Classic Deep CNN Architectures

Naeemullah Khan
naeemullah.khan@kaust.edu.sa

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

LMH
Lady Margaret Hall

July 25, 2025

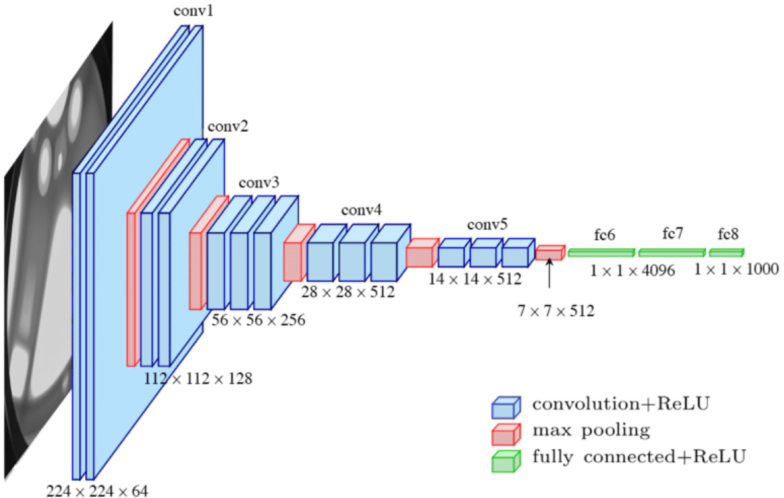

conv1
conv2
conv3
conv4
conv5
fc6
fc7
fc8
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU

Table of Contents

Table of Contents (cont.)

⇒ Learning Outcomes

By the end of this session, you will be able to:

- See how deep CNNs have changed over time.
- Get to know popular models like LeNet, AlexNet, VGG, Inception, ResNet, and MobileNet.
- Study influential deep CNN architectures including AlexNet, VGG, InceptionNet, ResNet, MobileNet, etc.
- Learn simple tricks to make your models work better, like using more data or transfer learning.
- Model performance enhancement techniques including Data Augmentation, Transfer Learning, and Batch Normalization.
- Spot some challenges in this area and think about what's next.

CNN Architectures

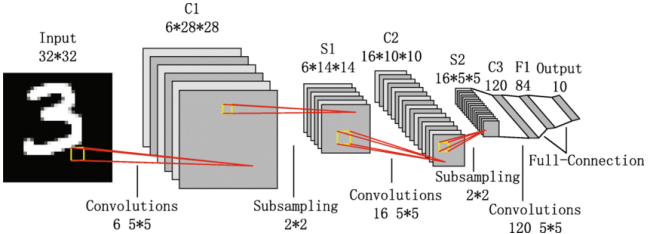
Most Notable CNN-based Architectures

Over time, researchers built advanced CNN architectures to improve performance and efficiency. These architectures introduced key innovations:

- **LeNet [LeCun et al., 1998]**: The first CNN architecture, designed for handwritten digit recognition.
- **AlexNet [Krizhevsky et al. 2012]**: The first CNN to achieve breakthrough performance on image classification.
- **VGGNet [Simonyan and Zisserman, 2014]**: Used very deep networks (up to 19 layers).
- **InceptionNet (GoogLeNet) [Szegedy et al., 2014]**: Used multiple filter sizes per layer (Inception modules).
- **ResNet [He et al., 2015]**: Introduced skip connections for training very deep networks.
- **EfficientNet [Tan and Le, 2019]**: Found a scaling method that simultaneously scales a CNN's depth, width, and resolution optimally using a single scaling coefficient.
- **MobileNet [Howard et al., 2017]**: Designed for mobile and embedded vision applications, using depthwise separable convolutions.

CNN Architectures: **LeNet-5**

LeNet-5



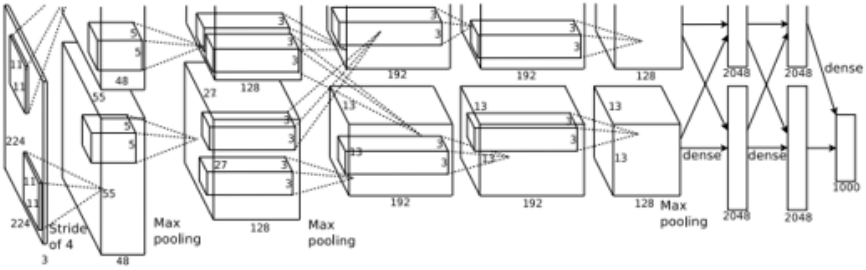
LeNet-5 architecture.

- LeNet-5 is a pioneering convolutional neural network (CNN) architecture developed by Yann LeCun and his collaborators in the late 1980s and early 1990s.
- It was built to recognize handwritten digits, like those in the MNIST dataset.
- The network uses layers that look for patterns (convolution), shrink images (pooling), and finally make decisions (fully connected layers).
- LeNet-5 introduced ideas like using filters to find features and pooling to reduce size, which are now common in modern CNNs.

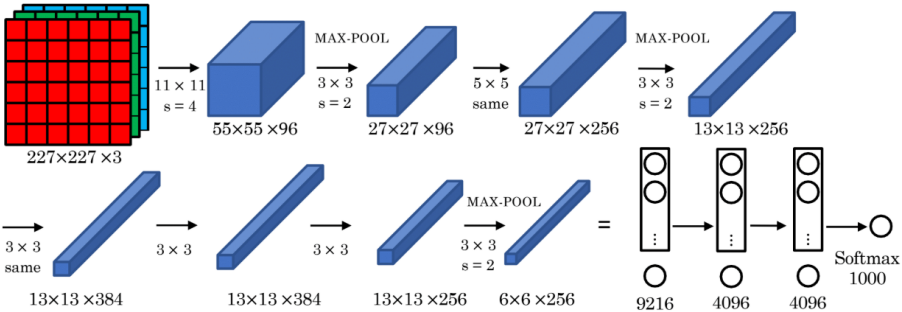
CNN Architectures: **AlexNet**

AlexNet

- First big improvement in image classification.
- Made use of CNN, pooling, dropout, ReLU and training on GPUs.
- 5 convolutional layers, followed by max-pooling layers; with three fully connected layers at the end



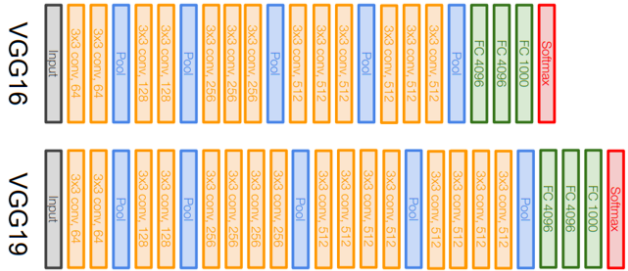
AlexNet (cont.)



[Krizhevsky et al., 2012. ImageNet classification with deep convolutional neural networks]

CNN Architectures: **VGGNet**

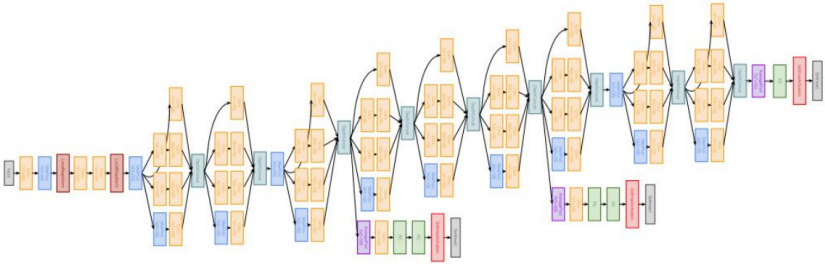
- **Improvement over AlexNet:** Uses a deeper network with small filters instead of a shallow network with larger filters.
- A stack of 3x3 conv layers (vs. 11x11 conv) has same receptive field, more non-linearities, fewer parameters and deeper network representation.
- Two variants: VGG16 or VGG19 conv layers plus 3 FC layers.



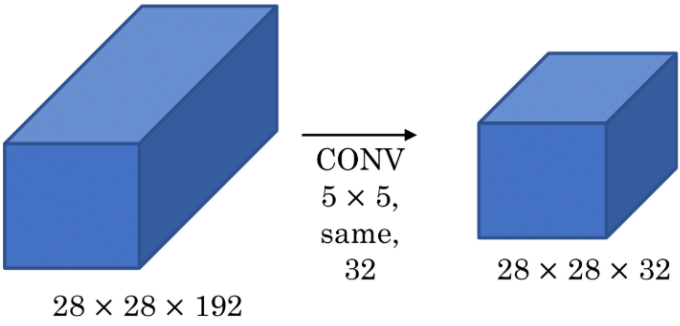
CNN Architectures: **InceptionNet**

InceptionNet

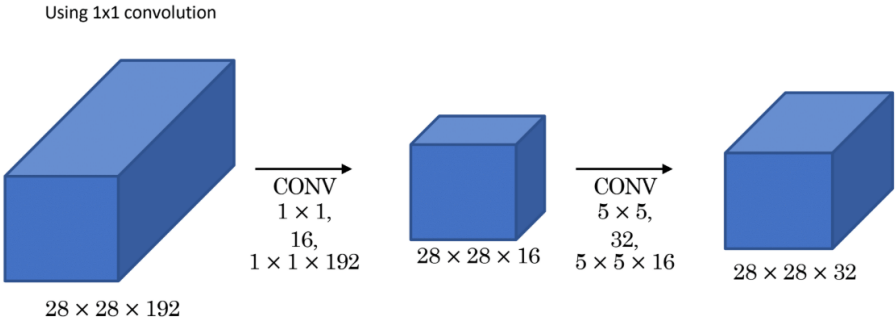
- InceptionNet goes **deeper** with 22 layers.
- It is much more **efficient**: only 5 million parameters.
- Uses a special building block called the **Inception module**.
- Adds **1x1 convolutions** (bottleneck layers) to make the network faster and smarter.



The problem of computational cost

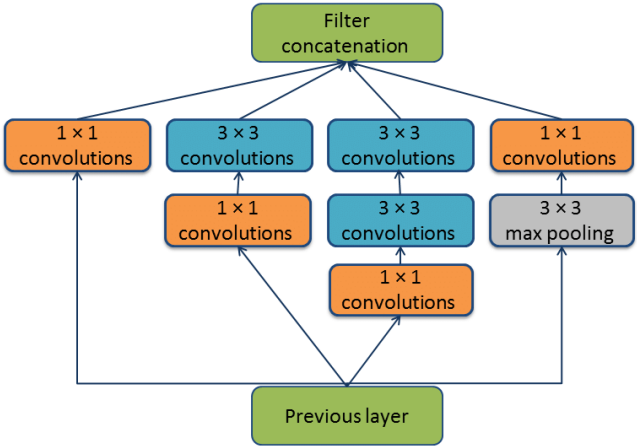


InceptionNet (cont.)

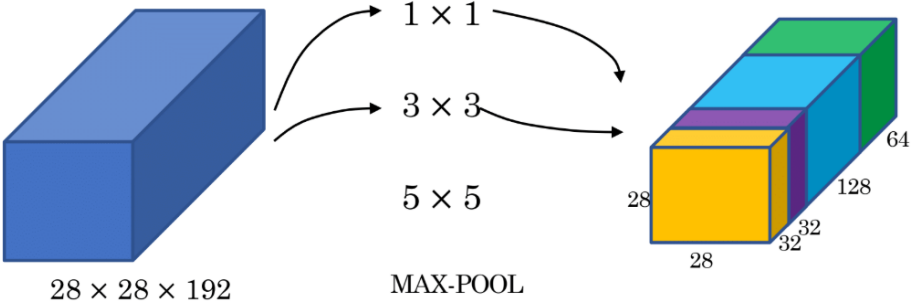


InceptionNet (cont.)

- **Inception module:** Uses multiple filter sizes (1×1, 3×3, 5×5), in parallel, to capture different features, then combines their outputs.



InceptionNet (cont.)



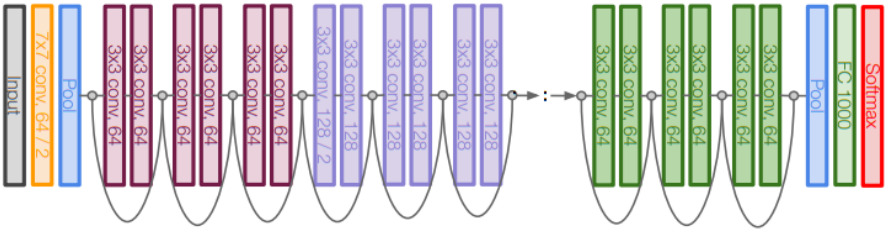
[Szegedy et al. 2014. Going deeper with convolutions]

CNN Architectures: **ResNet**

ResNet: Motivation

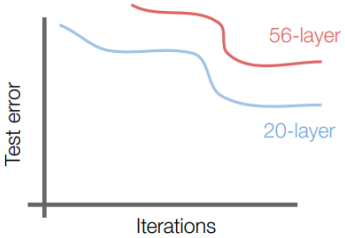
- **Problem:** Making networks deeper does not always improve accuracy.
- **Why?** In very deep networks, gradients become extremely small as they move backward through layers, making learning slow or stopping it altogether (**vanishing gradient problem**).
- **Solution:** Residual Network (ResNet) introduces **skip connections (residuals)**, allowing information to flow more easily.

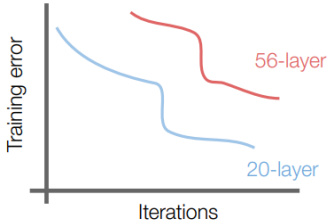
- Very deep networks using residual connections
- 152-layer model for ImageNet
- Stacked Residual Blocks
- **Residual:** A shortcut connection that helps the network pass information through layers more easily.



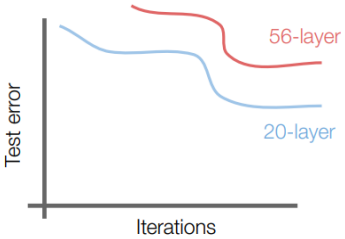
- What happens when we continue stacking deeper layers on a "plain" convolutional neural network?

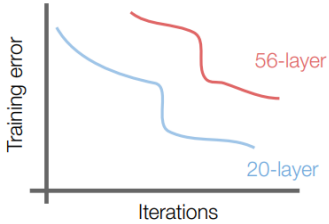
- What happens when we continue stacking deeper layers on a "plain" convolutional neural network?





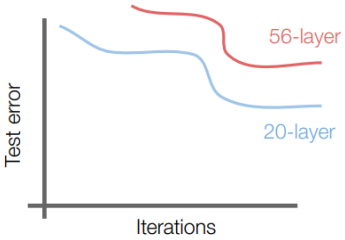
- What happens when we continue stacking deeper layers on a "plain" convolutional neural network?

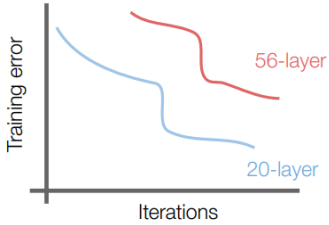




- 56-layer model performs worse on both test and training error

- What happens when we continue stacking deeper layers on a "plain" convolutional neural network?





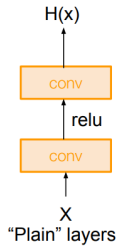
- 56-layer model performs worse on both test and training error
- The deeper model performs worse, but it's not caused by overfitting!

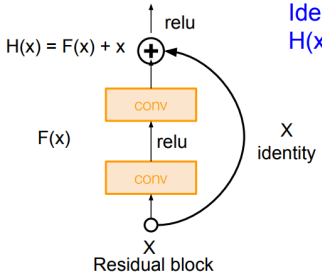
- **Fact:** Deep models have more representation power (more parameters) than shallower models.

- **Fact:** Deep models have more representation power (more parameters) than shallower models.
- **Hypothesis:** The problem is an optimization problem, deeper models are harder to optimize

- **Fact:** Deep models have more representation power (more parameters) than shallower models.
- **Hypothesis:** The problem is an optimization problem, deeper models are harder to optimize
- **Solution:** Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

- **Fact:** Deep models have more representation power (more parameters) than shallower models.
- **Hypothesis:** The problem is an optimization problem, deeper models are harder to optimize
- **Solution:** Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping





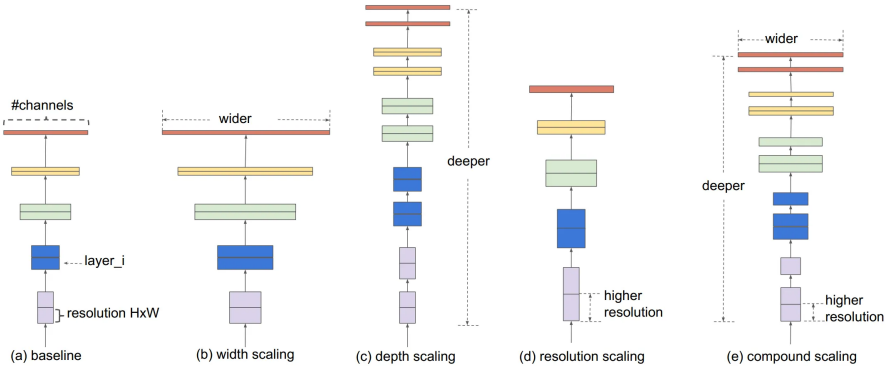
Identity mapping:
H(x) = x if F(x) = 0

CNN Architectures: **EfficientNet**

EfficientNet: Motivation

- **Problem:** To improve accuracy, we can:
  - Increase the number of layers (**depth**)
  - Increase the number of neurons in each layer (**width**)
  - Use higher-resolution images (**resolution**)

  But finding the right balance of these three was largely based on trial and error.
- **Solution:** EfficientNet introduced a **compound scaling** method—a mathematical formula to systematically find the optimal balance among **depth**, **width**, and **resolution**.



EfficientNet Scaling.

EfficientNet (cont.)

- **Compound Scaling** Uses a single **scaling coefficient** (ϕ) to control:
  - **Network Depth** (α^ϕ) $\rightarrow$ More layers
  - **Network Width** (β^ϕ) $\rightarrow$ More channels per layer
  - **Input Resolution** (γ^ϕ) $\rightarrow$ Larger input images
- The goal: find α, β, γ that balance accuracy & efficiency, then scale up optimally by increasing the global coefficient ϕ.

EfficientNet (cont.)

- EfficientNet optimizes depth, width, and resolution using this constraint:

$$\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$

- **Why this equation?**
  - Increasing **depth** (α) increases FLOPs **linearly**.
  - Increasing **width** (β) increases FLOPs **quadratically** (β^2).
  - Increasing **resolution** (γ) increases FLOPs **quadratically** (γ^2).
- To double total FLOPs, the three factors must be balanced together.

EfficientNet (cont.)

- The authors of EfficientNet searched for the best scaling factors on a small baseline model.
- They found:

$$\alpha = 1.2, \quad \beta = 1.1, \quad \gamma = 1.15$$

EfficientNet Scaling: B0 to B7

- The EfficientNet family (B0 to B7) is generated using:

$$\text{Depth} = 1.2^{\phi}, \quad \text{Width} = 1.1^{\phi}, \quad \text{Resolution} = 1.15^{\phi}$$

Model	ϕ	**Depth** (α^{ϕ})	**Width** (β^{ϕ})
B0	0	$1.2^0 = 1.0$	$1.1^0 = 1.0$
B1	1	$1.2^1 = 1.2$	$1.1^1 = 1.1$
B2	2	$1.2^2 = 1.44$	$1.1^2 = 1.21$
B3	3	$1.2^3 = 1.73$	$1.1^3 = 1.33$
B4	4	$1.2^4 = 2.07$	$1.1^4 = 1.46$
B5	5	$1.2^5 = 2.49$	$1.1^5 = 1.61$
B6	6	$1.2^6 = 2.99$	$1.1^6 = 1.77$
B7	7	$1.2^7 = 3.58$	$1.1^7 = 1.94$

Scaling EfficientNet from B0 to B7

- We multiply these scaling factors by the baseline EfficientNet-B0 values and round to the nearest integer to get the new depth, width, and resolution for each model.

- EfficientNet models achieve state-of-the-art accuracy with significantly fewer parameters and FLOPs.

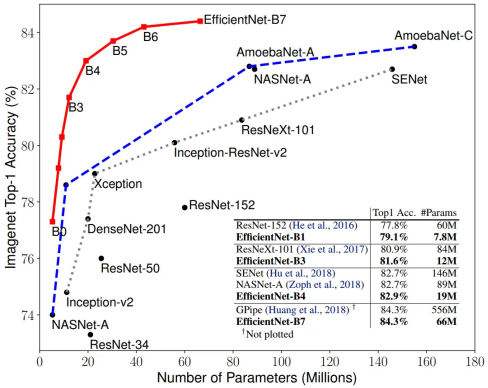
EfficientNet

- EfficientNet models achieve state-of-the-art accuracy with significantly fewer parameters and FLOPs.
- EfficientNet-B7 reaches 84.4% Top-1 and 97.3% Top-5 accuracy on ImageNet.

EfficientNet

- EfficientNet models achieve state-of-the-art accuracy with significantly fewer parameters and FLOPs.
- EfficientNet-B7 reaches 84.4% Top-1 and 97.3% Top-5 accuracy on ImageNet.
- More efficient than previous CNN models—8.4x smaller and 6.1x faster than competitors.

EfficientNet Performance



EfficientNet Performance on Imagenet.

CNN Architectures: **MobileNet**

Motivation for MobileNets

- **Why MobileNets?**
  - Small-sized models are crucial for mobile and embedded devices.
  - MobileNets reduce computational cost and memory usage while maintaining good accuracy.
- **Key Idea:**
  - Use **depthwise-separable convolutions** to significantly reduce computation compared to standard convolutions.

Computational Cost of Convolutions

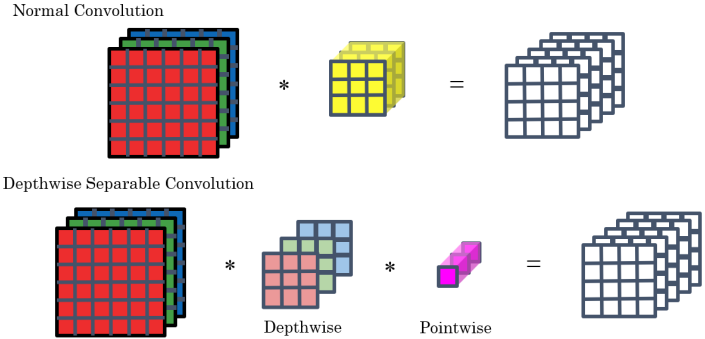
- **Computational cost of standard convolution:**

$$\text{Cost} = \#\text{ filter params} \times \#\text{ filter positions} \times \#\text{ filters}$$

- Filters operate on all input channels, increasing computation significantly.

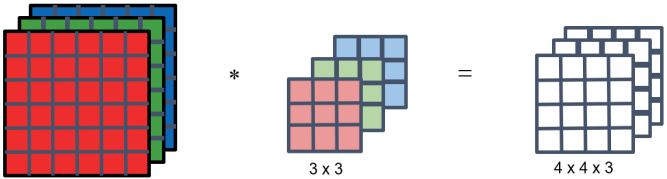
Depthwise-Separable Convolutions

- Split standard convolution into two steps:
  - **Depthwise Convolution:** Applies a single filter per input channel.
  - **Pointwise Convolution:** Combines outputs from depthwise convolution.
- **Key Benefit:** Reduces computational cost significantly compared to standard convolution.



Depthwise Convolution

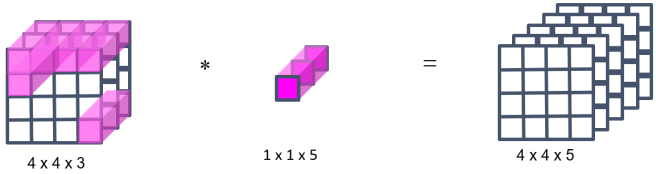
- Operates on each input channel separately.



Computational cost = #filter params x # filter positions x # of filters

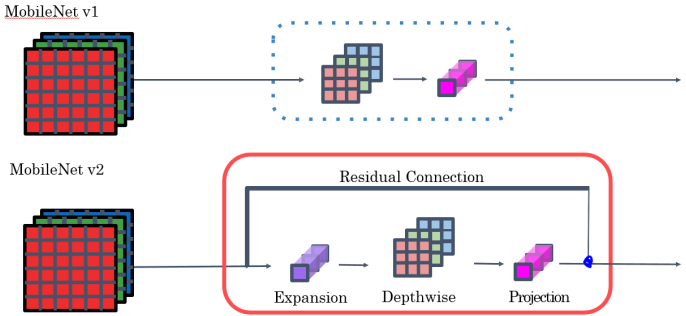
Pointwise Convolution

- Combines outputs from depthwise convolution using 1×1 convolutions (mixes channels).



Computational cost = #filter params x # filter positions x # of filters

- **MobileNet v2:**
  - Adds **residual connections**.
  - Introduces:
    - **Expansion step:** Expands input dimensions before depthwise convolution.
    - **Projection step:** Reduces dimensions after processing.



Summary of CNN Architectures

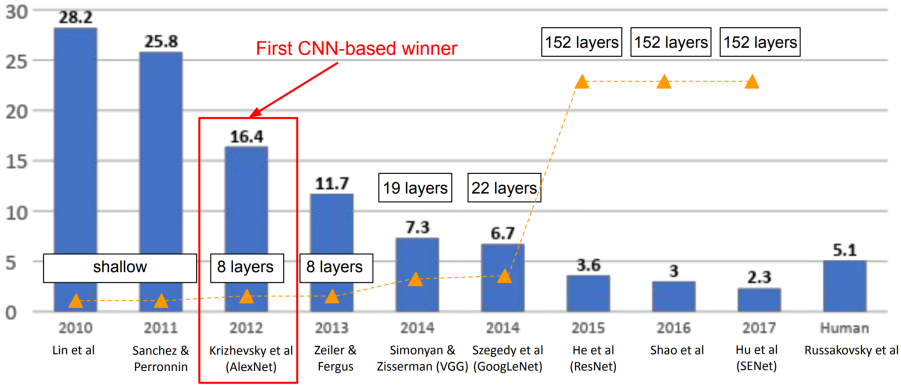
Model	Year	Key Idea	#Params	Strength
LeNet-5	1998	Early CNN for digit recognition	∼60K	Pioneered CNNs
AlexNet	2012	Deeper + ReLU + Dropout	∼60M	Started Deep Learning
VGG	2014	3×3 filters, uniform design	∼138M	Simplicity
Inception	2014	Multi-scale filters	∼6.8M	Efficient
ResNet	2015	Skip connections	∼25M (50)	Very deep models
EfficientNet	2019	Compound scaling	∼5M (B0)	Accuracy/efficiency tr
MobileNet	2017	Lightweight architecture	∼4M	Mobile-ready

CNN Datasets: **ImageNet**

ImageNet

- The most extensive data for Image Classification
- 3 RGB channels from 0 to 255
- 14,197,122 images
- 1000 classes

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



CNN: **Data Augmentation**

What is Data Augmentation?

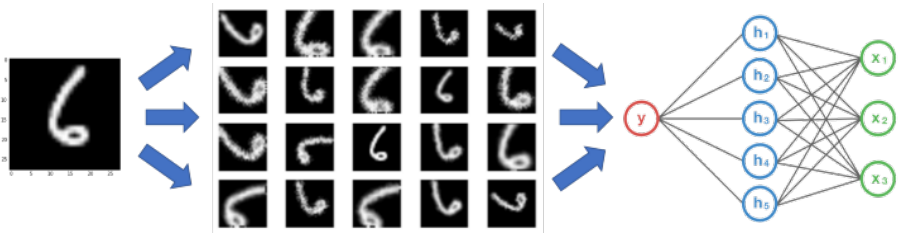
- **Definition:** Create more training data by modifying existing data
- **Examples:**
  - Flipping, rotating, cropping
  - Color jittering
  - Adding noise
- **Why:** Helps model generalize better

Data Augmentation in Practice

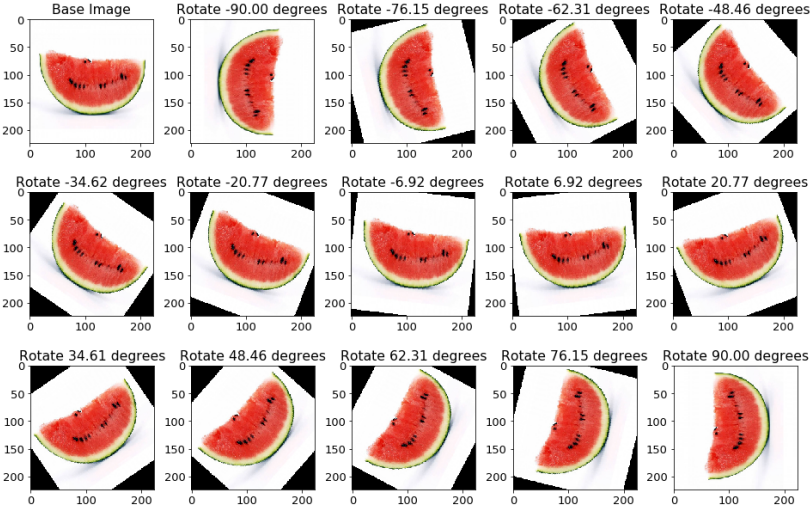
- **Popular Libraries:**
  - `torchvision.transforms`
  - `imgaug`
  - `albumentations`
- **Combine augmentations:** e.g., flip + rotate + brightness
- **Apply randomly during training**
- **Goal:** Make your data more realistic and diverse

Data Augmentation in Practice (cont.)

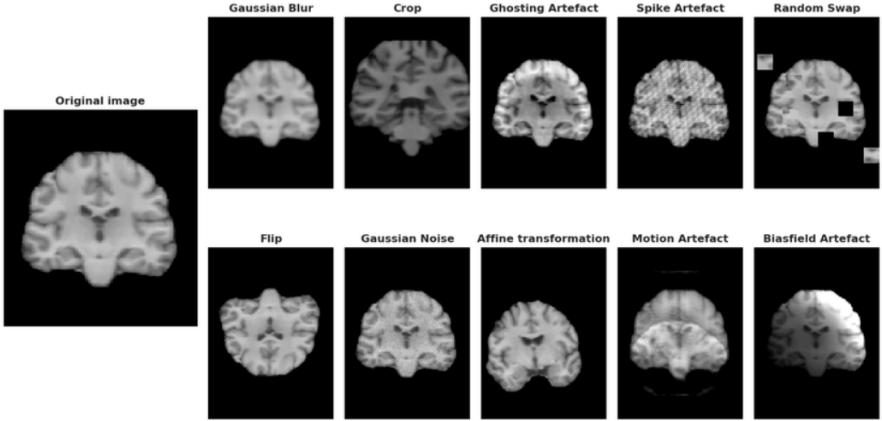
- Example of data augmentation techniques applied to images



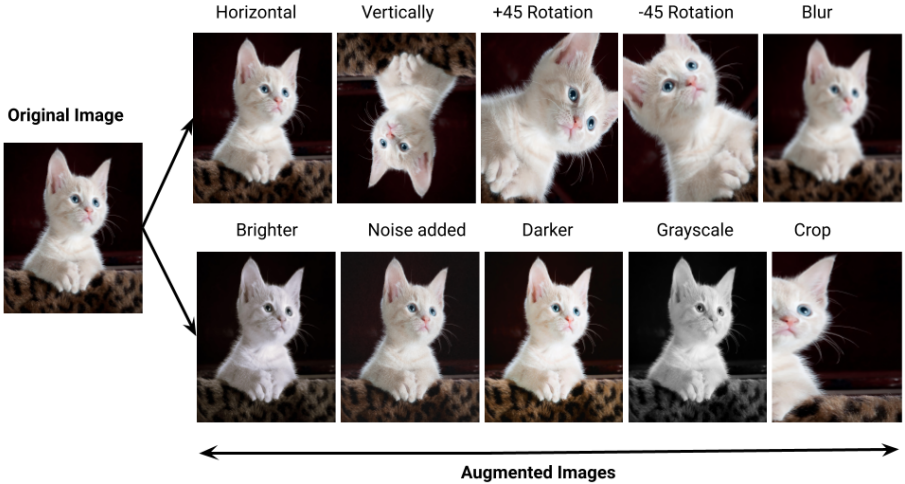
Data Augmentation in Practice (cont.)



Data Augmentation in Practice (cont.)



Data Augmentation in Practice (cont.)



CNN: **Normalization (Batch Norm)**

Batch Normalization

- Consider a single layer $y = Wx$
- The following could lead to tough optimazation
  - Inputs x are not centered around zero (need large bias)
  - Inputs x have different scaling per element (entries in W will need to vary a lot)

Batch Normalization (cont.)

- Consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

- **Problem:** What if zero-mean, unit variance is too hard of a constraint?

Batch Normalization (cont.)

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization (cont.)

Estimates depend on minibatch; can't do this at test-time!

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization (cont.)

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel var, shape is D

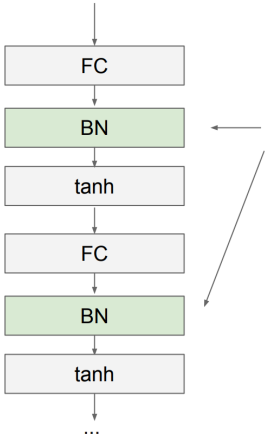
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization (cont.)



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization (cont.)

Batch Normalization for **fully-connected** networks

$x: N \times D$

Normalize

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

$x: N \times C \times H \times W$

Normalize

$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization (cont.)

- Advantages:
  - Makes deep networks much easier to train!
  - Improves gradient flow
  - Allows higher learning rates, faster convergence
  - Networks become more robust to initialization
  - Acts as regularization during training
  - Zero overhead at test-time: can be fused with conv!
- Disadvantages:
  - Behaves differently during training and testing: this is a very common source of bugs!

Normalization: Summary

What:

- Normalize layer inputs over mini-batch, then scale shift.

Benefits:

- Faster training,
- Allows higher learning rates,
- Reduces sensitivity to initialization,
- Acts as a regularizer.

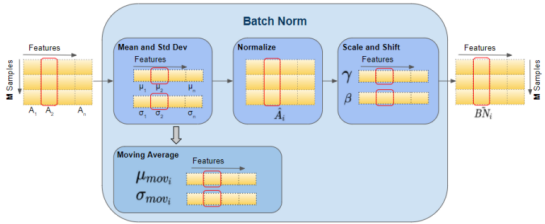
Listing 1: Code snippet (PyTorch)

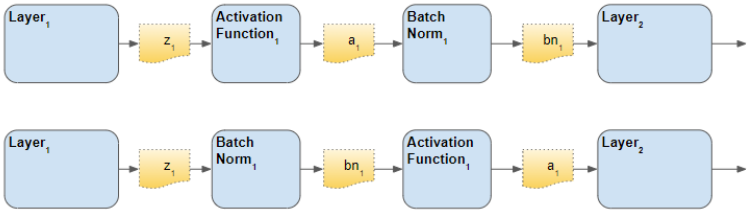
```
import torch.nn as nn

bn = nn.BatchNorm2d(16)
x = bn(x)
```

Note:

- BatchNorm is typically used after convolutional layers and before activation functions.
- BatchNorm normalizes activations per-batch, then scales/shifts them.

Normalization: Summary





Order of placement of Batch Norm layer

More on Batch Norm from Towards Data Science

CNN: **Regularization (Dropout)**

Regularization (Dropout)

What:

- Randomly drop units with probability p during training to prevent co-adaptation.

Where:

- Often after FC layers, sometimes after conv layers.

Effect on images:

- Regularization does not alter the input images but modifies the training process to improve generalization.

```
import torch.nn as nn

drop = nn.Dropout(p=0.5)
x = drop(x)
```

CNN: **Loss Functions & Optimizers**

Loss Functions & Optimizers

Loss:

- Loss functions measure how well the model's predictions match the true labels.
- Common loss functions include:
  - Cross-Entropy Loss (for classification)
  - Mean Squared Error (for regression)
- The choice of loss function depends on the task at hand.

Optimizers:

- Optimizers update the model's parameters based on the gradients computed during backpropagation.
- Common optimizers include:
  - Stochastic Gradient Descent (SGD)
  - Adam
  - RMSprop
- The choice of optimizer can significantly affect the training speed and convergence.

Loss Functions & Optimizers (cont.)


```
import torch.nn as nn
import torch

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss = criterion(preds, labels)
loss.backward()
optimizer.step()
```

CNN: **Transfer Learning**

What is Transfer Learning?

- **Transfer Learning** is the process of reusing a model trained on one task (e.g., ImageNet classification) for a different but related task (e.g., medical image analysis).
- **Why use it?**
  - Saves training time and computational resources.
  - Achieves good performance with limited data.
- **Types:**
  - **Feature extraction:** Freeze convolutional layers and only train the final classifier.
  - **Fine-tuning:** Unfreeze some or all layers and retrain on the new task.

Transfer Learning Example

- **Example:** Load a pretrained ResNet50 from PyTorch.
- Replace the last (classification) layer to match your number of classes.
- Optionally unfreeze the last few layers and retrain (fine-tuning).

Use cases:

- Medical imaging
- Satellite imagery
- Small business datasets with limited data

PyTorch Transfer Learning Code

```
import torch
import torchvision.models as models
import torch.nn as nn

# Load pretrained ResNet50
model = models.resnet50(pretrained=True)

# Freeze all layers
for param in model.parameters():
    param.requires_grad = False

# Replace the final layer for 3 classes
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 3)

# Optionally unfreeze last few layers for fine-tuning
for param in list(model.parameters())[-10:]:
    param.requires_grad = True
```

Benefits of Transfer Learning

- **Faster Training:** Pretrained models converge faster than training from scratch.
- **Better Performance:** Leverages learned features from large datasets.
- **Reduced Overfitting:** Especially useful when training data is limited.
- **Flexibility:** Can adapt to various tasks with minimal changes.

Considerations:

- Ensure the pretrained model is suitable for your task.
- Fine-tuning requires careful selection of layers to unfreeze.
- Monitor for overfitting, especially with small datasets.

Putting it All Together

- **Pick a solid architecture:** Start with proven models like ResNet, MobileNet, etc.
- **Use data augmentation:** Enrich your dataset with transformations to improve generalization.
- **Start with a pretrained model:** Leverage transfer learning for faster convergence and better performance.
- **Normalize activations:** Use Batch Normalization to stabilize and speed up training.
- **Regularize:** Apply Dropout and Early Stopping to prevent overfitting.

CNN: **Limitations and Future Directions**

Limitations of CNNs

- **Understanding global context:** CNNs are inherently local due to convolutional filters, making it hard to capture long-range dependencies.
- **Variations outside training data:** Performance drops significantly when faced with data distributions not seen during training.
- **Transfer learning limitations:** Pre-trained CNNs do not always generalize well to new tasks or domains.
- **BatchNorm sensitivity:** Batch Normalization effectiveness depends on batch size, which can be problematic for small-batch or online learning.
- **Compute constraints:** Deploying CNNs on mobile or edge devices remains challenging due to high computational and memory requirements.

Future Directions

- **Vision Transformers (ViTs):** Leveraging self-attention mechanisms to capture global context and long-range dependencies.
- **Self-supervised learning:** Reducing reliance on labeled data by learning useful representations from unlabeled data.
- **More efficient models:** Architectures like EfficientNet optimize accuracy and efficiency for deployment on resource-constrained devices.
- **Better augmentation:** Automated data augmentation techniques (e.g., AutoAugment, RandAugment) improve generalization.
- **Low-power inference:** Techniques such as quantization and pruning enable CNNs to run efficiently on edge and mobile devices.

CNN: **References & Resources**

Further Reading

Books:

- **Deep Learning** by Ian Goodfellow, Yoshua Bengio, and Aaron Courville
- **Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow** by Aurélien Géron
- **Pattern Recognition and Machine Learning** by Christopher Bishop
- **Deep Learning for Computer Vision with Python** by Adrian Rosebrock

Further Reading (cont.)

Online Courses:

- **Deep Learning Specialization** by Andrew Ng (Coursera)
- **Convolutional Neural Networks for Visual Recognition** by Fei-Fei Li (Stanford University)
- **Practical Deep Learning for Coders** by Jeremy Howard (fast.ai)
- **Deep Learning with PyTorch** by Facebook AI Research

Further Reading (cont.)

Research Papers:

- **ImageNet Classification with Deep Convolutional Neural Networks** by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton
- **Very Deep Convolutional Networks for Large-Scale Image Recognition** by K. Simonyan and A. Zisserman
- **Deep Residual Learning for Image Recognition** by Kaiming He et al.
- **Densely Connected Convolutional Networks** by Gao Huang et al.

Further Reading (cont.)

Websites and Blogs:

- **Towards Data Science** (Medium)
- **Distill.pub** (Research and visualization)
- **Kaggle** (Datasets and competitions)
- **Papers with Code** (Research papers with code implementations)

Further Reading (cont.)

YouTube Channels:

- **3Blue1Brown** (Mathematics and deep learning)
- **Two Minute Papers** (Research summaries)
- **Sentdex** (Python programming and machine learning)
- **StatQuest with Josh Starmer** (Statistics and machine learning)

Further Reading (cont.)

GitHub Repositories:

- **TensorFlow Models** (TensorFlow implementations)
- **PyTorch Examples** (PyTorch implementations)
- **fastai** (High-level library for deep learning)
- **Keras** (High-level neural networks API)

Further Reading (cont.)

Conferences:

- **NeurIPS** (Neural Information Processing Systems)
- **CVPR** (Computer Vision and Pattern Recognition)
- **ICML** (International Conference on Machine Learning)
- **ICLR** (International Conference on Learning Representations)

Credits

Dr. Prashant Aparajeya

Computer Vision Scientist — Director(AISimply Ltd)

p.aparajeya@aisimply.uk

This project benefited from external collaboration, and we acknowledge their contribution with gratitude.