Machine Learning for Computer Vision

Last update: 26 September 2024

Academic Year 2024 – 2025
Alma Mater Studiorum · University of Bologna

Contents

[1] Excerpt from IPCV2
[2] Excerpt from IPCV2

1 Optimizers

1.1 Stochastic gradient descent with mini-batches

SGD

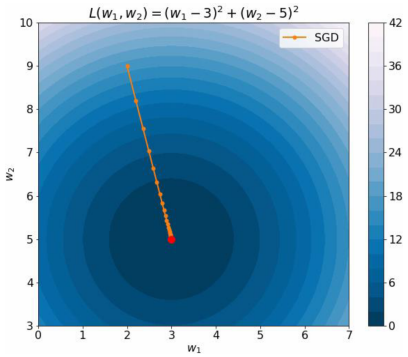
Stochastic gradient descent (SGD) Gradient descent based on a noisy approximation of the gradient computed on mini-batches of B data samples.

An epoch e of SGD with mini-batches of size B does the following:

1. Shuffle the training data $\boldsymbol{D}^{\text{train}}$.
2. For $u = 0, \dots, U$, with $U = \lceil \frac{N}{B} \rceil$:
   a) Classify the examples $\boldsymbol{X}^{(u)} = \{\mathbf{x}^{(Bu)}, \dots, \mathbf{x}^{(B(u+1)-1)}\}$ to obtain the predictions $\hat{Y}^{(u)} = f(\mathbf{X}^{(u)}; \boldsymbol{\theta}^{(e*U+u)})$ and the loss $\mathcal{L}\big(\boldsymbol{\theta}^{(e*U+u)}, (\boldsymbol{X}^{(u)}, \hat{Y}^{(u)})\big)$.
   b) Compute the gradient $\nabla\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}\big(\boldsymbol{\theta}^{(e*U+u)}, (\boldsymbol{X}^{(u)}, \hat{Y}^{(u)})\big)$.
   c) Update the parameters $\boldsymbol{\theta}^{(e*U+u+1)} = \boldsymbol{\theta}^{(e*U+u)} - \texttt{lr} \cdot \nabla\mathcal{L}$.

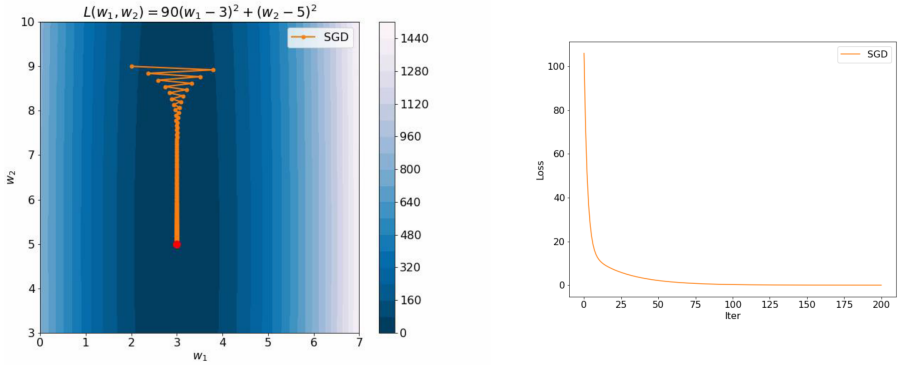
SGD spheres

Remark (Spheres)**.** GD/SGD works better on convex functions (e.g., paraboloids) as there are no preferred directions to reach a minimum. Moreover, faster convergence can be obtained by using a higher learning rate.



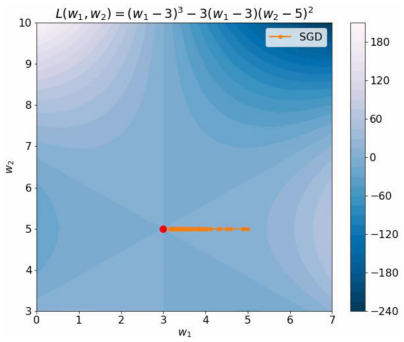
SGD canyons

Remark (Canyons)**.** A function has a canyon shape if it grows faster in some directions. The trajectory of SGD oscillates in a canyon (the steep area) and a smaller learning rate is required to reach convergence. Note that, even though there are oscillations, the loss alone decreases and is unable to show the oscillating behavior.



Remark (Local minima)**.** GD/SGD converges to a critical point. Therefore, it might end up in a saddle point or local minima.

SGD local minima



1.2 Second-order methods

Methods that also consider the second-order derivatives when determining the step.

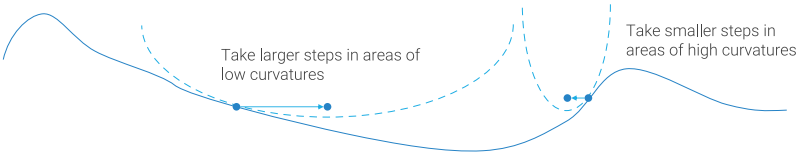
Newton's method Second-order method for the 1D case based on the Taylor expansion:

Newton's method

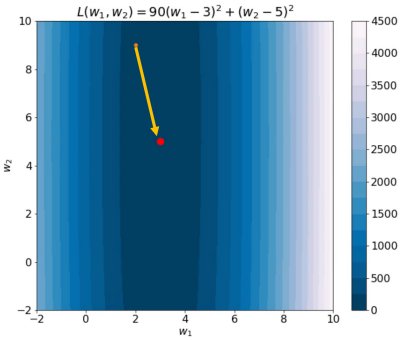
$$f(x_t + \Delta x) \approx f(x_t) + f'(x_t)\Delta x + \frac{1}{2} f''(x_t)\Delta x^2$$

which can be seen as a paraboloid over the variable Δx.

Given a function f and a point x_t, the method fits a paraboloid at x_t with the same slope and curvature at $f(x_t)$. The update is determined as the step required to reach the minimum of the paraboloid from x_t. It can be shown that this step is $-\frac{f'(x_t)}{f''(x_t)}$.



Remark. For quadratic functions, second-order methods converge in one step.



General second-order method For a generic multivariate non-quadratic function, the update is:

General second-order method

$$-\texttt{lr} \cdot \boldsymbol{H}_f^{-1}(\mathbf{x}_t)\nabla f(\mathbf{x}_t)$$

where $\boldsymbol{H}_f$ is the Hessian matrix.

Remark. Given k variables, $\boldsymbol{H}$ requires $O(k^2)$ memory. Moreover, inverting a matrix has time complexity $O(k^3)$. Therefore, in practice second-order methods are not applicable for large models.

1.3 Momentum

Standard momentum Add a velocity term $v^{(t)}$ to account for past gradient updates: Standard momentum

$$v^{(t+1)} = \mu v^{(t)} - \texttt{lr}\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})$$
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + v^{(t+1)}$$

where $\mu \in [0, 1[$ is the momentum coefficient.

In other words, $v^{(t+1)}$ represents a weighted average of the updates steps done up until time t.

Remark. Momentum helps to counteract a poor conditioning of the Hessian matrix when working with canyons.

Remark. Momentum helps to reduce the effect of variance of the approximated gradients (i.e., acts as a low-pass filter).

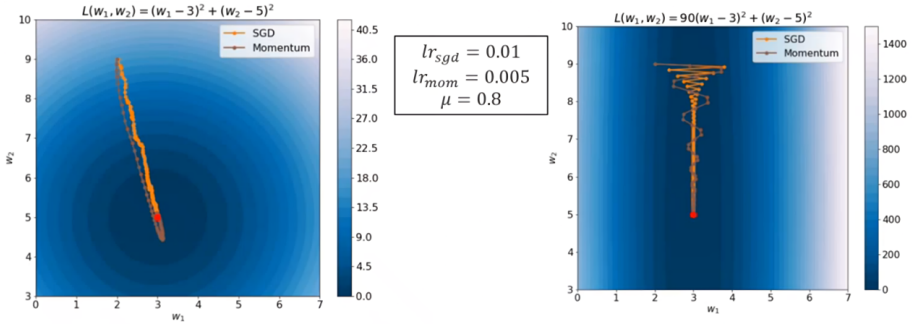


Figure 1.1: Plain SGD vs momentum SGD in a sphere and a canyon. In both cases, momentum converges before SGD.

Nesterov momentum Variation of the standard momentum that computes the gradient step considering the velocity term: Nesterov momentum

$$v^{(t+1)} = \mu v^{(t)} - \texttt{lr}\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)} + \mu v^{(t)})$$
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + v^{(t+1)}$$

Remark. The key idea is that, once $\mu v^{(t)}$ is summed to $\boldsymbol{\theta}^{(t)}$, the gradient computed at $\boldsymbol{\theta}^{(t)}$ is obsolete as $\boldsymbol{\theta}^{(t)}$ has been partially updated.

Remark. In practice, there are methods to formulate Nesterov momentum without the need of computing the gradient at $\boldsymbol{\theta}^{(t)} + \mu v^{(t)}$.

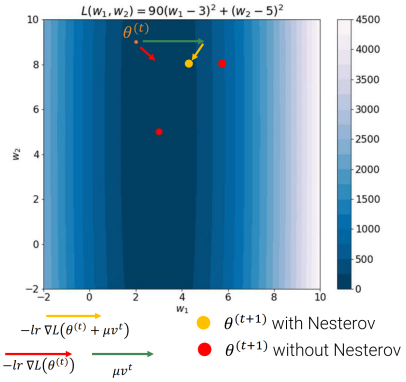


Figure 1.2: Visualization of the step in Nesterov momentum

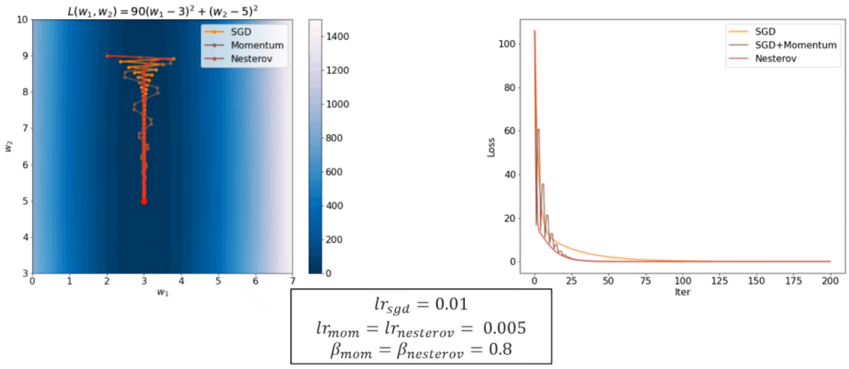


Figure 1.3: Plain SGD vs standard momentum vs Nesterov momentum

1.4 Adaptive learning rates methods

Adaptive learning rates Methods to define per-parameter adaptive learning rates.

Adaptive learning rates

Ideally, assuming that the changes in the curvature of the loss are axis-aligned (i.e., the parameters are independent), it is reasonable to obtain a faster convergence by:

- Reducing the learning rate along the dimension where the gradient is large.
- Increasing the learning rate along the dimension where the gradient is small.

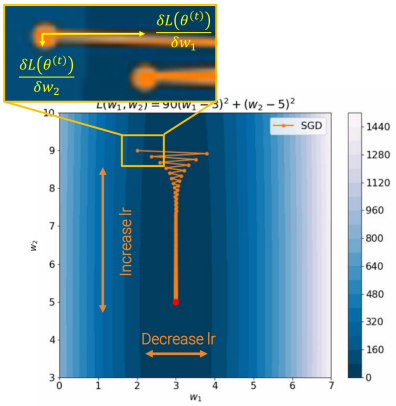


Figure 1.4: Loss where the w_1 parameter has a larger gradient, while w_2 has a smaller gradient

Remark. As the landscape of a high-dimensional loss cannot be seen, automatic methods to adjust the learning rates must be used.

1.4.1 AdaGrad

Adaptive gradient (AdaGrad) Each entry of the gradient is rescaled by the inverse of the history of its squared values: Adaptive gradient (AdaGrad)

$$\mathbb{R}^{N\times 1} \ni \mathbf{s}^{(t+1)} = \mathbf{s}^{(t)} + \nabla\mathcal{L}(\boldsymbol{\theta}^{(t)}) \odot \nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})$$
$$\mathbb{R}^{N\times 1} \ni \boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\texttt{lr}}{\sqrt{\mathbf{s}^{(t+1)}} + \varepsilon} \odot \nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})$$

where:

- $\odot$ is the element-wise product.
- Division and square root are element-wise.
- ε is a small constant.

Remark. By how it is defined, $\mathbf{s}^{(t)}$ is monotonically increasing which might reduce the learning rate too early when the minimum is still far away.

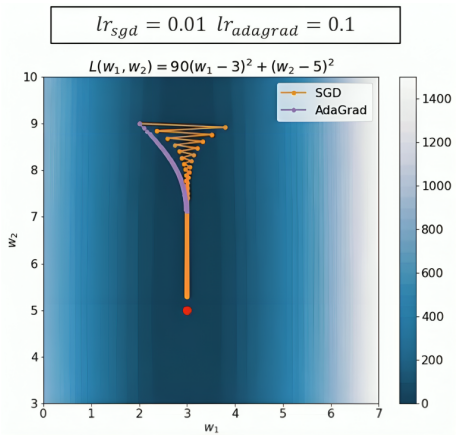


Figure 1.5: SGD vs AdaGrad. AdaGrad stops before getting close to the minimum.

1.4.2 RMSProp

RMSProp Modified version of AdaGrad that down-weighs the gradient history $s^{(t)}$: RMSProp

$$\mathbf{s}^{(t+1)} = \beta\mathbf{s}^{(t)} + (1-\beta)\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)}) \odot \nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})$$
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\texttt{lr}}{\sqrt{\mathbf{s}^{(t+1)}} + \varepsilon} \odot \nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})$$

where $\beta \in [0, 1]$ (typically 0.9 or higher) makes $s^{(t)}$ an exponential moving average.

Remark. RMSProp is faster than SGD at the beginning and slows down reaching similar performances as SGD.

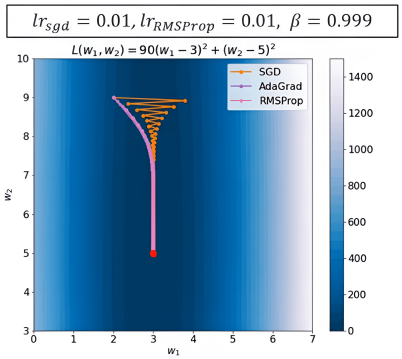


Figure 1.6: SGD vs AdaGrad vs RMSProp

1.4.3 Adam

Adaptive moments (Adam) Extends RMSProp by also considering a running average for the gradients:

Adaptive moments (Adam)

$$\mathbf{g}^{(t+1)} = \beta_1 \mathbf{g}^{(t)} + (1 - \beta_1)\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})$$
$$\mathbf{s}^{(t+1)} = \beta_2 \mathbf{s}^{(t)} + (1 - \beta_2)\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)}) \odot \nabla\mathcal{L}(\boldsymbol{\theta}^{(t)})$$

where $\beta_1, \beta_2 \in [0, 1]$ (typically $\beta_1 = 0.9$ and $\beta_2 = 0.999$).

Moreover, as $\mathbf{g}^{(0)} = 0$, $\mathbf{s}^{(0)} = 0$, and β_1, β_2 are typically large (i.e., past history weighs more), Adam starts by taking small steps (e.g., $\mathbf{g}^{(1)} = (1 - \beta_1)\nabla\mathcal{L}(\boldsymbol{\theta}^{(0)})$ is simply rescaling the gradient for no reason). To cope with this, a debiased formulation of $\mathbf{g}$ and $\mathbf{s}$ is used:

$$\mathbf{g}^{(t)}_{\text{debiased}} = \frac{g^{(t+1)}}{1 - \beta_1^{t+1}} \qquad \mathbf{s}^{(t)}_{\text{debiased}} = \frac{s^{(t+1)}}{1 - \beta_2^{t+1}}$$

where the denominator $(1 - \beta_i^{t+1}) \to 1$ for increasing values of t.

Finally, the update is defined as:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\texttt{lr}}{\sqrt{s^{(t)}_{\text{debiased}}} + \varepsilon} \odot g^{(t)}_{\text{debiased}}$$

Remark. It can be shown that $\frac{g^{(t)}_{\text{debiased}}}{\sqrt{s^{(t)}_{\text{debiased}}}}$ has a bounded domain, making it more controlled than RMSProp.

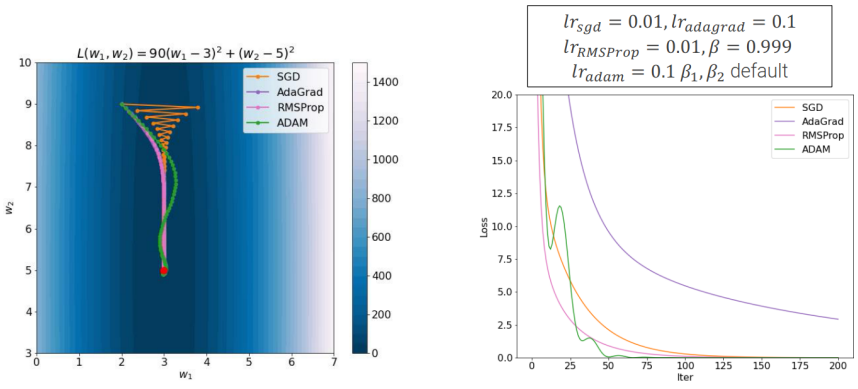


Figure 1.7: SGD vs AdaGrad vs RMSProp vs Adam

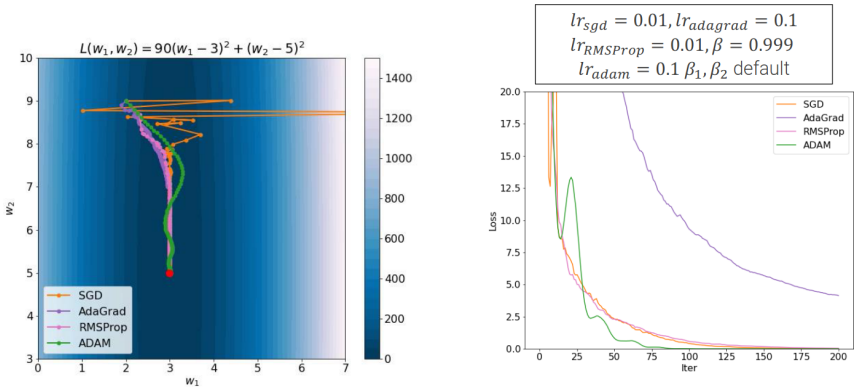
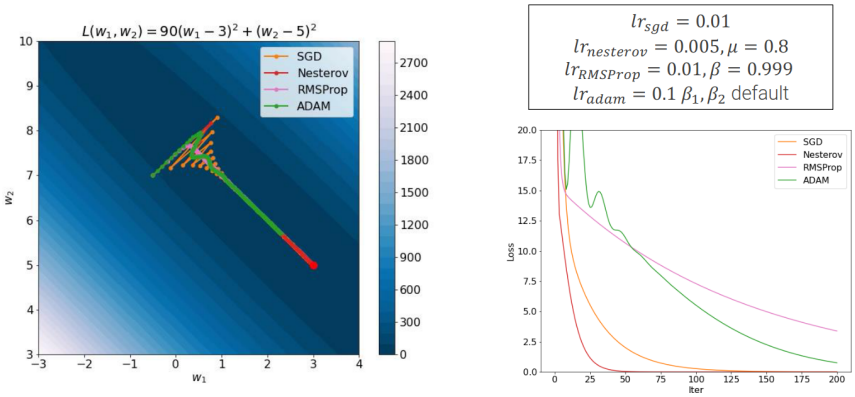


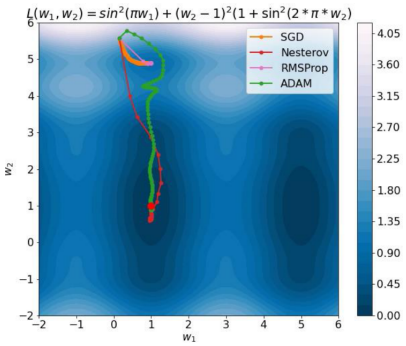
Figure 1.8: SGD vs AdaGrad vs RMSProp vs Adam with a smaller batch size

Remark. Adam is based on the assumption of unrelated parameters (i.e., axis-aligned). If this does not actually hold, it might be slower to converge.



Remark. Empirically, in computer vision Nesterov momentum (properly tuned) works better than Adam.

Remark. Momentum based approaches tend to prefer large basins. Intuitively, by accumulating momentum, it is possible to "escape" small basins.



1.4.4 AdamW

Adam with weight decay (AdamW) Modification on the gradient update of Adam to include weight decay:

Adam with weight decay (AdamW)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\texttt{lr}}{\sqrt{s^{(t)}_{\text{debiased}}} + \varepsilon} \odot g^{(t)}_{\text{debiased}} - \lambda \boldsymbol{\theta}^{(t)}$$

Remark. Differently from SGD, L2 regularization on Adam is not equivalent to applying weight decay. In fact, by definition, the regularization term is applied to the gradient and not on the update step:

$$\nabla_{\text{actual}}\mathcal{L}(\boldsymbol{\theta}^{(t)}) = \nabla\mathcal{L}(\boldsymbol{\theta}^{(t)}) + \lambda\boldsymbol{\theta}^{(t)}$$

where $\nabla_{\text{actual}}\mathcal{L}(\boldsymbol{\theta}^{(t)})$ is the actual gradient used to compute the running averages $\mathbf{g}$ and $\mathbf{s}$.

2 Architectures

2.1 Inception-v1 (GoogLeNet)[1]

Network that aims to optimize computing resources (i.e., small amount of parameters and FLOPs). Inception-v1 (GoogLeNet)

Stem layers Down-sample the image from a shape of 224 to 28. As in ZFNet, multiple layers are used (5) and the largest convolution is of shape 7×7 with stride 2.

Inception module Main component of Inception-v1 that computes multiple convolutions on the input. Inception module

Given the input activation, the output is the concatenation of:

- A 1×1 (stride 1) and a 5×5 (stride 1, padding 2) convolution.
- A 1×1 (stride 1) and a 3×3 (stride 1 and padding 1) convolution.
- A 1×1 (stride 1 and padding 0) convolution.
- A 1×1 (stride 1) convolution and a 3×3 (stride 1 and padding 1) max-pooling.

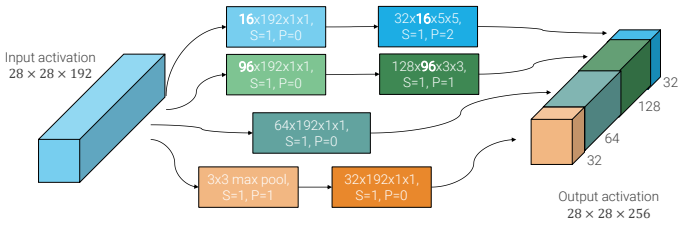


Figure 2.1: Inception module on the output of the stem layers

> **Remark.** The multiple convolutions of an inception module can be seen as decision components.

Auxiliary `softmax` Intermediate `softmax`s are used to ensure that hidden features are good enough. They also act as regularizers. During inference, they are discarded.

Global average pooling classifier Instead of flattening between the convolutional and fully connected layers, global average pooling is used to reduce the number of parameters. Global average pooling classifier

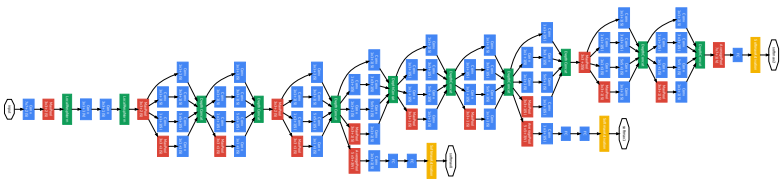


Figure 2.2: Architecture of Inception-v1

[1] Excerpt from IPCV2

2.2 Residual networks[2]

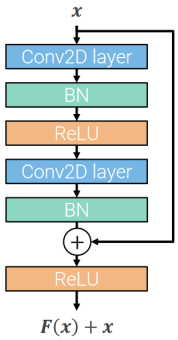
Standard residual block

Standard residual block Block that allows to easily learn the identity function through a skip connection. The output of a residual block with input x and a series of convolutional layers F is:

$$F(x;\boldsymbol{\theta}) + x$$

Skip connection

Skip connection Connection that skips a certain number of layers (e.g. 2 convolutional blocks).



Remark. Training starts with small weights so that the network starts as the identity function. Updates can be seen as perturbations of the identity function.

Remark. Batch normalization is heavily used.

Remark. Skip connections are applied before the activation function (ReLU) as otherwise it would be summed to all positive values making the perturbation of the identity function less effective.

2.2.1 ResNet

ResNet-18

VGG-inspired network with residual blocks. It has the following properties:

- A stage is composed of residual blocks.
- A residual block is composed of two 3×3 convolutions followed by batch normalization.
- The first residual block of each stage halves the spatial dimension and doubles the number of channels (there is no pooling).

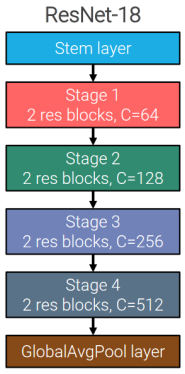


Figure 2.3: Architecture of ResNet-18

Bottleneck residual network

Bottleneck residual network Variant of residual blocks that uses more layers with approximately the same number of parameters and FLOPs of the standard residual block. Instead of using two 3×3 convolutions, bottleneck residual network has the following structure:

- 1×1 convolution to compress the channels of the input by an order of 4 (and the spatial dimension by 2 if it is the first block of a stage, as in normal ResNet).
- 3×3 convolution.
- 1×1 convolution to match the shape of the skip connection.

[2]Excerpt from IPCV2

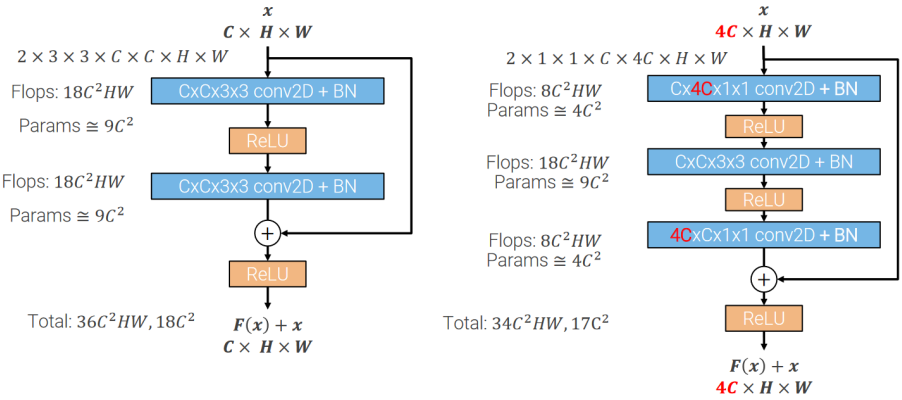


Figure 2.4: Standard residual block (left) and bottleneck block (right)

2.2.2 Inception-ResNet-v4

Network with bottleneck-block-inspired inception modules.

Inception-ResNet-A Three 1×1 convolutions are used to compress the input channels. Each of them leads to a different path: Inception-ResNet-A

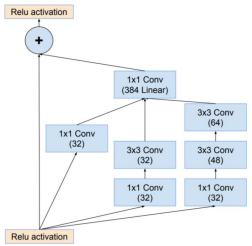
- Directly to the final concatenation.
- To a 3×3 convolution.
- To two 3×3 convolutions (i.e. a factorized 5×5 convolution).

The final concatenation is passed through a 1×1 convolution to match the skip connection shape.

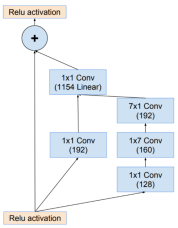
Inception-ResNet-B Three 1×1 convolutions are used to compress the input channels. Each of them leads to: Inception-ResNet-B

- Directly to the final concatenation.
- A 1×7 and 7×1 convolutions (i.e. a factorized 7×7 convolution).

The final concatenation is passed through a 1×1 convolution to match the skip connection shape.



Inception-resnet A, used for fine-scale activations (35x35)



Inception-resnet B, used for mid-scale activations (17x17)

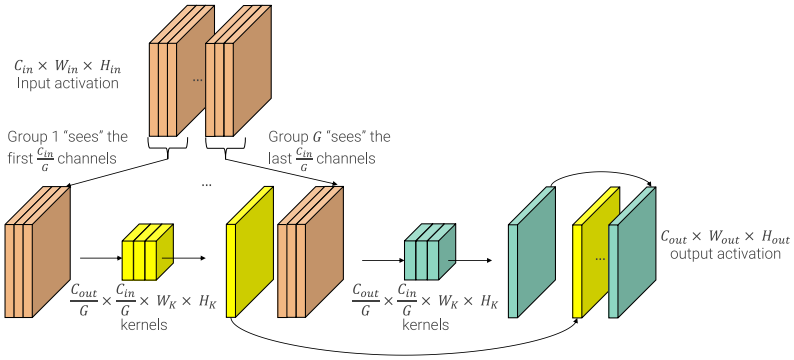
2.3 ResNeXt

Remark. Inception and Inception-ResNet modules are multi-branch architectures and can be interpreted as a split-transform-merge paradigm. Moreover, their architectures have been specifically "hand" designed.

Grouped convolution Given: Grouped convolution

- The input activation of shape $C_{\text{in}} \times W_{\text{in}} \times H_{\text{in}}$,
- The desired number of output channels C_{out},
- The number of groups G,

a grouped convolution splits the input into G chunks of $\frac{C_{\text{in}}}{G}$ channels and processes each with a dedicated set of kernels of shape $\frac{C_{\text{out}}}{G} \times \frac{C_{\text{in}}}{G} \times W_K \times H_K$. The output activation is obtained by stacking the outputs of each group.



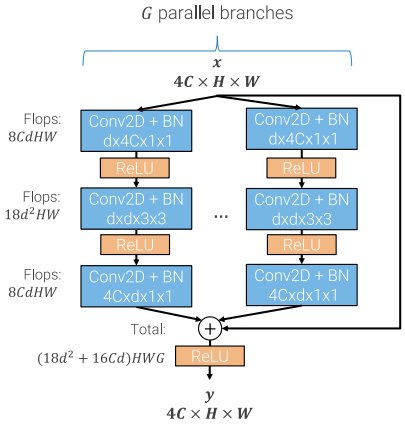
By processing the input in smaller chunks, there are the following gains:

- The number of parameters is G times less.
- The number of FLOPs is G times less.

Remark. Grouped convolutions are trivially less expressive than convolving on the full input activation. However, as convolutions are expected to build a hierarchy of features, it is reasonable to process the input in chunks as, probably, not all of it is needed.

2.3.1 Architecture

ResNetXt block Given the number of branches G and the number of intermediate channels d, a ResNeXt block decomposes a bottleneck residual block into G parallel branches that are summed out at the end. ResNetXt block



Remark. The branching in a ResNeXt block should not be confused with grouped convolutions.

Remark. Parametrizing G and d allows obtaining configurations that are FLOP-wise comparable with the original ResNet by fixing G and solving a second-order equation over d.

Equivalent formulation Given an input activation $\mathbf{x}$ of shape $4C \times H \times W$, each layer of the ResNeXt block can be reformulated as follows:

Second 1×1 convolution Without loss of generality, consider a ResNeXt block with $G = 2$ branches.

The output $\mathbf{y}_k$ at each channel $k = 1, \ldots, 4C$ is obtained as:

$$\mathbf{y}_k = \mathbf{y}_k^{(1)} + \mathbf{y}_k^{(2)} + \mathbf{x}_k$$

where the output $\mathbf{y}_k^{(b)}$ of a branch b is computed as:

$$\begin{aligned}\mathbf{y}_k^{(b)}(j,i) &= \left[\mathbf{w}^{(b)} * \mathbf{a}^{(b)}\right]_k (j,i) \\ &= \mathbf{w}_k^{(b)} \cdot \mathbf{a}^{(b)}(j,i) \\ &= \mathbf{w}_k^{(b)}(1)\mathbf{a}^{(b)}(j,i,1) + \cdots + \mathbf{w}_k^{(b)}(d)\mathbf{a}^{(b)}(j,i,d)\end{aligned}$$

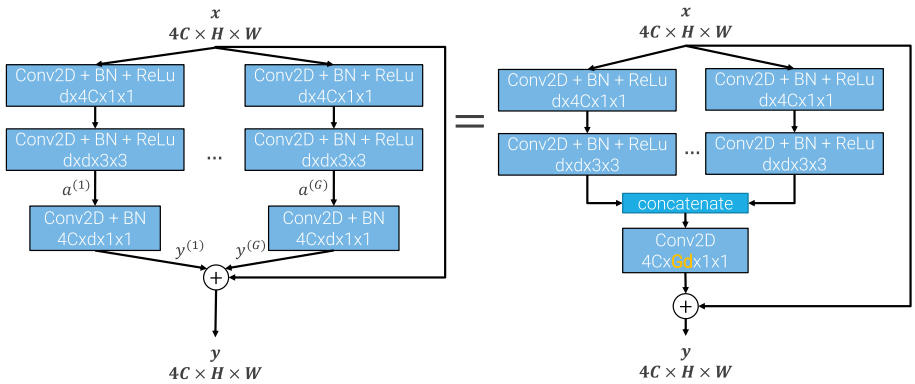
where:

- $*$ represents a convolution,
- $\mathbf{a}^{(b)}$ is the input activation with d channels from the previous layer.
- $\mathbf{w}^{(b)}$ is the convolutional kernel. $\mathbf{w}_k^{(b)} \in \mathbb{R}^d$ is the kernel used to obtain the k-th output channel.

By putting everything together:

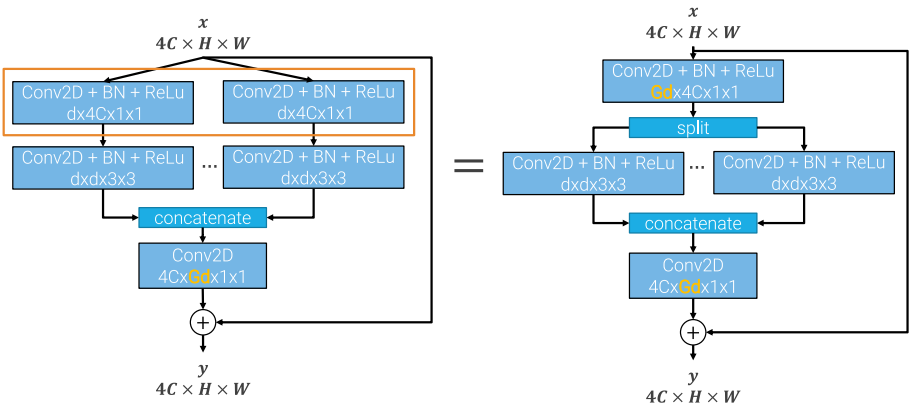
$$\begin{aligned}\mathbf{y}_k(j,i) &= \mathbf{w}_k^{(1)} \cdot \mathbf{a}^{(1)}(j,i) + \mathbf{w}_k^{(2)} \cdot \mathbf{a}^{(2)}(j,i) + \mathbf{x}_k \\ &= \underbrace{\left[\mathbf{w}_k^{(1)} \mathbf{w}_k^{(2)}\right]}_{\text{by stacking, this is a } 1\times 1 \text{ convolution with } 2d \text{ channels}} \cdot \underbrace{\left[\mathbf{a}^{(1)}(j,i)\mathbf{a}^{(2)}(j,i)\right]}_{\text{by stacking depth-wise, this is an activation with } 2d \text{ channels}} + \mathbf{x}_k\end{aligned}$$

Therefore, the last ResNeXt layer with G branches is equivalent to a single convolution with Gd input channels that processes the concatenation of the activations of the previous layer.

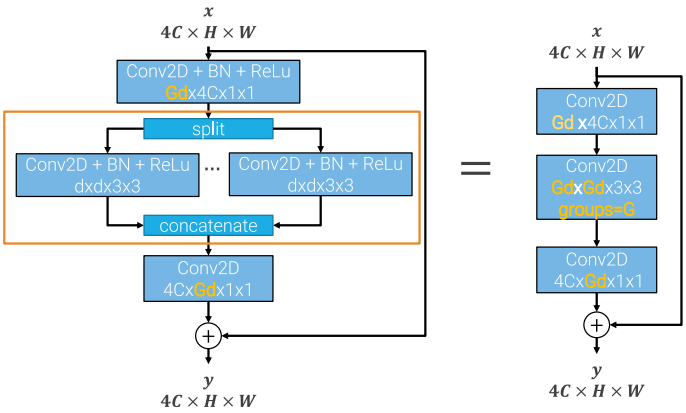


First 1×1 convolution The G 1×1 convolutions at the first layer of ResNeXt all process the same input $\mathbf{x}$. Trivially, this can also be represented using

a single 1×1 convolution with G times more output channels that can be split afterwards.



3×3 **convolution** By putting together the previous two equivalences, the middle layer has the same definition of a grouped convolution with G groups. Therefore, it can be seen as a single grouped convolution with G groups and Gd input and output channels.



| **Remark.** Therefore, a ResNeXt block is similar to a bottleneck block.

2.3.2 Properties

The following holds:

- It has been empirically seen that, with the same FLOPs, it is better to have more groups (i.e., wider activations).