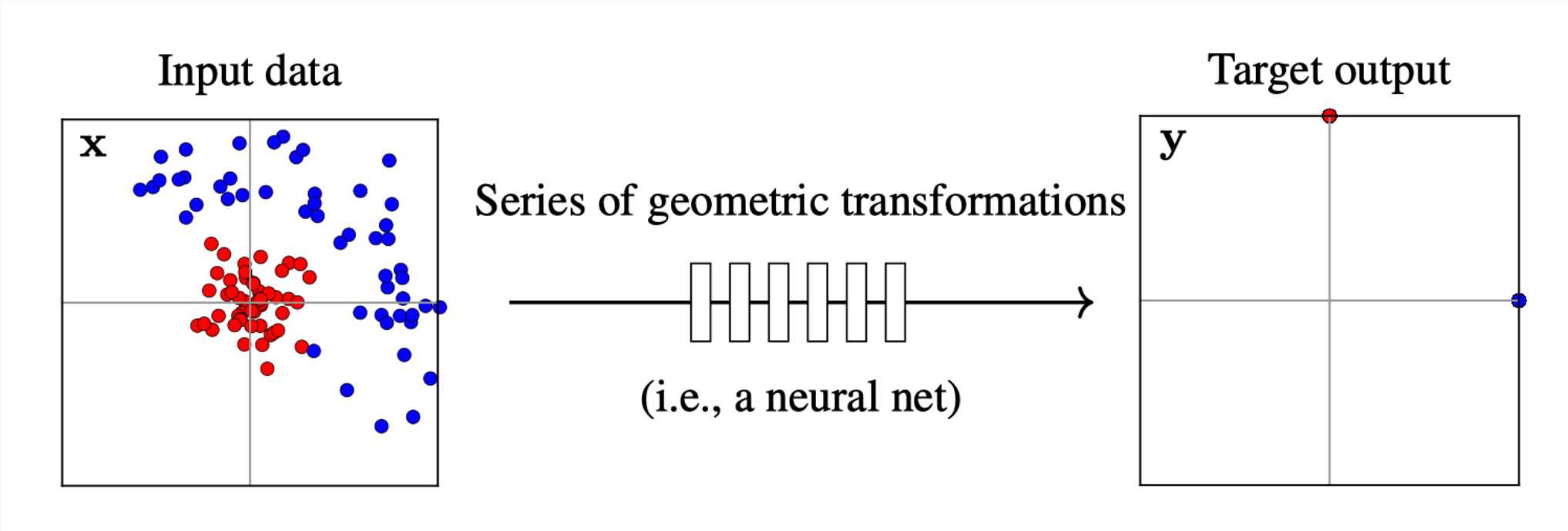


Figure 4: source: visionbook

DNN Input Transformations (2)

- The DNN gradually learns to make the data linearly separable during training
- linear separability is best at the final layer

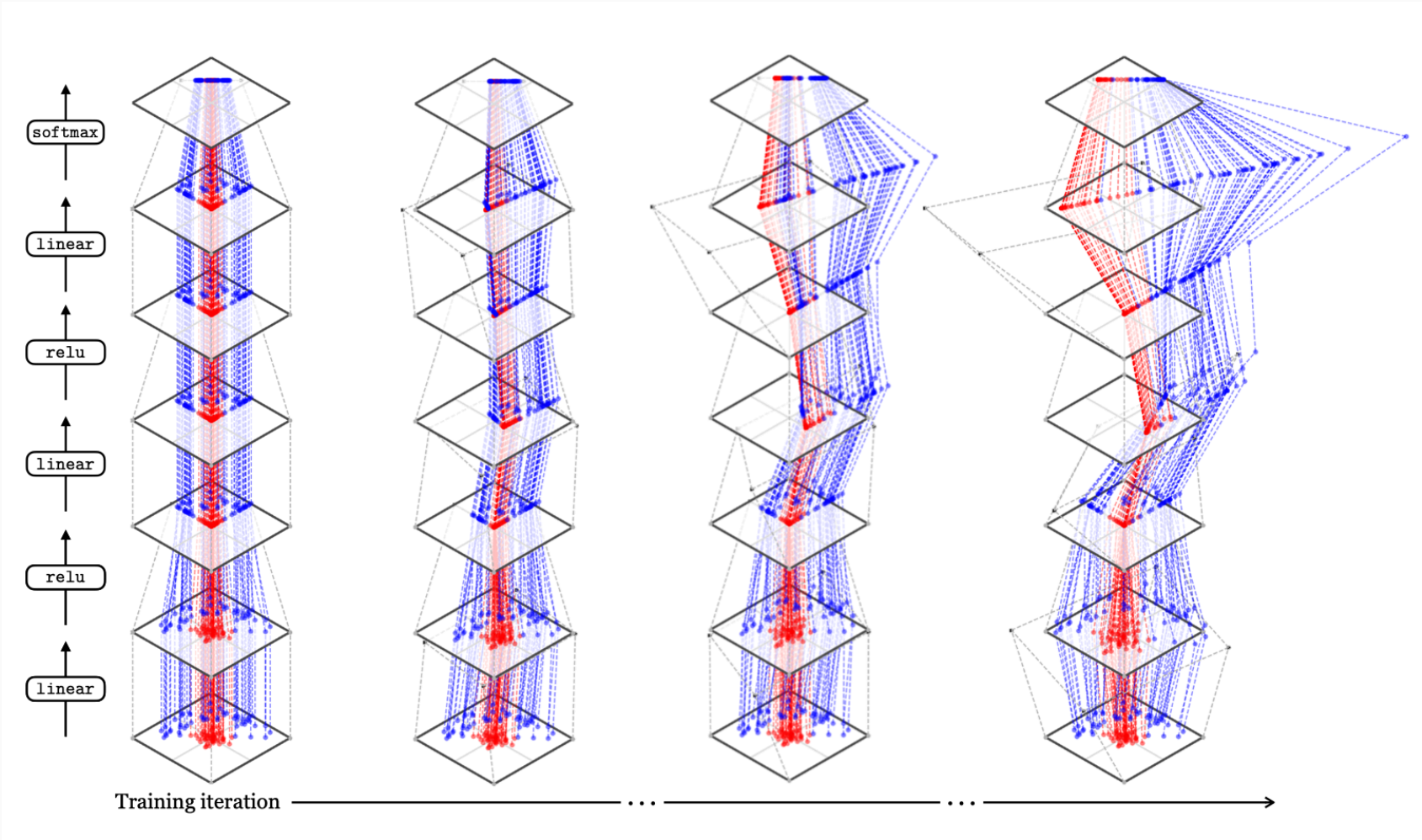


Figure 5: source: visionbook

Embeddings

- **embedding**: a high-dimensional vector that is the representation of the input as the output at some DNN layer
- We will often freeze a model and use its embeddings to solve a new ML problem.
- **visualization**: embeddings are plotted via dimensionality reduction techniques (e.g. PCA, t-SNE)

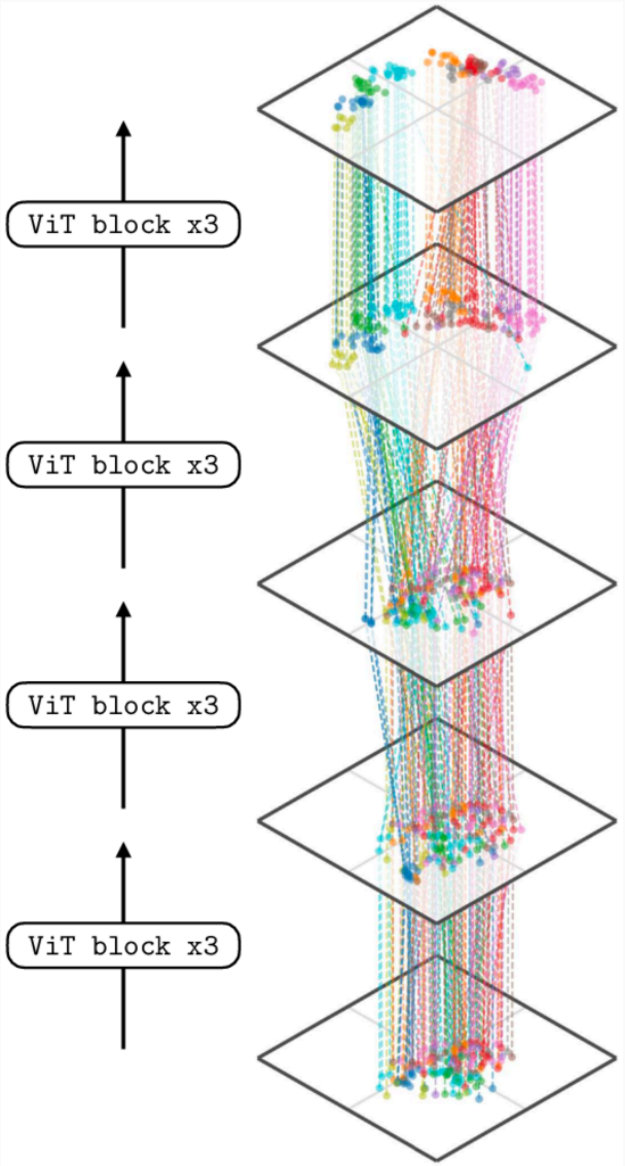


Figure 6: source: visionbook

Embeddings and Generalization

- the goal of DNN training is to achieve a **disentangled** representation: a representation where each concept (e.g. class) is well separated from the others
  - a good disentangled representation should generalize to extreme shifts in views, day/night, ...
- with pretraining we aim to achieve disentanglement for novel concepts (e.g. **unseen classes**)

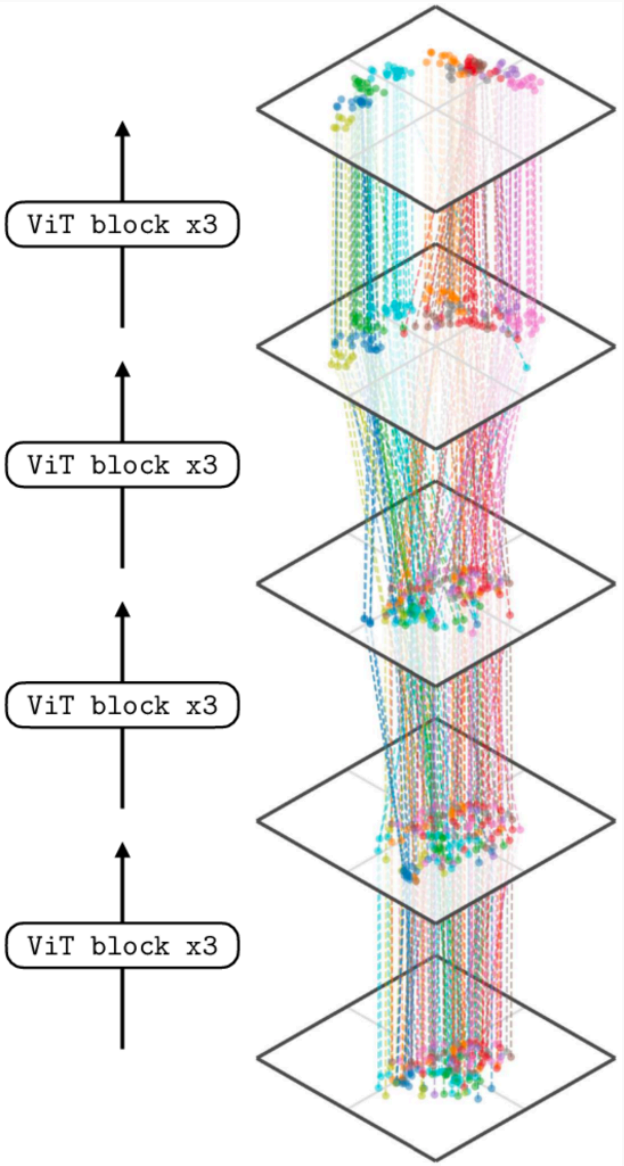


Figure 7: source: visionbook

Depth vs Width

Depth: allows to represents more complex Functions

- example figure: ReLU splits the input space in different regions. The number of regions is exponential in the depth.

Width: dimensionality of the embeddings

- higher dimensionality results in sparser embeddings
- example: random embeddings are roughly orthogonal ($v^T u = 0$) in high-dimensional spaces
- notice how standard ML theory (e.g. curse of dimensionality) would discourage you from trying high-dimensional spaces

Modern SotA architectures need to be both wide and deep.

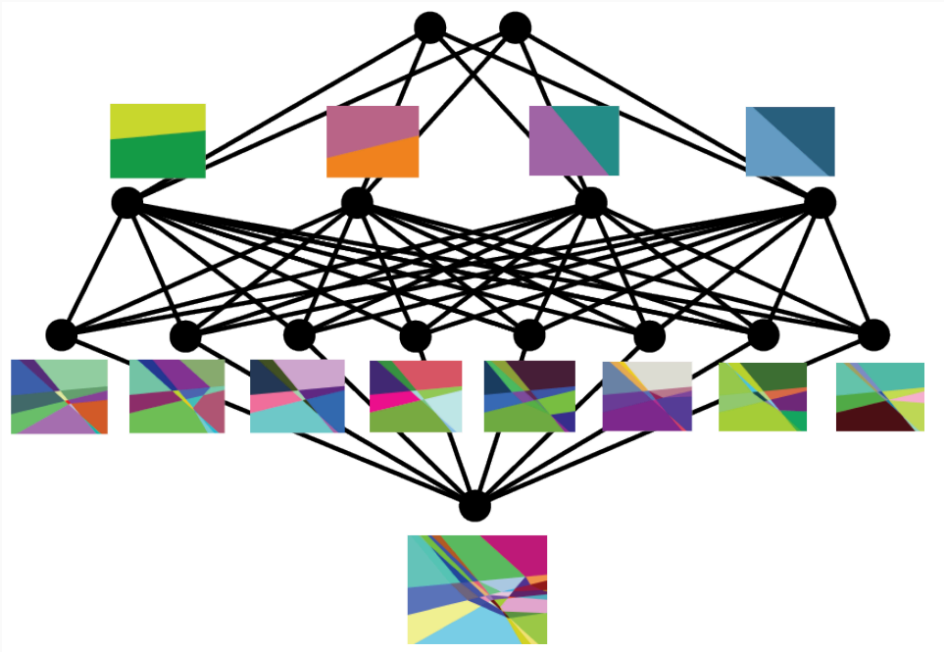


Figure 8: Intuition on how ReLU splits the input space. source: B. Hanin, and D. Rolnick. 'Complexity of linear regions in deep networks.' ICML 2019.

Regularization in DL

in classic ML:

- curse of dimensionality: high-dimensional problems are harder to solve
- Bias-Variance tradeoff: a good model finds a balance between overfitting and underfitting

in DNNs:

- (infinite-width) DNNs are universal approximators
- Theoretical complexity is basically infinite for any standard measures from theory (e.g. VC-dimensionality)
- In practice, a DNN can easily fit a dataset with random labels
- ref: Zhang, Chiyuan, et al. “Understanding deep learning requires rethinking generalization.” ICLR 2017.

Regularization in DL

in DNNs:

- the stronger regularization effect is implicit in the SGD parameters (learning rate, batch size, epochs) and architecture
- explicit regularization via weight decay and dropout is also used

rule of thumbs about model size:

- Use a model as big as you can afford. Computational limitations are used to select the model size, not overfitting issues.
- *scaling laws*: for foundation models we know that model size is related to dataset size to achieve cost-optimal training.

Basic DNN Components

Activation Functions

ReLU or leaky-ReLU are the standard choice for large DNNs:

- no vanishing gradient issues
- fast implementation

$$x_{\text{out}[i]} = \begin{cases} 1 \text{ if } x_{\text{in}[i]} > 0 \\ 0 \text{ otherwise} \end{cases} \quad \triangleleft \text{ threshold}$$

$$x_{\text{out}[i]} = \frac{1}{1 + e^{-x_{\text{in}[i]}}} \quad \triangleleft \text{ sigmoid}$$

$$x_{\text{out}[i]} = 2 * \text{sigmoid}\left(2 * x_{\text{in}[i]}\right) - 1 \quad \triangleleft \text{ tanh}$$

$$x_{\text{out}[i]} = \max\left(x_{\text{in}[i]}, 0\right) \quad \triangleleft \text{ relu}$$

$$x_{\text{out}[i]} = \begin{cases} x_{\text{in}[i]} & \text{if } x_{\text{in}[i]} > 0 \\ \alpha x_{\text{in}[i]} & \text{otherwise} \end{cases} \quad \triangleleft \text{ leaky-relu}$$

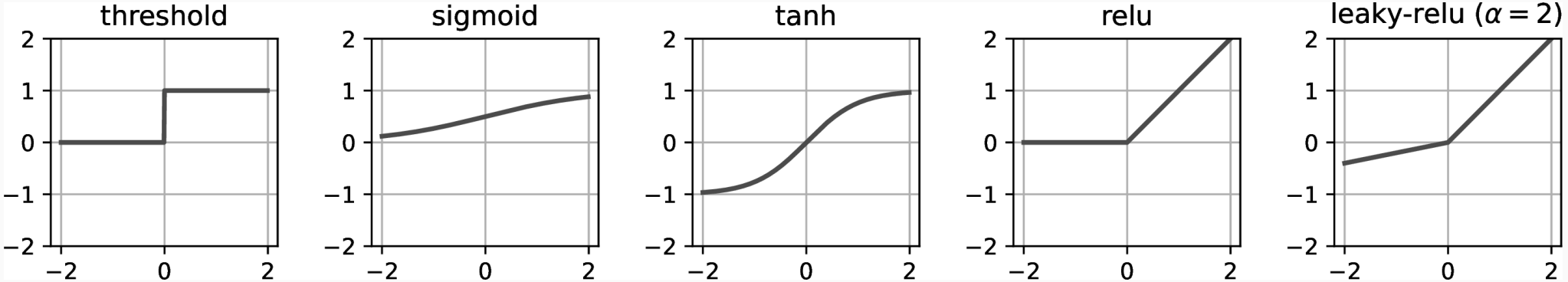


Figure 9: source: visionbook

Activation Functions (2)

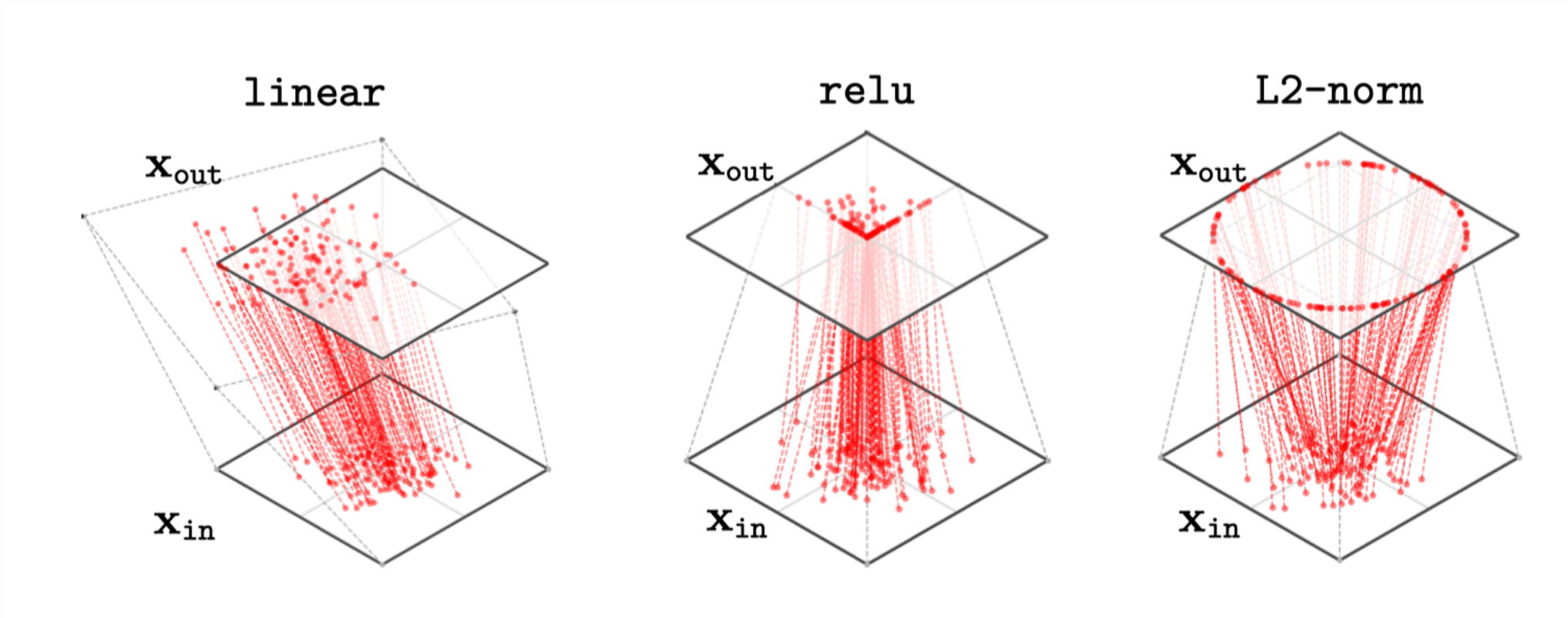


Figure 10: source: visionbook

Batch Normalization

Batch Normalization standardizes its inputs:

$$x_{\text{out}[i]} = \gamma \frac{x_{\text{in}[i]} - \mathbb{E}\left[x_{\text{in}[i]}\right]}{\sqrt{\text{Var}\left[x_{\text{in}[i]}\right]}} + \beta$$

- γ and β are learned scalar parameters

the mean $\mathbb{E}\left[x_{\text{in}[i]}\right]$ and variance $\text{Var}\left[x_{\text{in}[i]}\right]$

- at training time: estimate on the current mini-batch
- at inference time: uses running average statistics computed during training
- this allows inference to be independent from mini-batch choices

Note: BatchNorm introduces a dependence on batch size which can cause issues sometimes (e.g. batch-size=1, multi-task learning, ...). For these reasons, many modern DNNs use LayerNorm.

Layer Normalization

- instance-wise average: $\mu = \frac{1}{n}\sum_{i=1}^{n} x_{\text{in}[i]}$
- instance-wise variance: $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_{\text{in}[i]} - \mu\right)^2$

Layer Normalization:

$$x_{\text{out}[i]} = \gamma\frac{x_{\text{in}[i]} - \mu}{\sigma} + \beta$$

- each instance is standardized independently
- for 1D representations (2D data matrix $N_{\text{batch}} \times N_{\text{features}}$) LayerNorm is the "transpose" of BatchNorm

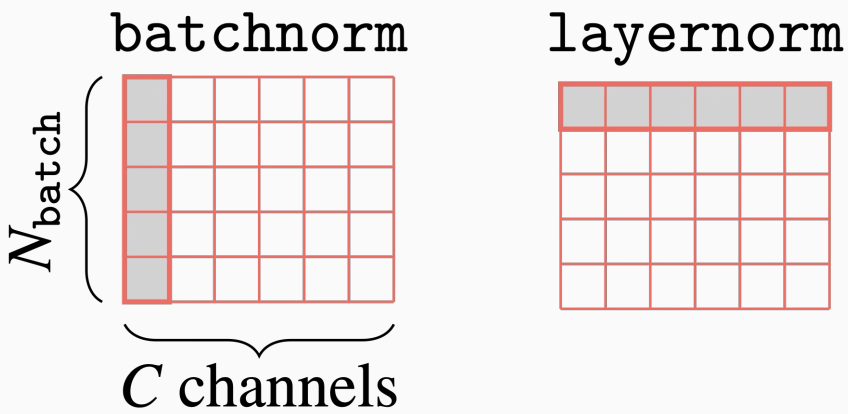


Figure 11: BatchNorm vs LayerNorm. source: visionbook

Temperature-Scaled Softmax

The output layer for classification problems is the **Temperature-Scaled Softmax**:

$$x_{\text{out}[i]} = \frac{e^{\frac{x_{\text{in}[i]}}{\tau}}}{\sum_{k=1}^{K} e^{\frac{x_{\text{in}[k]}}{\tau}}}$$

- a soft approximation of argmax to select the output with highest response (e.g. the predicted class)
- its output x_{out} are used as the output probabilities in classification problems
  - its inputs x_{in} are the logits
  - at inference time, if you only care about the class with max probability you don't need to compute the softmax: the unit with max logit is also the unit with the max probability
- τ is the temperature:
  - $\tau \to +\text{inf}$: uniform logits =0 $\to$ uniform probabilities
  - $\tau \to 0$: peaked probability probability distribution with $x_{\text{out}}[i_{\text{best}}] = 1$

Output for Regression problems

- for regression problems we just have the identity activation function (i.e. a linear layer)
- if we want a bounded output we can use the sigmoid (bounded in $(0, 1)$) and rescale it appropriately

Example in $(0, 255)$ (e.g. image generation):

$$x_{\text{out}[i]} = 255 * \text{sigmoid}\Big(x_{\text{in}[i]}\Big)$$

CrossEntropy loss

in classification problems we use the crossentropy loss:

$$l = -\sum_{c=1}^{C} y[c] \log \hat{y}[c]$$

- y true probabilities
- $\hat{y}$ predicted probabilities

For numerical stability reasons, most deep learning frameworks compute the crossentropy from the logits to combine softmax and crossentropy in a single operation:

$$l = -\sum_{c=1}^{C} y_c \log \left(\frac{\exp(x_{\text{out}}[c])}{\sum_{i=1}^{C} \exp(x_{\text{out}}[i])} \right)$$

Convolutions in DNNs

Convolution Recap

$$x_{\text{out}[n,m]} = b + \sum_{k_1,k_2=-K}^{K} w[k_1, k_2] x_{\text{in}[n+k_1,m+k_2]}$$

- w is the **kernel** (filter) $((2K+1) \times (2K+1))$
- x is a 2D input $(\text{width} \times \text{height})$
- CNNs implement **cross-correlation** (no kernel flip)
- we also added a learnable scalar bias b

Parameter Sharing

- a convolution is just a special linear operation
- the resulting weight matrix has several zeros and several equal parameters
- with convolutions we save compute and memory compared to the equivalent matrix multiplication

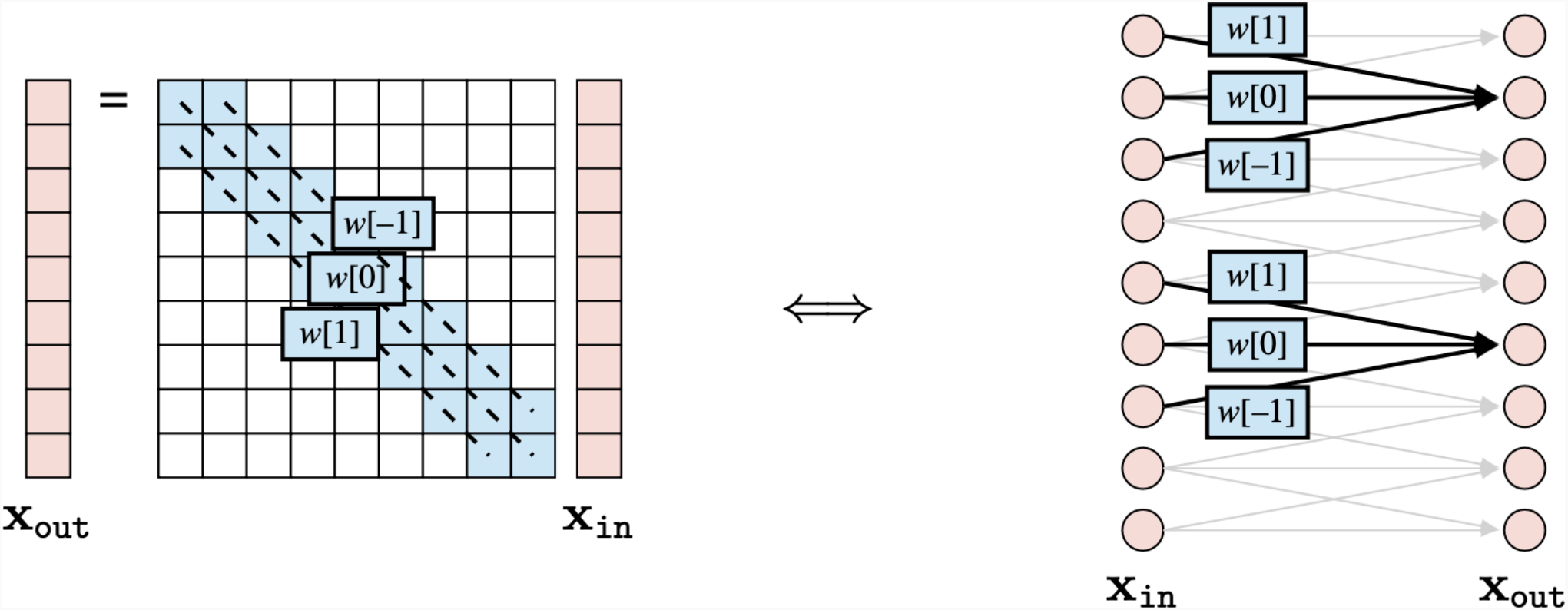


Figure 12: source: visionbook

Multi-Channel Inputs

We only defined convolution with 1 input and 1 output channels. DNN use multiple input and output channels.

Multi-Channel Inputs:

$$\boldsymbol{x}_{\text{out}} = \sum_{c} \boldsymbol{w}[c, :, :] \star \boldsymbol{x}_{\text{in}[c,:,:]} + b[c]$$

where $*$ is the single-channel conv defined earlier.

Multi-Channel Ouptut:

$$\begin{aligned} \boldsymbol{x}_{\text{out}[0,:,:]} &= \boldsymbol{w}[0,:,:] \star \boldsymbol{x}_{\text{in}} + b[0] \\ &\vdots \\ \boldsymbol{x}_{\text{out}[C-1,:,:]} &= \boldsymbol{w}[C-1,:,:] \star \boldsymbol{x}_{\text{in}} + b[C-1] \end{aligned}$$

- equivalent to a filter bank. Each output channel correspond to a different conv filter
- an output channel is also called a **feature map**

$$x_{\text{out}[c_2,:,:]} = \sum_{c_1=1}^{C_{\text{in}}} w[c_1, c_2, :, :] \star x_{\text{in}[c_1,:,:]} + b[c_2]$$

- total kernel tensor shape: $C_{\text{out}} \times C_{\text{in}} \times K \times K$

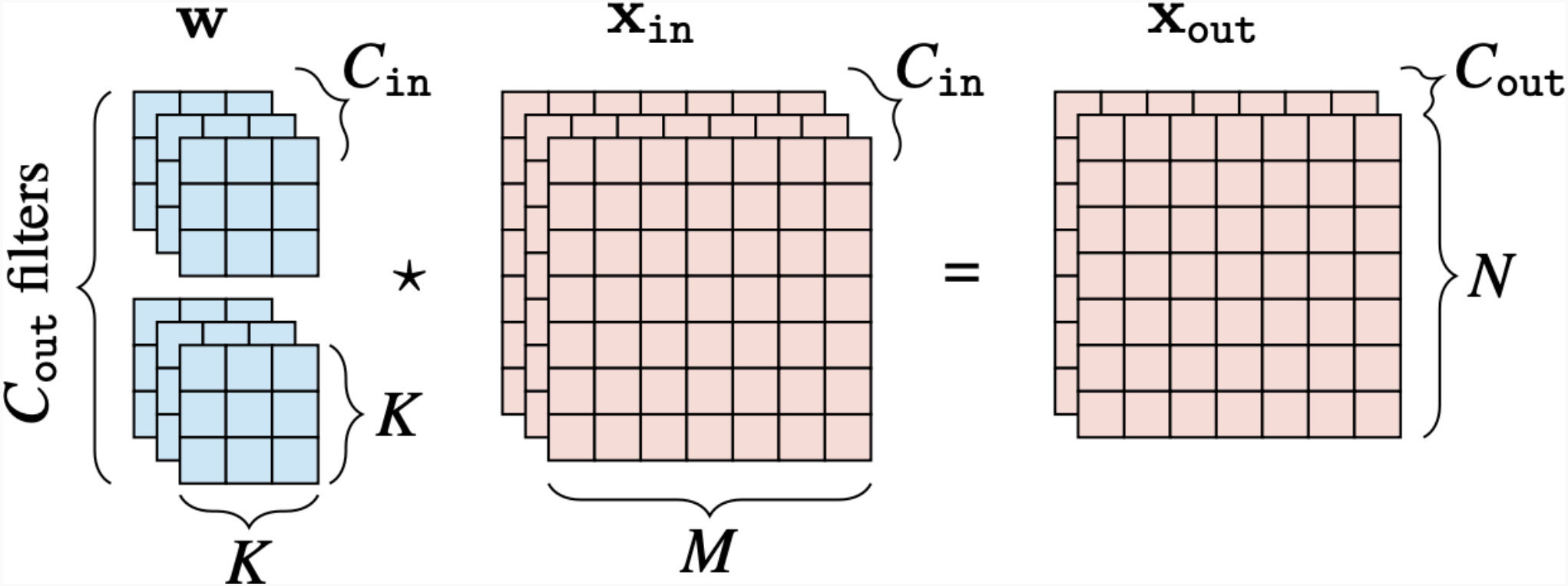


Figure 13: source: visionbook

Example: Parameter Efficiency

Example: $H \times W \times C_{\text{in}}$ input to an $H \times W \times C_{\text{out}}$ output:

Fully-connected layer:

- every output unit connects to every input unit
- parameters: $(HWC_{\text{in}}) \times (HWC_{\text{out}})$
- for a $64 \times 64 \times 3$ image, $C_{\text{out}} = 64$ we have $\approx 3B$ weights

Conv (kernel size $k \times k$):

- parameters: $k^2 C_{\text{in}} C_{\text{out}}$
  - independent of image resolution (H, W)
- for $k = 3$ we have 1728 weights

Stride

$$x_{\text{out}[n,m]} = b + \sum_{k_1,k_2=-K}^{K} w[k_1, k_2] x_{\text{in}[s_n n - k_1, s_m m - k_2]}$$

- stride s_n, s_m. Usually a single parameter $s = s_n = s_m$
- greatly reduces the computational cost

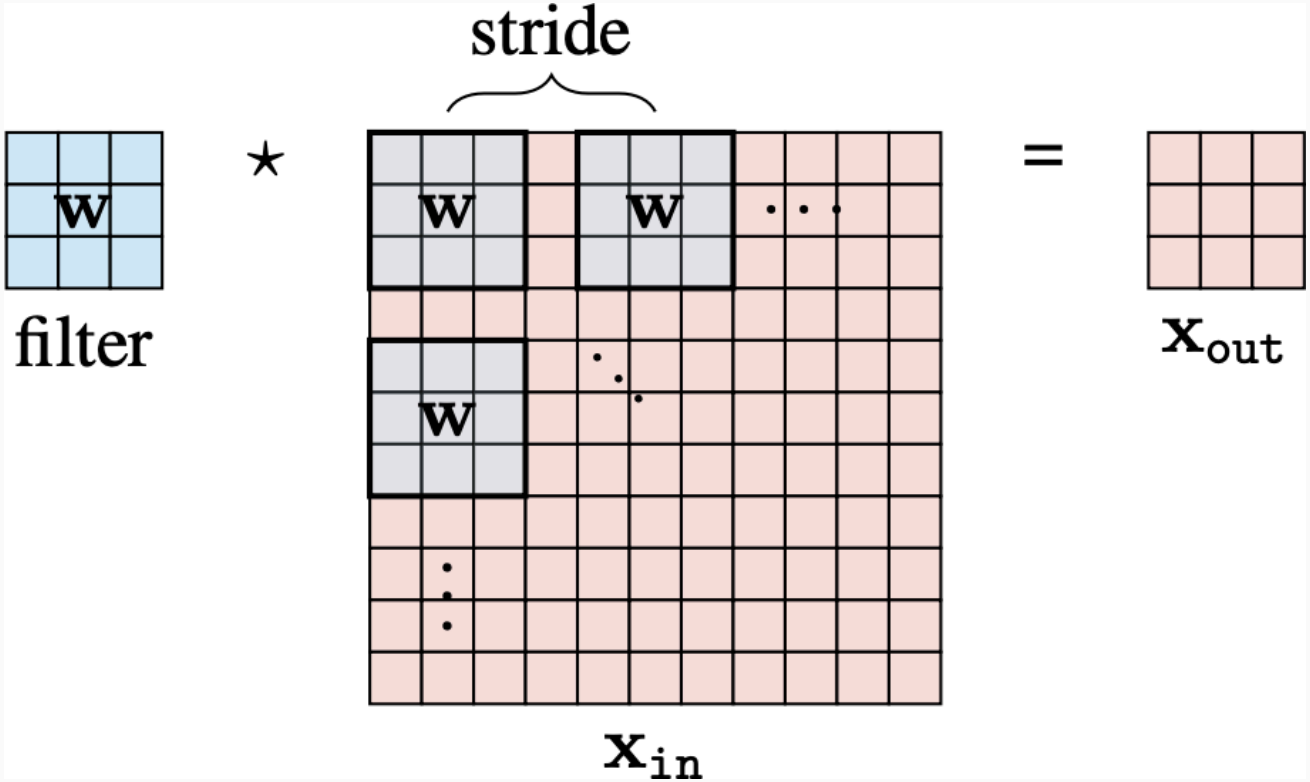


Figure 14: source: visionbook

Stride Aliasing

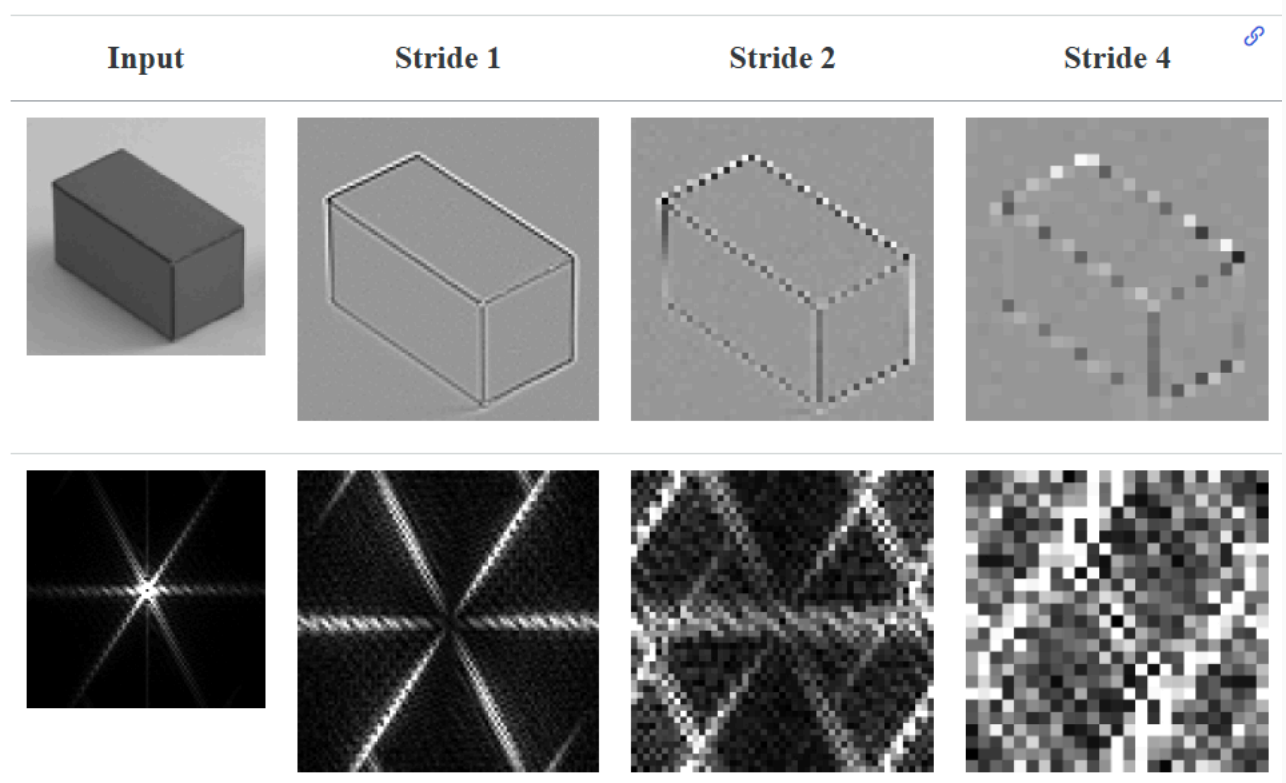


Figure 15: source: visionbook

Dilated Convolution

$$x_{\text{out}[n,m]} = b + \sum_{k_1,k_2=-K}^{K} w[k_1, k_2] x_{\text{in}[n-d_k k_1, m-d_k k_2]}$$

- d_k dilation parameter

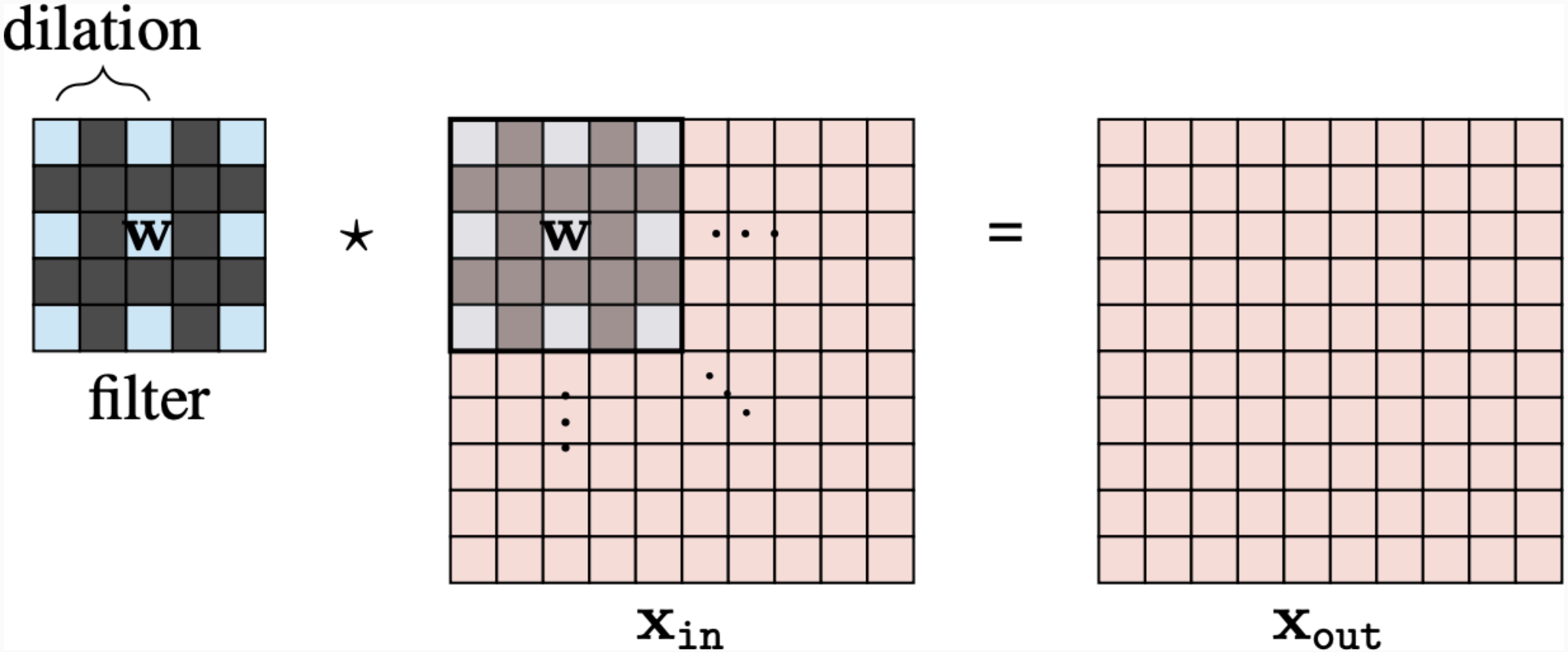


Figure 16: source: visionbook

Pooling

Pooling Layers

- **Max pooling**: replace each $k \times k$ neighborhood with its **maximum** value
- **Average pooling**: replace each $k \times k$ neighborhood with its **mean** value

$$x_{\text{out}[i]} = \max_{i \in \mathcal{N}(i)} x_{\text{in}[i]}$$

$$x_{\text{out}[i]} = \frac{1}{|\mathcal{N}|} \sum_{i \in \mathcal{N}(i)} x_{\text{in}[i]}$$

- reduces spatial resolution (less compute)
- introduces a small amount of local translation robustness, small shifts within a pooling window do not change the max
- Max pooling preserves the strongest activation ("is this pattern present somewhere nearby"), average pooling smooths/blends nearby responses

Global Average Pooling

- **Global average pooling (GAP)**: average a feature map over its entire spatial dimensions, producing one number per channel, regardless of input image size
- Used at the end of the convolutional part of a classification model, right before the linear layers, to turn a $H \times W \times C$ feature map into a C-dimensional vector
- translation **invariance**:
  - convolutions are **equivariant** w.r.t. translation, not invariant
  - the average over all positions induces some **invariance**
- Also removes the constraint of a fixed input resolution: different input resolutions result in different resolutions for the feature map, but the number of channels doesn't change

Hierarchical Feature Learning

Receptive Field

- The **receptive field** of a unit is the region of the **original input** that can affect that unit's value
- A unit in a shallow layer is affected by the inputs "covered" by the kernel. A unit in a deep layer can be influenced by a much larger region, even the whole image
- Receptive field size determines what scale of structure a unit can respond to: small RF, local texture/edges, large RF, whole objects or global context
- receptive field grows with depth and kernel size

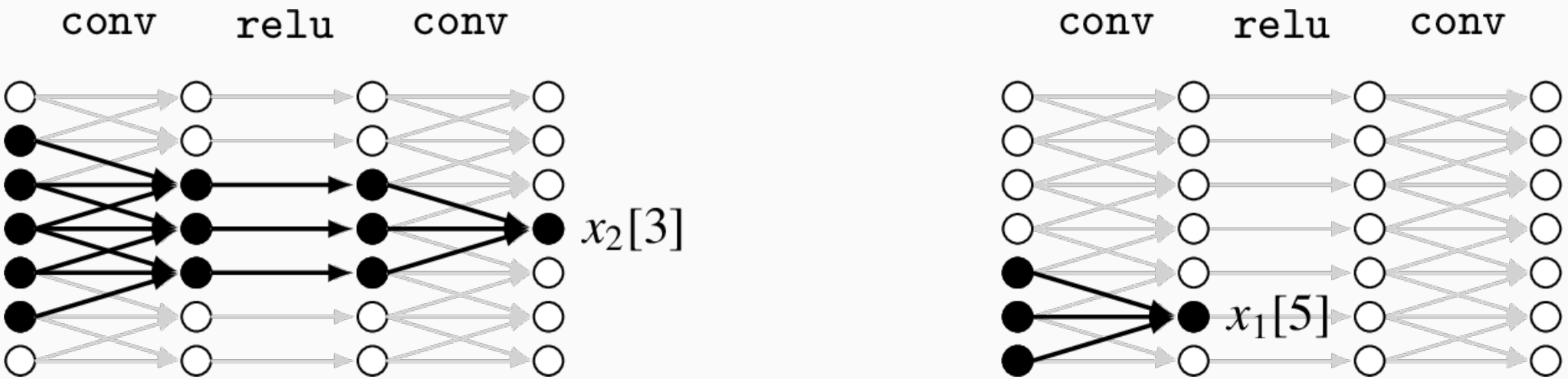


Figure 17: source: visionbook

The Hierarchy Hypothesis

- **Hypothesis**: a CNN trained on a complex visual task learns a **hierarchy** of representations: edges → textures → object parts → whole objects/semantic concepts, from early to late layers
- Motivated by
  - receptive field: early layers have small RF, can only represent local structure; late layers have large RF, can represent whole-object structure
  - empirical visualization of feature maps of trained networks

Feature Maps

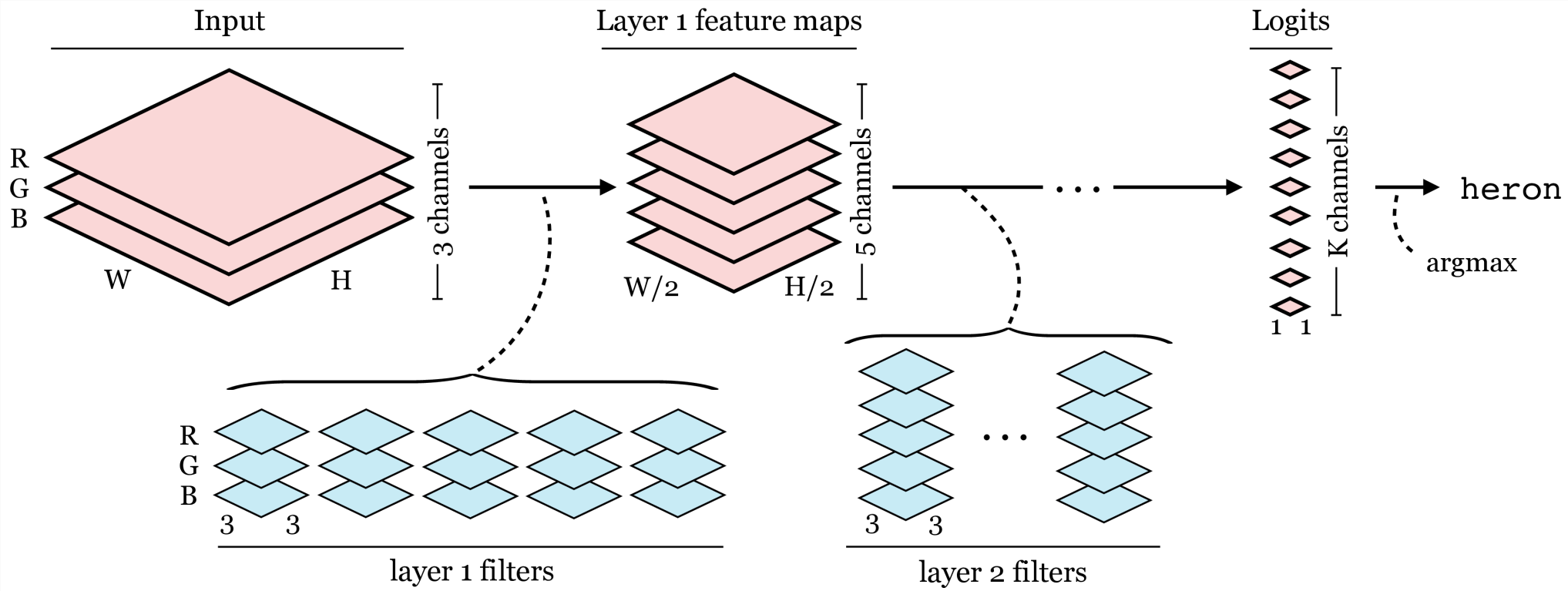


Figure 18: source: visionbook

Example: Feature Maps

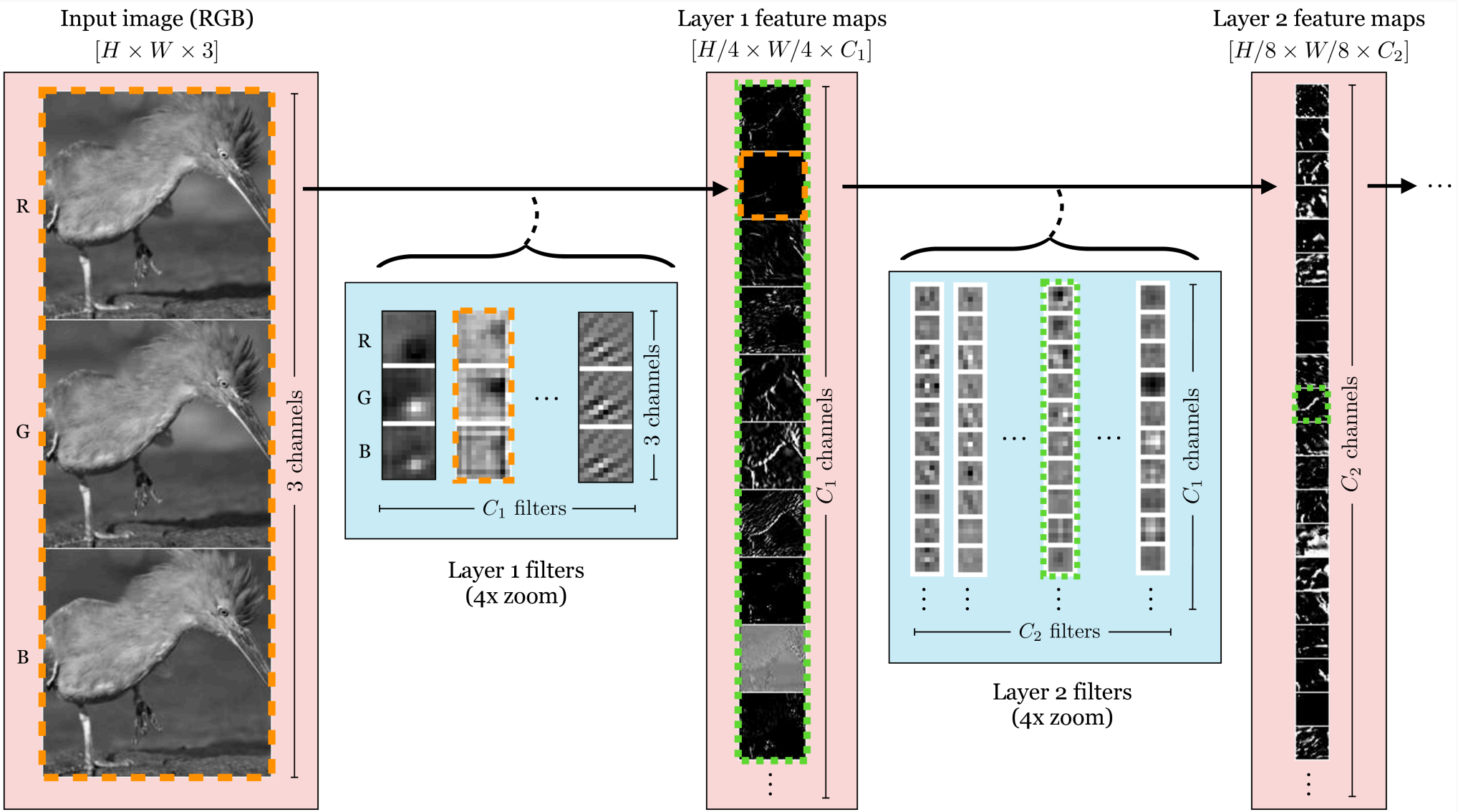


Figure 19: source: visionbook

PCA of Feature Maps visualization

- Since we have a high number of channels in deep layers, it is hard to inspect all of them
- popular alternative: PCA
  - do the PCA on the feature maps of an image and select top-3 Components
  - now you have an image with 3 channels

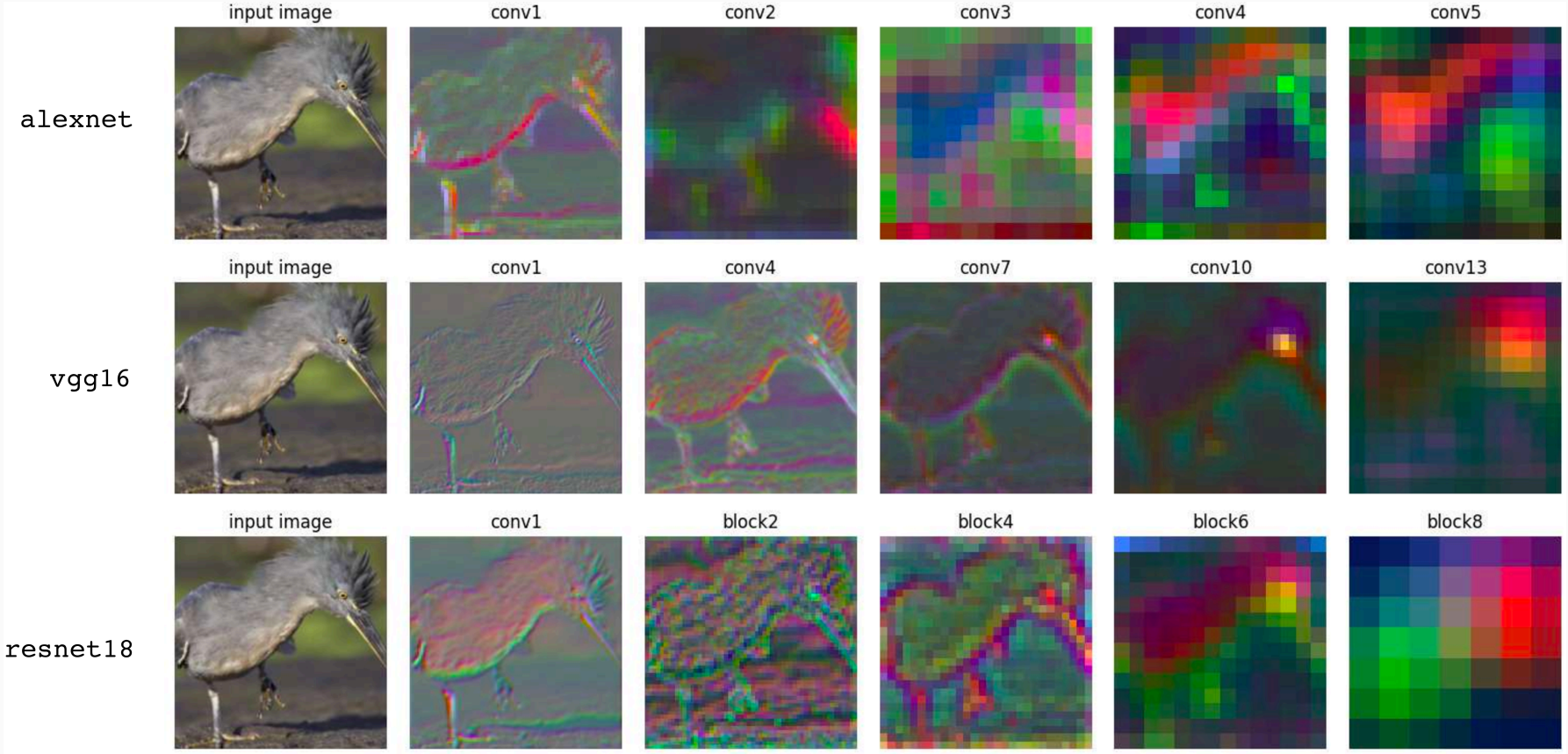


Figure 20: source: visionbook

Classic Architectures

A Basic CNN Architecture

applied in sequence:

- **conv** blocks: sequence of conv, nonlinearity, pooling
  - number of filters increases w.r.t. depth while image size decreases
- **dense** blocks: dense layers and nonlinearity
- **output**: softmax and loss

This is a classic standard template (LeNet, 1998, trained on MNIST)

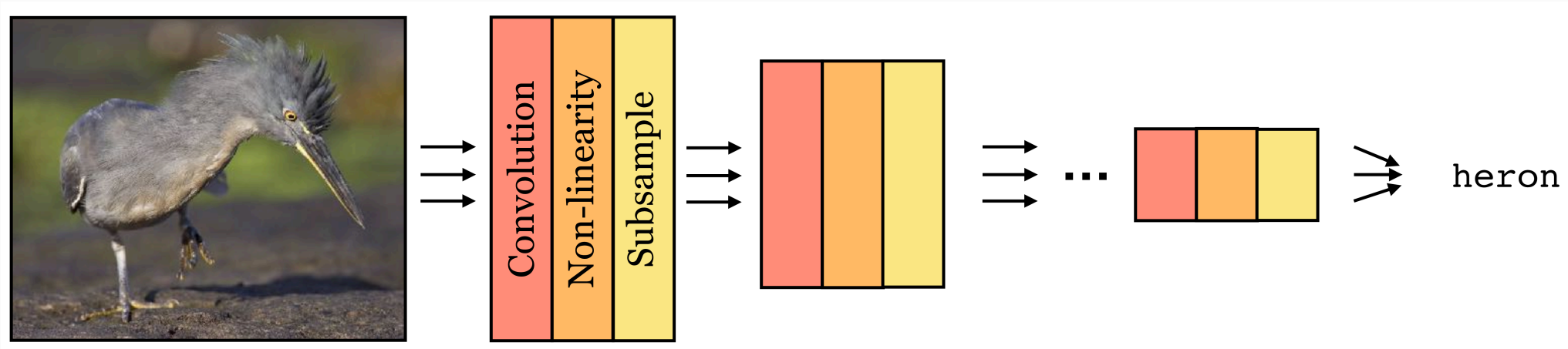


Figure 21: source: visionbook

ImageNet 1k

- progress in deep learning depends on high-quality data as much as technical advances
- ImageNet 1k: image classification on 1000 classes, 1.2M images

https://github.com/EliSchwartz/imagenet-sample-images/blob/master/gallery.md



Figure 22: source: ImageNet website

AlexNet (2012)

- First SotA model on image classification.
  - Trained on ImageNet (1000 classes, 1.2M images)
  - AlexNet is the start of the deep learning revolution in vision
- Key ingredients:
  - ReLU: cheap, no vanishing gradient
  - training on two consumer GPUs
  - dropout for regularization
- 8 weight layers, wide with many channels

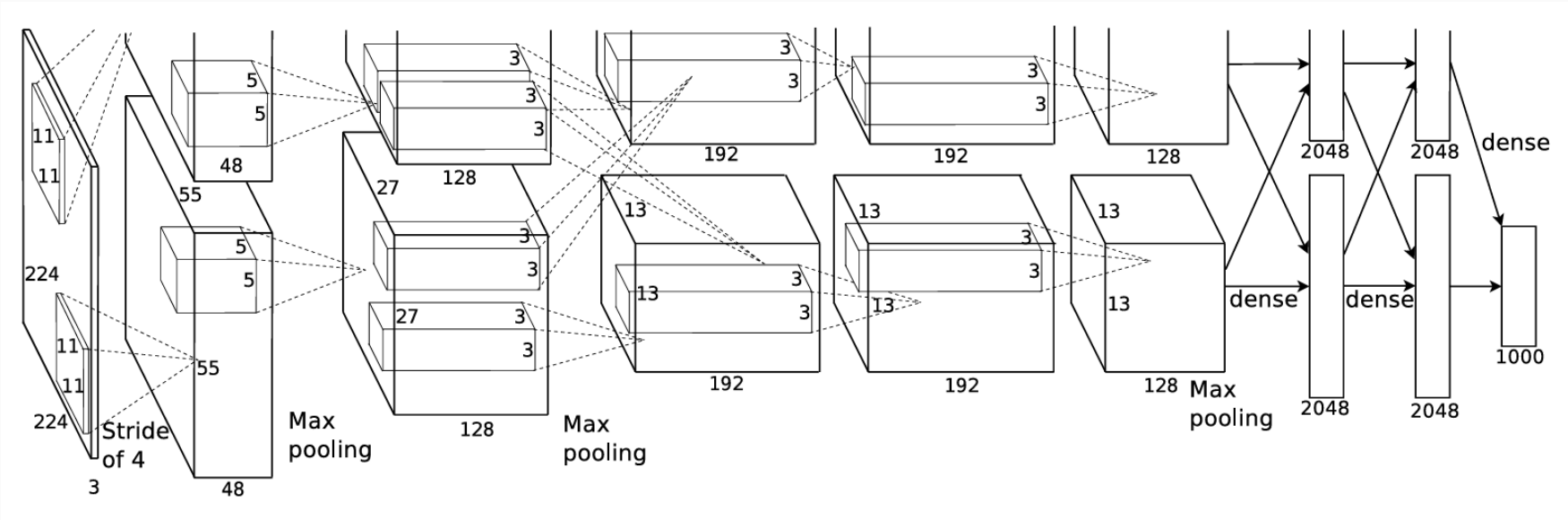


Figure 23: source: AlexNet paper

VGG (2014)

- Simple and uniform design: Use only 3×3 conv layers (stride 1) and 2×2 max pooling
  - trained up to 19 weight layers
- receptive field argument: stacking small kernels reaches large receptive fields with fewer parameters and more nonlinearities than a few large-kernel layers
- Showed that increasing depth improves accuracy, while the exact kernel size is less important

Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." ICLR 2015.

GoogLeNet / Inception (2014)

- **Inception module**: apply several kernel sizes (1×1, 3×3, 5×5) in parallel
- Motivation: the "correct" scale of structure varies by location and by image
- 1×1 conv used as a cheap channel-mixing/dimensionality-reduction step before the larger 3×3 and 5×5 conv

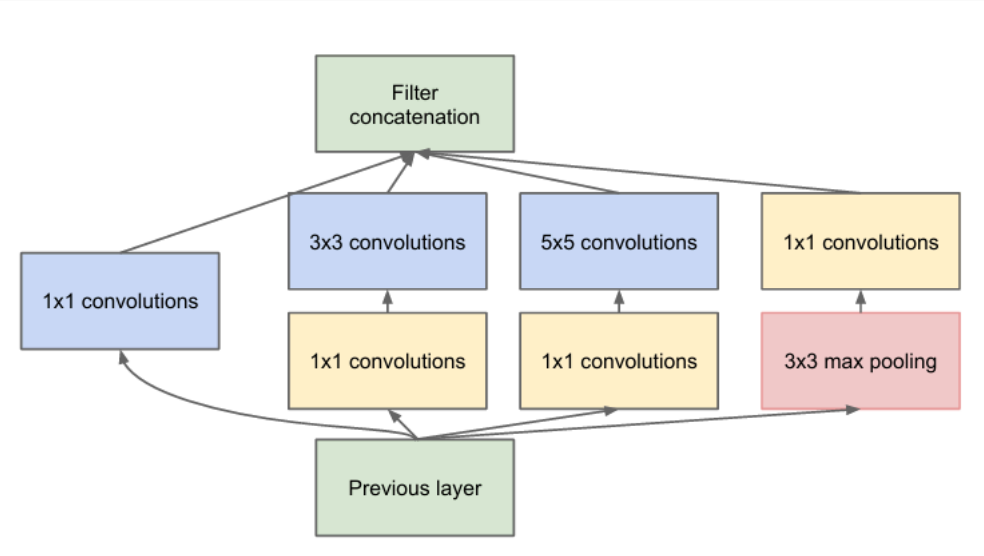


Figure 24: source: Inception module paper

ResNet: Residual Connections (2015)

- **vanishing/exploding gradients**: in deep networks, gradients backpropagated through many stacked nonlinear layers tend to shrink (or, less commonly, explode) exponentially with depth
  - this limits the accuracy benefits of increasing depth
  - it's an optimization issue that we can solve by better architectural design, to enable deeper architectures
- **Residual block**: $y = F(x) + x$ (**skip connection** adds the input)
  - the identity mapping is trivial to represent ($F(x) = 0$), so layers can be (close to) the identity if they are unnecessary
  - Gradient flow: by the chain rule, $\partial y/\partial x = \partial F/\partial x + I$. The identity guarantees a direct gradient path around each block, independent of how small $\partial F/\partial x$ becomes
  - in practice: with residual connections, adding more layers properly initialized with small weights should never hurt the model's accuracy
- Enabled training networks with over 100 layers (ResNet-152)

DenseNet (2017)

- **Feature reuse**: each layer receives as input the concatenation of the feature maps of all preceding layers in the block
- **Gradient flow**: direct connection to every earlier layer's output, shortening gradient paths
- Tradeoff: more memory due to concatenation

Huang, Gao, et al. "Densely connected convolutional networks." CVPR 2017.

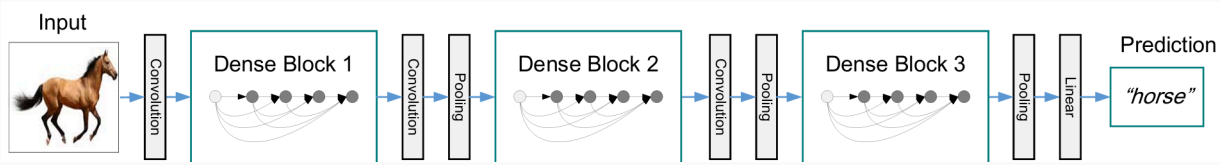


Figure 25: source: DenseNet paper

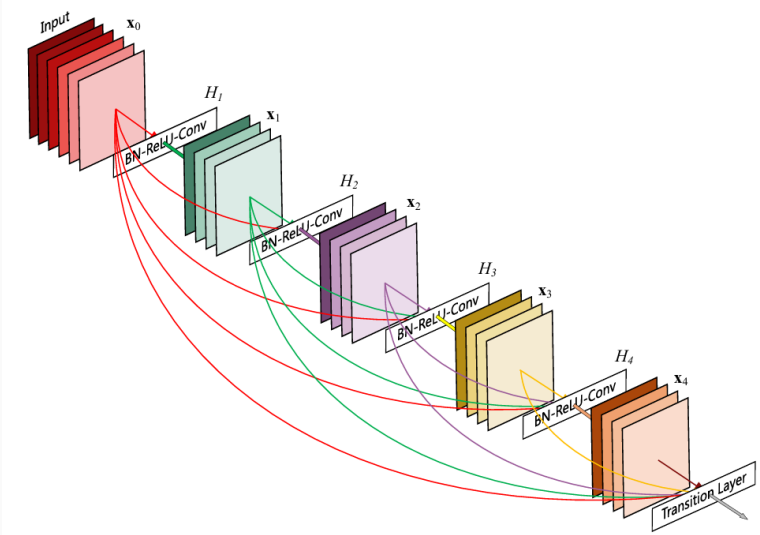


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

Figure 26: source: DenseNet paper

EfficientNet:

- Scaling a CNN up (more accuracy, more compute) can independently increase depth, width (channels), or input resolution
- Finding: scaling **all three jointly**, by a fixed compound ratio, gives better accuracy-per-FLOP than scaling any single dimension alone

MobileNet, efficiency for deployment:

- **depthwise separable convolutions**: a $k \times k$ depthwise conv (one filter per channel, no channel mixing) followed by a 1×1 pointwise conv (channel mixing)
- Reduces compute from $O(k^2 C_{\text{in}} C_{\text{out}})$ to $O(k^2 C_{\text{in}} + C_{\text{in}} C_{\text{out}})$, enabling real-time inference on mobile/edge hardware

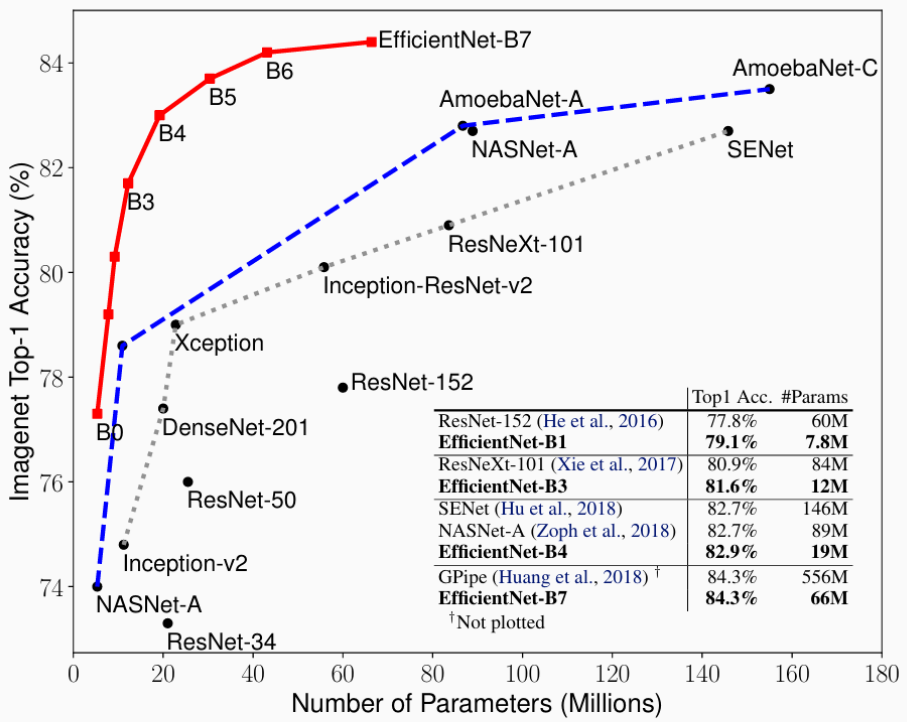


Figure 1. **Model Size vs. ImageNet Accuracy.** All numbers are for single-crop, single-model. Our EfficientNets significantly outperform other ConvNets. In particular, EfficientNet-B7 achieves new state-of-the-art 84.3% top-1 accuracy but being 8.4x smaller and 6.1x faster than GPipe. EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152. Details are in Table 2 and 4.

Figure 27: source: EfficientNet paper

ConvNeXt (2022)

- ConvNext borrows successful Transformer-era design choices into CNNs: larger kernels, fewer normalization/activation layers, LayerNorm instead of BatchNorm, an inverted bottleneck
- Result: keep CNN inductive bias while matching ViT on large-scale benchmarks

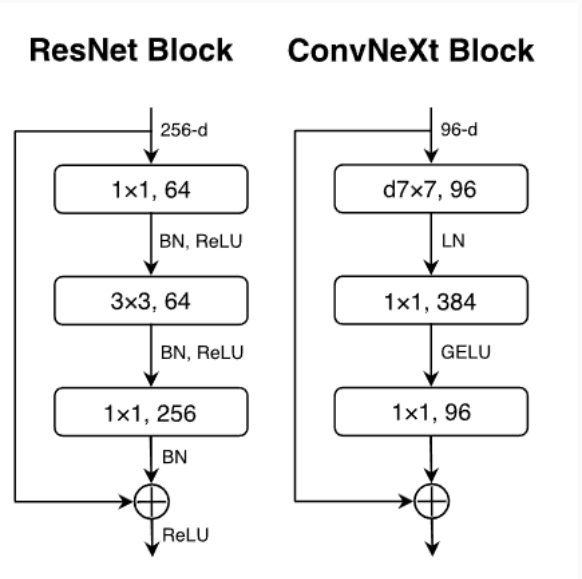


Figure 28: source: convnext paper

Recap: Architectural Ideas

- **Depth**: deep architectures enabled by residual connections, proper inits, and optimizers
- **Kernels**: deep stack of small kernels often favored over large kernels due to more nonlinearity and fewer parameters for the same receptive field
- **Modern architectures (EfficientNet, MobileNet, ConvNeXt)**: finetuned architectural ideas to achieve better compute/memory/accuracy tradeoffs

Practical notes:

- always start from existing architectures / pretrained models and existing training recipes. It is very time consuming to replicate years of optimization for your own custom architecture
- compute depends on kernel, image, and channels. Even deep conv layers are expensive due to the high number of channels.

Cost vs accuracy

Top1 accuracy on ImageNet-1K vs. compute cost of forward pass. The size of the blobs is proportional to the number of network parameters.

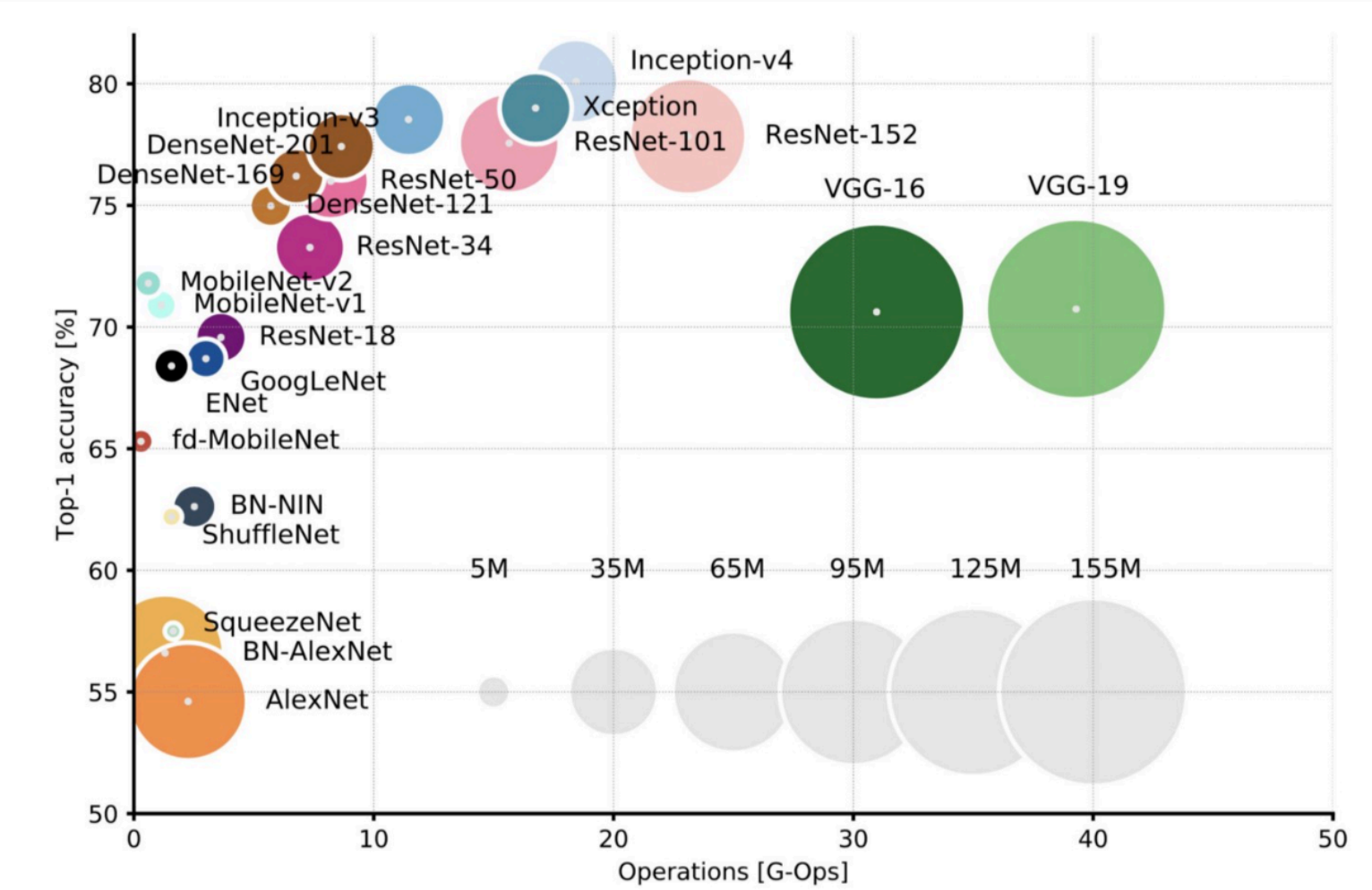


Figure 29: source: https://euge-blog.github.io/2018/09/08/analysis-nn.html

Conclusions

Strengths and Limitations of CNNs

Strengths:

- Locality + weight sharing are good priors for natural images
  - CNNs are the strong default choice on small-to-medium datasets, where a less-constrained architecture (e.g. a ViT) would need to relearn locality and translation structure
- Efficiency in both memory and compute

Limitations:

- Even with a theoretically high receptive field (deep net, large kernels), the effective receptive field is much smaller and concentrated near the center
  - even deep CNNs may struggle to relate distant locations of an image
- Rigid translation equivariance: some tasks do not want position to be treated identically everywhere
  - example: a face-verification system may want different sensitivity near the image center vs. the border

These limitations are solved by **Vision Transformers**.

Key Takeaways

- DNN for vision allow us to switch from static to learned filter banks and enable hierarchical feature learning
- despite their complexity, modern DNNs are very robust to overfitting and optimization issues
- convolution is the natural operation for DNN for vision

References

- visionbook, Chapter 24
- https://github.com/vdumoulin/conv_arithmetic/tree/master