7 Modern Convolutional Neural Networks

Lecture based on "Dive into Deep Learning" **http://D2L.AI** (Zhang et al., 2020)

Prof. Dr. Christoph Lippert

Digital Health & Machine Learning

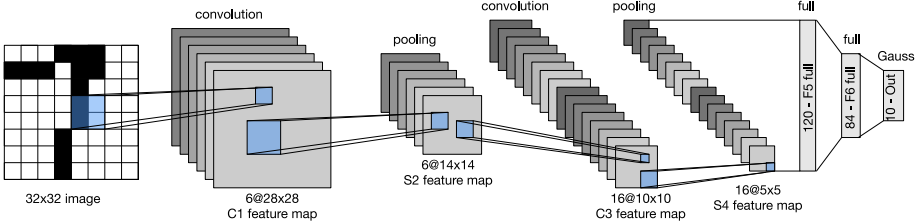
One of the first published convolutional neural networks whose benefit was first demonstrated by Yann Lecun, at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images—LeNet5.

In the 90s, their experiments with **LeNet** gave the first compelling evidence that it was possible to train convolutional neural networks by backpropagation.

Their model achieved outstanding results at the time (only matched by Support Vector Machines at the time) and was adopted to recognize digits for processing deposits in ATM machines.

Some ATMs still runn the code that Yann and his colleague Leon Bottou wrote in the 1990s!

In a rough sense, we can think LeNet as consisting of two parts: (i) a block of convolutional layers; and (ii) a block of fully-connected layers.



- The basic units in the convolutional block are a convolutional layer and a subsequent average pooling layer (note that max-pooling works better, but it had not been invented in the 90s yet).
- In each layer, there is an increase in the number of channels, but the height and width are shrunk considerably.

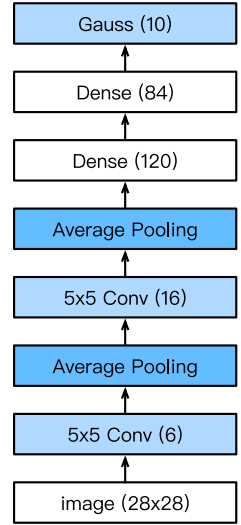


Figure: Compressed notation for LeNet5

Summary

- A convolutional neural network (in short, ConvNet) is a network using convolutional layers.
- In a ConvNet we alternate between convolutions, nonlinearities and often also pooling operations.
- Ultimately the resolution is reduced prior to emitting an output via one (or more) dense layers.
- LeNet was the first successful deployment of such a network.

[1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

- Although LeNet achieved good results on early small data sets, the performance and feasability of training convolutional networks on larger, more realistic datasets had yet to be established.
- Between the early 1990s and the watershed results of 2012, neural networks were often surpassed by other machine learning methods, such as support vector machines.
- Typical computer vision pipelines consisted of manually engineering feature extraction pipelines. Rather than *learn the features*, the features were *crafted*.
- Most of the progress came from having more clever ideas for features, and the learning algorithm was often relegated to an afterthought.

Thus, rather than training *end-to-end* (pixel to classification) systems, classical pipelines looked more like this:

1. Obtain an interesting dataset. In early days, these datasets required expensive sensors (at the time, 1 megapixel images were state of the art).
2. Preprocess the dataset with hand-crafted features based on some knowledge of optics, geometry, other analytic tools.
3. Feed the data through a standard set of feature extractors such as SIFT, the Scale-Invariant Feature Transform, or SURF, the Speeded-Up Robust Features, or any number of other hand-tuned pipelines.
4. Use the resulting representations into your favorite classifier, likely a linear model or kernel method, to learn a classifier.

The most important part of the pipeline was the representation.

Until 2012 the representation was calculated mechanically.

Examples are

- scale-invariant feature transform (SIFT)
- speeded up robust features (SURF)
- histogram of oriented gradients (HOG)
- Bags of visual words

Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, and Juergen Schmidhuber believed that features themselves ought to be learned and ought to be hierarchically composed with multiple jointly learned layers, each with learnable parameters.

Krizhevski, Sutskever and Hinton, 2012 designed a convolutional neural network which achieved excellent performance on ImageNet.

- The lowest layers might come to detect edges, colors, and textures.
- Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, etc.
- Even higher layers might represent whole objects like people, airplanes, dogs, or frisbees.
- Ultimately, the final hidden state learns a compact representation of the image that summarizes its contents such that data belonging to different categories be separated easily.

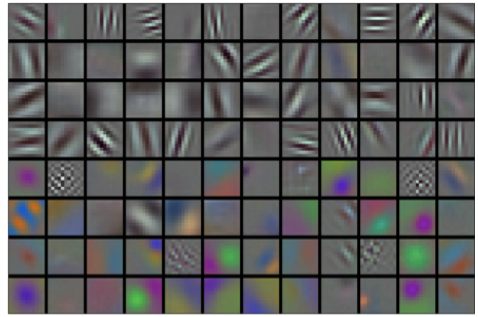


Figure: Image filters learned by the first layer of AlexNet

The ultimate breakthrough in 2012 can be attributed to two key factors.

1. Deep models with many layers require large amounts of data in order to enter the regime where they signficantly outperform traditional methods based on convex optimizations (e.g. linear and kernel methods).

   In 2009, the ImageNet data set was releasedconsisting of 1 million examples, 1,000 each from 1,000 distinct categories of objects.

   Fei-Fei Li leveraged Google Image Search to prefilter large candidate sets for each category and employed the Amazon Mechanical Turk crowdsourcing pipeline to determine the associated category.

2. Deep learning models are varacious consumers of compute cycles.

   Graphical processing units (GPUs) proved to be a game changer in make deep learning feasible.

   A major breakthrough came in 2012, when Alex Krizhevsky and Ilya Sutskever implemented a deep convolutional neural network that could run on GPU hardware: AlexNet

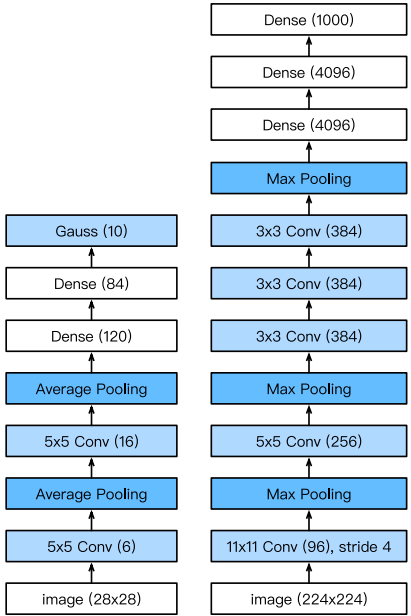


Figure: LeNet (left) and AlexNet (right)

AlexNet

- 8-layer convolutional neural network
- AlexNet won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin.
- After the last convolutional layer are two fully-connected layers with 4096 outputs.
- These two huge fully-connected layers produce model parameters of nearly 1 GB.
- Due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that each of their two GPUs could store and compute only its half of the model.
- AlexNet changed the sigmoid activation function to a simpler ReLU activation function. (easier to evaluate and avoid vanishing gradients)
- AlexNet regularizes the fully-connected layer by dropout, while LeNet only uses weight decay.

Summary — AlexNet

- AlexNet has a similar structure to that of LeNet, but uses more convolutional layers and a larger parameter space to fit the large-scale data set ImageNet.
- Today AlexNet has been surpassed by much more effective architectures but it is a key step from shallow to deep networks that are used nowadays.
- Although it seems that there are only a few more lines in AlexNet's implementation than in LeNet, it took the academic community many years to embrace this conceptual change and take advantage of its excellent experimental results. This was also due to the lack of efficient computational tools.
- Dropout, ReLU and preprocessing were the other key steps in achieving excellent performance in computer vision tasks.

The design of neural nework architectures had grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers.

The idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University.

In their VGG network, these repeated structures are implemented using loops and subroutines.

The VGG Network consists of two parts:

1. 5 convolutional blocks

   One VGG block consistis of a sequence of convolutional layers, followed by a max pooling layer for spatial downsampling.

   first block has 64 output channels and each subsequent block doubles the number of output channels, until that number reaches 512.

   Simonyan and Zisserman, 2014 employed convolutions with 3×3 kernels and 2×2 max pooling with stride of 2 (halving the resolution after each block).

2. Fully-connected layers.

Since this network uses 8 convolutional layers and 3 fully-connected layers, it is often called VGG-11.

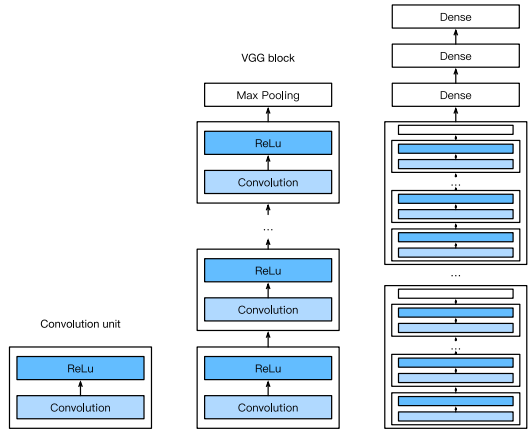


Figure: Designing a network from building blocks

Summary — VGG

- VGG-11 constructs a network using reusable convolutional blocks. Different VGG models can be defined by the differences in the number of convolutional layers and output channels in each block.
- The use of blocks leads to very compact representations of the network definition. It allows for efficient design of complex networks.
- In their work Simonyan and Ziserman experimented with various architectures. In particular, they found that several layers of deep and narrow convolutions (i.e. 3×3) were more effective than fewer layers of wider convolutions.

LeNet, AlexNet, and VGG all share a common design pattern:

- extract features exploiting *spatial* structure via a sequence of convolutions and pooling layers
- then post-process the representations via fully-connected layers.

The improvements upon LeNet by AlexNet and VGG mainly lie in how later networks *widen* and *deepen* these two modules.

Alternatively, one could imagine using fully-connected layers earlier in the process.

- However, a careless use of dense layers might give up the the spatial structure of the representation entirely, **Network in Network** (NiN) blocks offer an alternative.
- They were proposed by Lin, Chen and Yan, 2013 based on a very simple insight—to use an MLP on the channels for each pixel separately.

NiN uses 1D conv. to apply a fully-connected layer to each pixel location.

- In 1D conv. the weights are tied.
- Each element in the spatial dimension (height and width) can be interpreted as an example and the channel as equivalent to a feature.
- ReLu activations

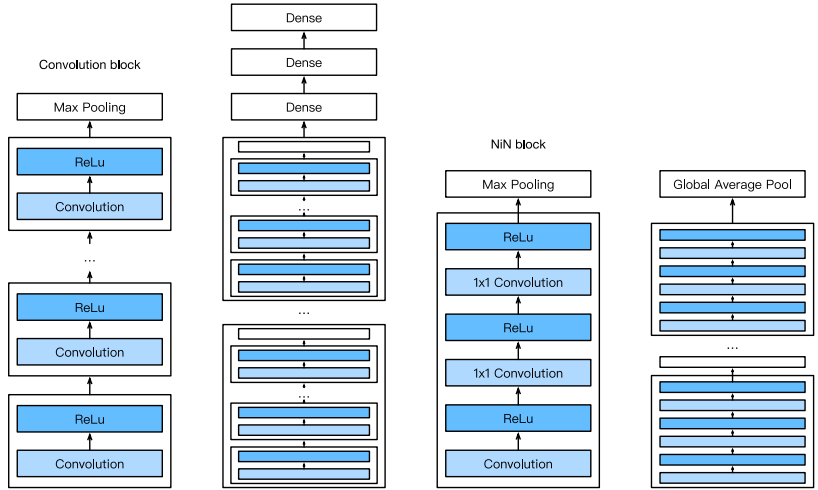


Figure: The figure on the left shows the network structure of AlexNet and VGG, and the figure on the right shows the network structure of NiN.

NiN avoids dense connections altogether by introducing a number of output channels equal to the number of label classes.

- *global* average pooling layer yields a vector of logits.
- Significantly reduces number of parameters.
- requires increased training time

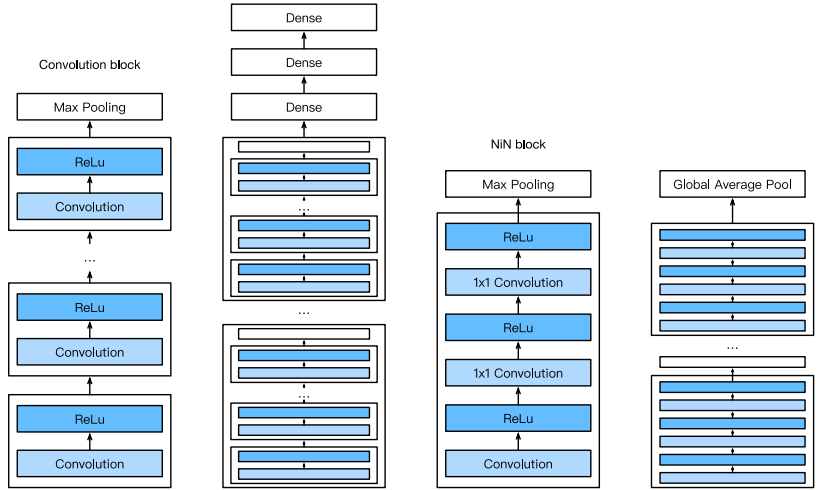


Figure: The figure on the left shows the network structure of AlexNet and VGG, and the figure on the right shows the network structure of NiN.

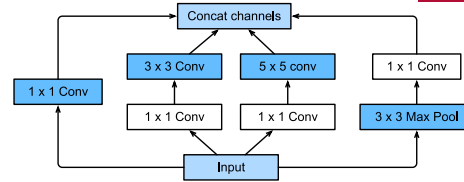
Summary — NiN

- NiN uses blocks consisting of a convolutional layer and multiple 1×1 convolutional layer. This can be used within the convolutional stack to allow for more per-pixel nonlinearity.
- NiN removes the fully connected layers and replaces them with global average pooling (i.e. summing over all locations) after reducing the number of channels to the desired number of output classes.
- Removing the dense layers reduces overfitting.
- NiN has dramatically fewer parameters.
- The NiN design influenced many subsequent convolutional neural networks designs.

In 2014, Szegedy et al., 2014 won the ImageNet Challenge, proposing a structure that combined the strengths of the NiN and repeated blocks paradigms.

- One focus of the paper was to address the question of which sized convolutional kernels are best.
- After all, previous popular networks employed choices as small as 1×1 and as large as 11×11.
- One insight in this paper was that sometimes it can be advantageous to employ a combination of variously-sized kernels.

The basic convolutional block in **GoogLeNet** is called an Inception block, likely named due to a quote from the movie **Inception** ("We Need To Go Deeper"), which launched a viral meme.



An inception block consists of four parallel paths.

- The first three paths use convolutional layers with window sizes of 1×1, 3×3, and 5×5 to extract information from different spatial sizes.
- The middle two paths perform a 1×1 convolution on the input to reduce the number of input channels, reducing the model's complexity.
- The fourth path uses a 3×3 maximum pooling layer, followed by a 1×1 convolutional layer to change the number of channels.

The four paths all use appropriate padding to give the input and output the same height and width.

Finally, the outputs along each path are concatenated along the channel dimension and comprise the block's output.

To gain some intuition for why this network works so well, consider the combination of the filters.

- They explore the image in varying ranges.
- Details at different extents can be recognized efficiently by different filters.
- Different amounts of parameters for different ranges (e.g. more for short range but not ignore the long range entirely).

GoogLeNet

- 9 inception blocks
- global average pooling to generate its estimates.
- Maximum pooling between inception blocks to reduce the dimensionality.
- The first part is identical to AlexNet and LeNet

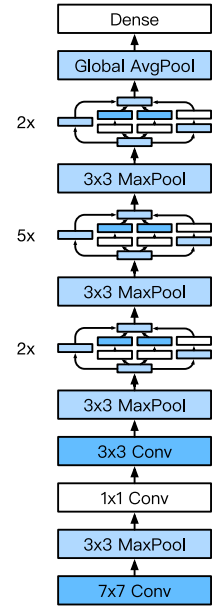


Figure: Full GoogLeNet Model

Summary — GoogLeNet

- The Inception block is equivalent to a subnetwork with four paths. It extracts information in parallel through convolutional layers of different window shapes and maximum pooling layers.
  - 1×1 convolutions reduce channel dimensionality on a per-pixel level.
  - Max-pooling reduces the resolution.
- GoogLeNet connects multiple well-designed Inception blocks with other layers in series.
  - The ratio of the number of channels assigned in the Inception block is obtained through a large number of experiments on the ImageNet data set.
- GoogLeNet, as well as its succeeding versions, was one of the most efficient models on ImageNet, providing similar test accuracy with lower computational complexity.

[1] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., & Anguelov, D. & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

[2] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

[3] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2818-2826).

[4] Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017, February). Inception-v4, inception-resnet and the impact of residual connections on learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 4, p. 12).

Training deep models is difficult and getting them to converge in a reasonable amount of time can be tricky.

batch normalization is one popular and effective technique that has been found to accelerate the convergence of deep nets that is one of the ingredients to routinely train networks with over 100 layers.

Let's review some of the practical challenges when training deep networks.

1. Data preprocessing often proves to be a crucial consideration for effective statistical modeling.
   - Typically, we standardize our input features to each have a mean of *zero* and variance of *one*.
   - Standardizing input data typically makes it easier to train models since parameters are a-priori at a similar scale.
2. For a typical MLP or CNN, the activations in intermediate layers of the network assume different orders of magnitude (both across nodes in the same layer, and over time due to updating the model's parameters).
3. The authors of the batch normalization technique postulated that this drift in the distribution of activations could hamper the convergence of the network.
   - Intuitively, we might conjecture that if one layer has activation values that are 100x that of another layer, we might need to adjust learning rates adaptively per layer (or even per node within a layer).
4. Deeper networks are complex and easily capable of overfitting. This means that regularization becomes more critical. Empirically, we note that even with dropout, models can overfit badly and we might benefit from other regularization heuristics.

In 2015, Ioffe and Szegedy introduced Batch Normalization (BN), a heuristic that has proved immensely useful for improving the reliability and speed of convergence when training deep models.

- BN normalizes the activations of a hidden layer node by subtracting its mean and dividing by its standard deviation, estimating both based on the current minibatch.
- With large enough minibatches, the approach proves effective and stable.[1]

The activation at a given layer is transformed from $\mathbf{x}$ to

$$\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta$$

Here, $\hat{\mu}$ is the estimate of the mean and $\hat{\sigma}$ is the estimate of the variance.

[1] Note that if our batch size was 1, we wouldn't be able to learn anything because during training, every hidden node would take value 0.

$$\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta$$

- The result is that the activations are approximately rescaled to zero mean and unit variance.
- Since this may not be quite what we want, we allow for a coordinate-wise scaling coefficient γ and an offset β.

Consequently, the activations for intermediate layers cannot diverge any longer: we are actively rescaling them back to a given order of magnitude via μ and σ.

- This normalization allows us to be more aggressive in picking large learning rates.
- To address the fact that in some cases the activations may actually *need* to differ from standardized data, BN also introduces scaling coefficients γ and an offset β.

$$\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta$$

In principle, we might want to use all of our training data to estimate the mean and variance.

- The activations correpsonding to each example change each time we update our model.
- Therefore, BN uses only the current minibatch for estimating $\hat{\mu}$ and $\hat{\sigma}$.
- As we normalize based only on the *currect batch* the procedure is called *batch normalization*
- To indicate which minibatch $\mathcal{B}$ we draw this from, we denote the quantities with $\hat{\mu}_\mathcal{B}$ and $\hat{\sigma}_\mathcal{B}$.

$$\hat{\mu}_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \quad \text{and} \quad \hat{\sigma}^2_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \mu_\mathcal{B})^2 + \epsilon$$

- Note that a small constant $\epsilon > 0$ is added to the variance estimate to ensure that we never end up dividing by zero.
- The estimates $\hat{\mu}_{\mathcal{B}}$ and $\hat{\sigma}_{\mathcal{B}}$ counteract the scaling issue by using unbiased but noisy estimates of mean and variance.
- Variation in $\hat{\mu}_{\mathcal{B}}$ and $\hat{\sigma}_{\mathcal{B}}$ appears to act as a form of regularization, conferring benefits (as observed empirically) in mitigating overfitting.
- Recent preliminary research, Teye, Azizpour and Smith, 2018 and Luo et al, 2018 relate the properties of BN to Bayesian Priors and penalties respectively.

The batch normalization methods for fully-connected layers and convolutional layers are slightly different. This is due to the dimensionality of the data generated by convolutional layers.

Usually we apply the batch normalization layer between the affine transformation and the activation function in a fully-connected layer.

Denote by $\mathbf{u}$ the input and by $\mathbf{x} = \mathbf{W}\mathbf{u} + \mathbf{b}$ the output of the linear transform. This yields the following variant of BN:

$$\mathbf{y} = \phi(\mathrm{BN}(\mathbf{x})) = \phi(\mathrm{BN}(\mathbf{W}\mathbf{u} + \mathbf{b}))$$

Recall that mean and variance are computed on the *same* minibatch $\mathcal{B}$ on which the transformation is applied.

Also recall that the scaling coefficient γ and the offset β are parameters that need to be learned.

For convolutional layers, batch normalization occurs after the convolution computation and before the application of the activation function.

If the convolution computation outputs multiple channels, we need to carry out batch normalization for *each* of the outputs of these channels, and each channel has an independent scale parameter and shift parameter, both of which are scalars.

Assume that there are m examples in the mini-batch. On a single channel, we assume that the height and width of the convolution computation output are p and q, respectively. We need to carry out batch normalization for $m \times p \times q$ elements in this channel simultaneously.

While carrying out the standardization computation for these elements, we use the same mean and variance. In other words, we use the means and variances of the $m \times p \times q$ elements in this channel rather than one per pixel.

At prediction time, we might not have the luxury of computing offsets per batch—we might be required to make one prediction at a time.

Secondly, the uncertainty in μ and σ, as arising from a minibatch are undesirable once we've trained the model.

One way to mitigate this is to compute more stable estimates on a larger set for once (e.g. via a moving average) and then fix them at prediction time.

Consequently, BN behaves differently during training and at test time.

Summary — Batch Norm

- During model training, batch normalization continuously adjusts the intermediate output of the neural network by utilizing the mean and standard deviation of the mini-batch, so that the values of the intermediate output in each layer throughout the neural network are more stable.
- The batch normalization methods for fully connected layers and convolutional layers are slightly different.
- Like a dropout layer, batch normalization layers have different computation results in training mode and prediction mode.
- Batch Normalization has many beneficial side effects, primarily that of regularization. On the other hand, the original motivation of reducing covariate shift seems not to be a valid explanation.

[1] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

As we design increasingly deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network.

- How can we design networks where adding layers makes networks strictly more expressive rather than just different?

- Let $\mathcal{F}$ be the class of functions that a specific network architecture can reach (incl. learning rates and other hyperparameters).
- That is, for all $f \in \mathcal{F}$ there exists a set of parameters W that can be learned from data.
- Let f^* be the unkown target function that we would like to learn.
- In practice, we try to find some $f^*_{\mathcal{F}} \in \mathcal{F}$ which is close enough.

  For instance, we might try finding it by solving:

$$f^*_{\mathcal{F}} := \operatorname*{argmin}_f L(X, Y, f) \text{ subject to } f \in \mathcal{F}$$

- It is only reasonable to assume that if we design a different and more powerful architecture $\mathcal{F}'$ we would expect that $f^*_{\mathcal{F}'}$ is 'better' than $f^*_{\mathcal{F}}$.
- However, unless $\mathcal{F} \not\subseteq \mathcal{F}'$ there is no guarantee.

  In fact, $f^*_{\mathcal{F}'}$ might well be worse.

This is a situation that we often encounter in practice:

- adding layers doesn't only make the network more expressive, it also changes it in sometimes not quite so predictable ways.
- Only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network.
- Therefore, He et al, 2016 included an identity mapping $f(\mathbf{x}) = \mathbf{x}$ in CNN layers

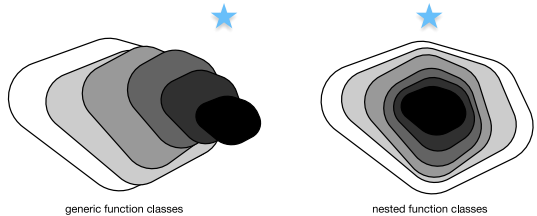


Figure: Left: non-nested function classes. The distance may in fact increase as the complexity increases. Right: with nested function classes this does not happen.

Building block of **ResNet**, He et al, 2015 (2015 ImageNet Visual Recognition Challenge winner) is the **residual block**:

- The identity function rather than the null $f(\mathbf{x}) = 0$ should be the simplest function within each layer.
- The portion within the dotted-line box in the right image now only needs to parametrize the *deviation* from the identity, since we return $\mathbf{x} + f(\mathbf{x})$.
- in the residual block $f(\mathbf{x}) = 0$ yields the identity

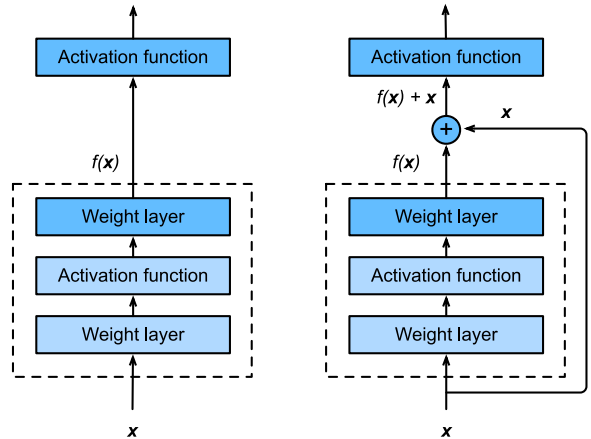
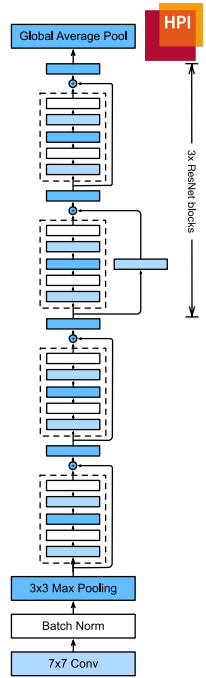


Figure: The difference between a **regular block** (left) and a **residual block** (right). In the latter case, we can short-circuit the convolutions.

ResNet-18 follows VGG's full 3×3 convolutional layer design.

- Residual block has two 3×3 convolutional layers with equal number of output channels.
- Each convolutional layer is followed by a batch norm layer and a ReLU.
- Then, we add the input before the final ReLU.
- Design requires that the output of the two convolutional layers are the same shape as the input (for addition).
- If we want to change the number of channels or the the stride, we need to introduce an additional 1×1 convolutional layer to achieve the needed shape for the addition.
- By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as ResNet-152.



Summary — ResNet

- Residual blocks allow for a parametrization relative to the identity function $f(\mathbf{x}) = \mathbf{x}$.
- Adding residual blocks increases the function complexity in a well-defined manner.
- We can train an effective deep neural network by having residual blocks pass through cross-layer data channels.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.

[1] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[2] He, K., Zhang, X., Ren, S., & Sun, J. (2016, October). Identity mappings in deep residual networks. In European Conference on Computer Vision (pp. 630-645). Springer, Cham.

Recall the Taylor expansion for functions.

- For scalars the Taylor expansion decomposes the function into increasingly higher order terms

$$f(x) = f(0) + f'(x)x + \frac{1}{2}f''(x)x^2 + \frac{1}{6}f'''(x)x^3 + o(x^3)$$

- In a similar vein, ResNet decomposes f into a simple linear term and a more complex nonlinear one.

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x})$$

What if we want to go beyond two terms?

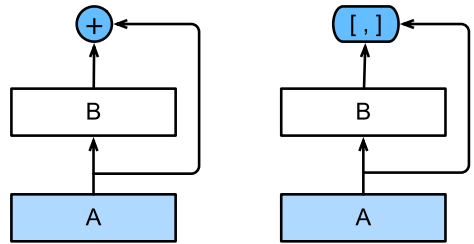
- A solution was proposed by Huang et al, 2016 in the form of DenseNet, an architecture that reported record performance on ImageNet.

The main difference between **ResNet** (left) and **DenseNet** (right) in cross-layer connections:

- use of **addition** and use of **concatenation**.
- As a result we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions.

$$\mathbf{x} \to \left[\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), f_3(\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), \ldots\right]$$

- In the end, all these functions are combined in an MLP to reduce the number of features again.
- the dependency graph between variables becomes dense.
- The last layer of such a chain is densely connected to all previous layers.

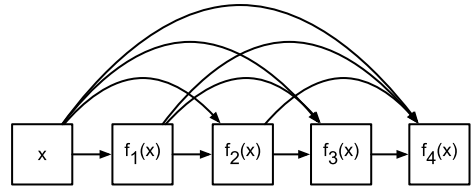


The main difference between **ResNet** (left) and **DenseNet** (right) in cross-layer connections:

- use of **addition** vs. use of **concatenation**.
- As a result we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions.

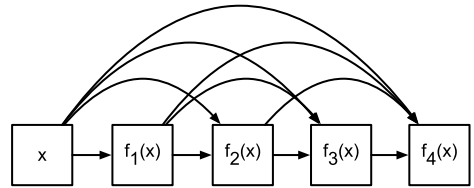
$$\mathbf{x} \rightarrow \left[\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), f_3(\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), \ldots\right]$$

- In the end, all these functions are combined in an MLP to reduce the number of features again.



The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense.

- The last layer of such a chain is densely connected to all previous layers.



The main components that compose a DenseNet are

- dense blocks that define how the inputs and outputs are concatenated
- transition layers that control the number of channels so that it is not too large.

Summary — DenseNet

- In terms of cross-layer connections, unlike ResNet, where inputs and outputs are added together, DenseNet concatenates inputs and outputs on the channel dimension.
- The main units that compose DenseNet are dense blocks and transition layers.
- We need to keep the dimensionality under control when composing the network by adding transition layers that shrink the number of channels again.

[1] Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (Vol. 1, No. 2).