Machine Learning for Computer Vision

Last update: 10 October 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

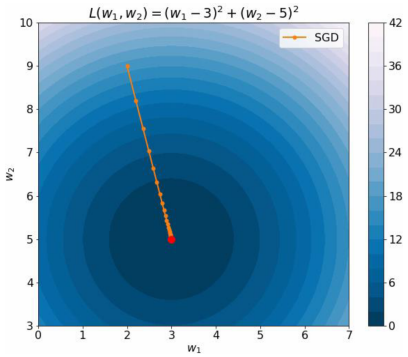
Stochastic gradient descent (SGD) Gradient descent based on a noisy approximation of the gradient computed on mini-batches of B data samples.

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 $\boldsymbol{\theta}^{(e*U+u+1)} = \boldsymbol{\theta}^{(e*U+u)} - \texttt{lr} \cdot \nabla\mathcal{L}$.

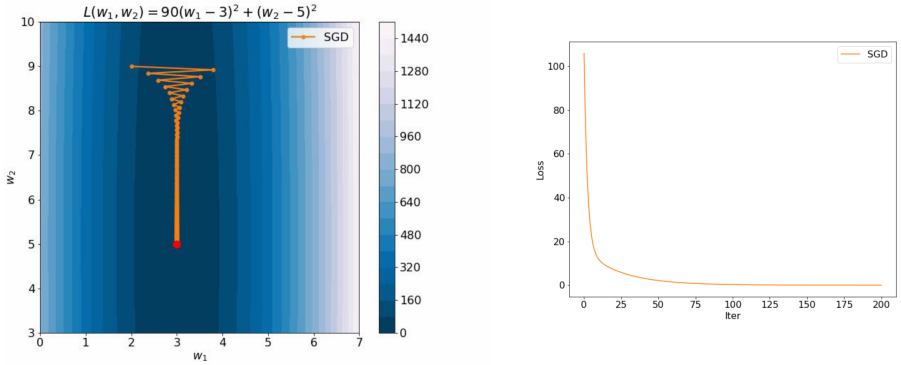
SGD spheres

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.



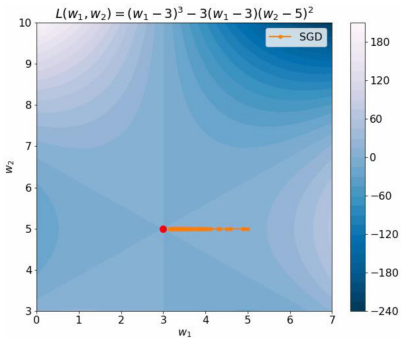
SGD canyons

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.

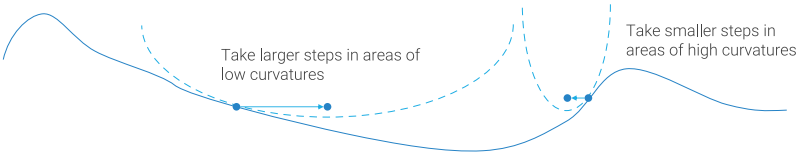
Newton's method Second-order method for the 1D case based on the Taylor expansion:

Newton's method

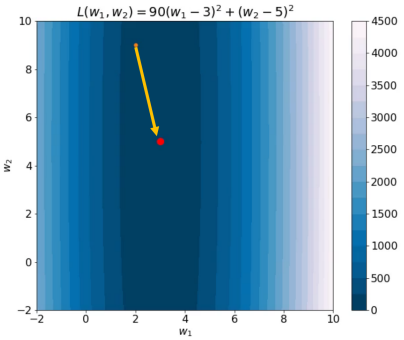
$$f(x_t + \Delta x) \approx f(x_t) + f'(x_t)\Delta x + \frac{1}{2} f''(x_t)\Delta x^2$$

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).

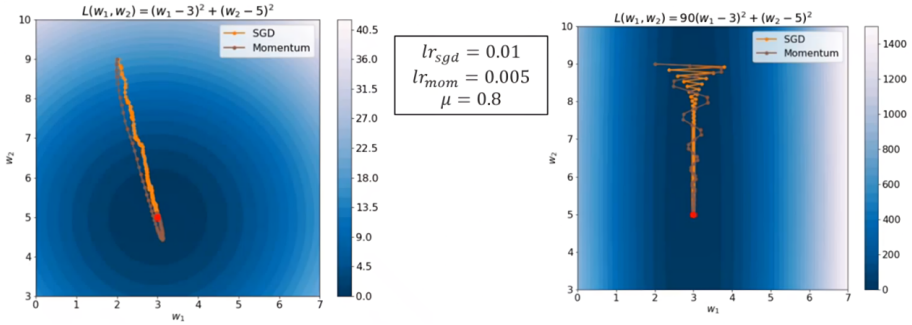


Figure 1.1: Plain SGD vs momentum SGD in a sphere and a canyon. In both cases, momentum converges before SGD.

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 $\boldsymbol{\theta}^{(t)} + \mu v^{(t)}$.

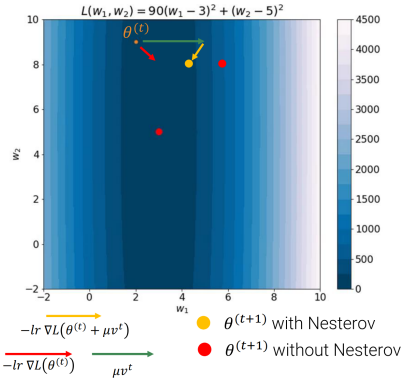


Figure 1.2: Visualization of the step in Nesterov momentum

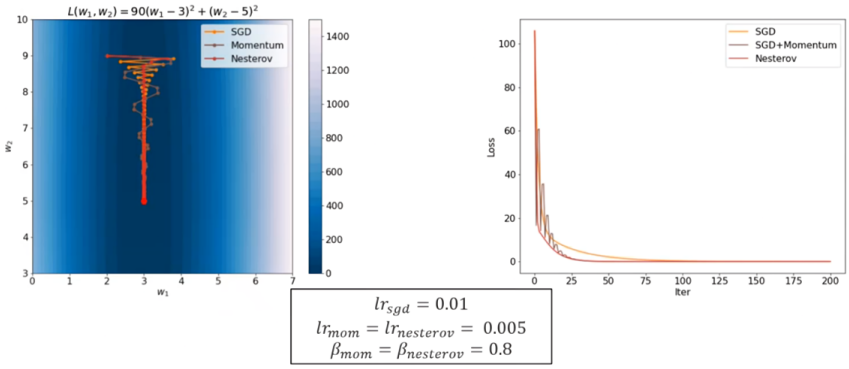


Figure 1.3: Plain SGD vs standard momentum vs Nesterov momentum

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.

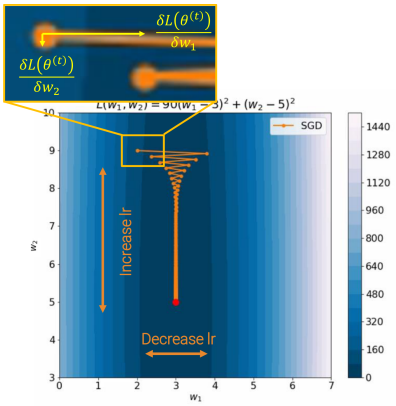


Figure 1.4: Loss where the w_1 parameter has a larger gradient, while w_2 has a smaller gradient

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.

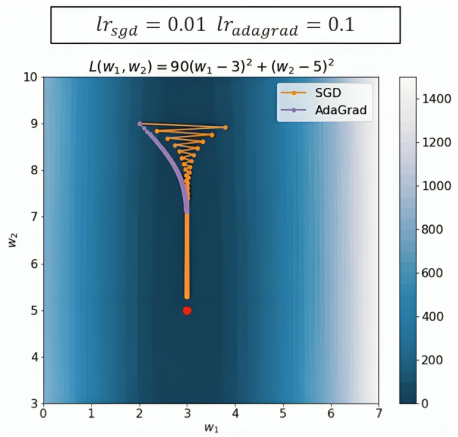


Figure 1.5: SGD vs AdaGrad. AdaGrad stops before getting close to the minimum.

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.

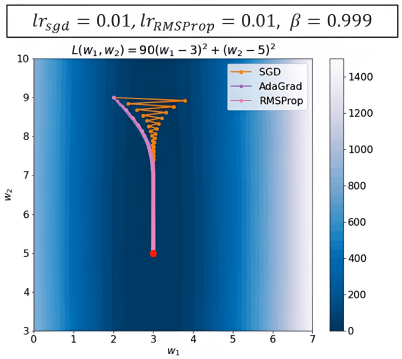


Figure 1.6: SGD vs AdaGrad vs RMSProp

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.

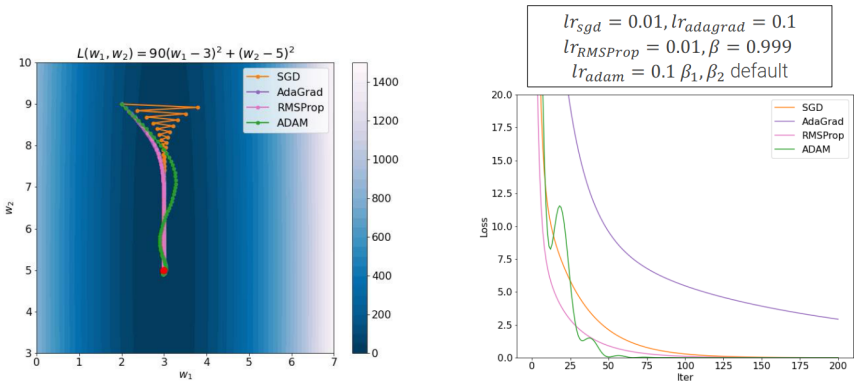


Figure 1.7: SGD vs AdaGrad vs RMSProp vs Adam

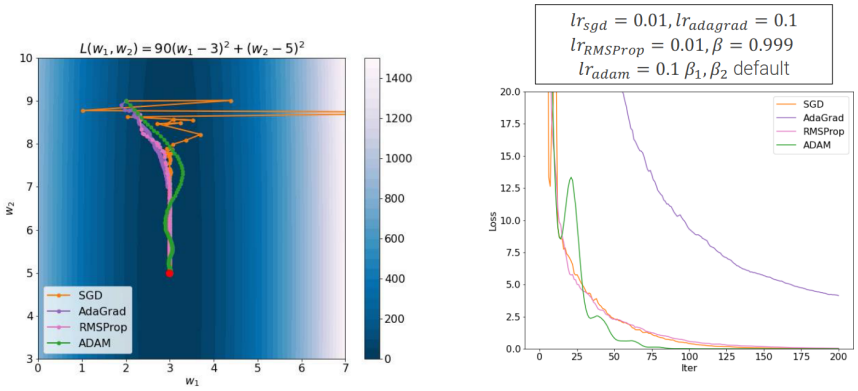
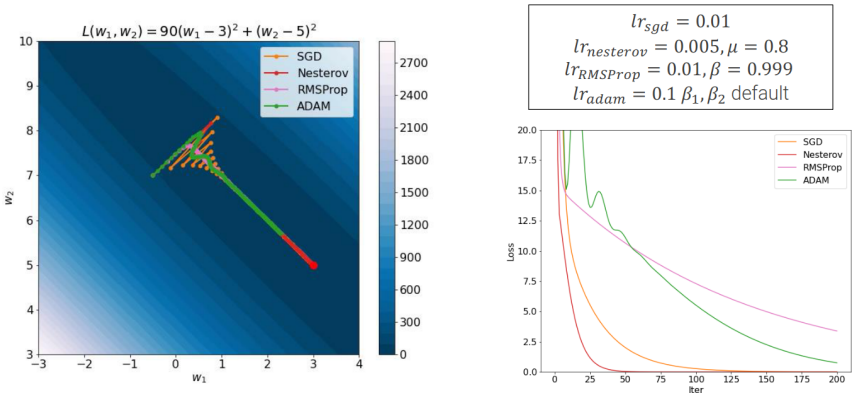


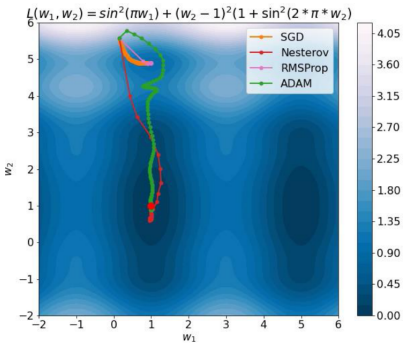
Figure 1.8: SGD vs AdaGrad vs RMSProp vs Adam with a smaller batch size

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.

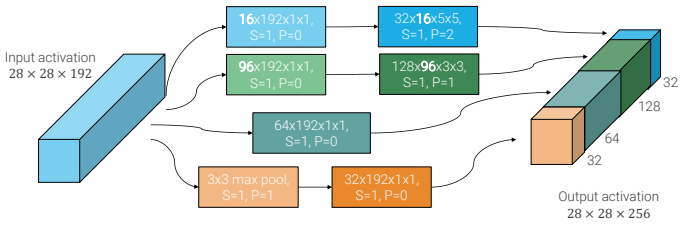


Figure 2.1: Inception module on the output of the stem layers

> **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

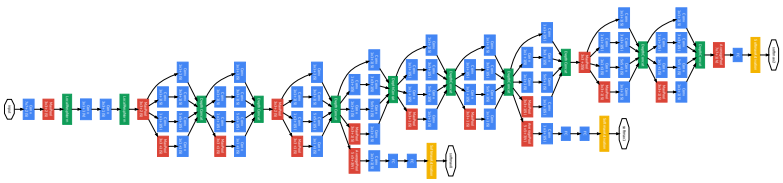


Figure 2.2: Architecture of Inception-v1

[1] Excerpt from IPCV2

2.2 Residual networks[2]

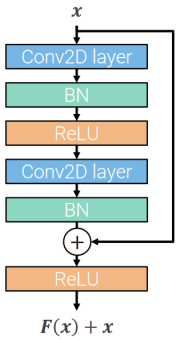
Standard residual block

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).

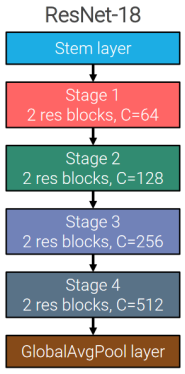


Figure 2.3: Architecture of ResNet-18

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

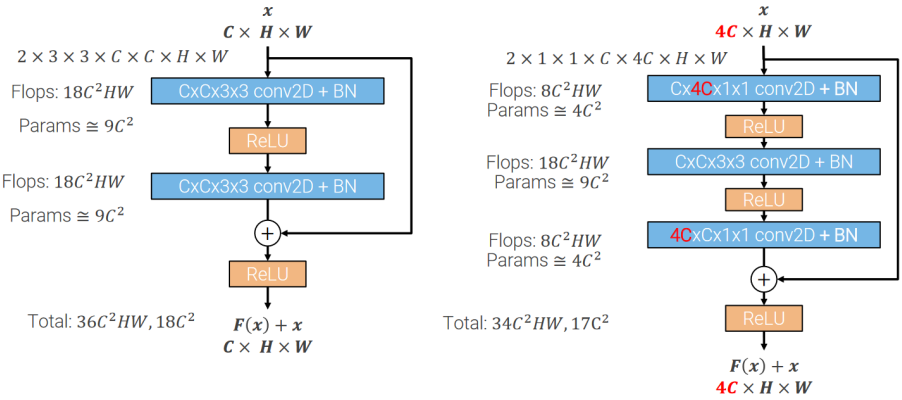


Figure 2.4: Standard residual block (left) and bottleneck block (right)

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

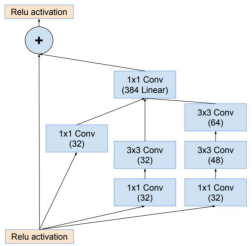
- Directly to the final concatenation.
- To a 3×3 convolution.
- To two 3×3 convolutions (i.e. a factorized 5×5 convolution).

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

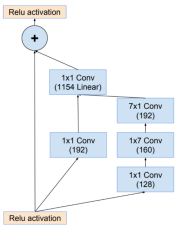
Inception-ResNet-B Three 1×1 convolutions are used to compress the input channels. Each of them leads to: Inception-ResNet-B

- 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)

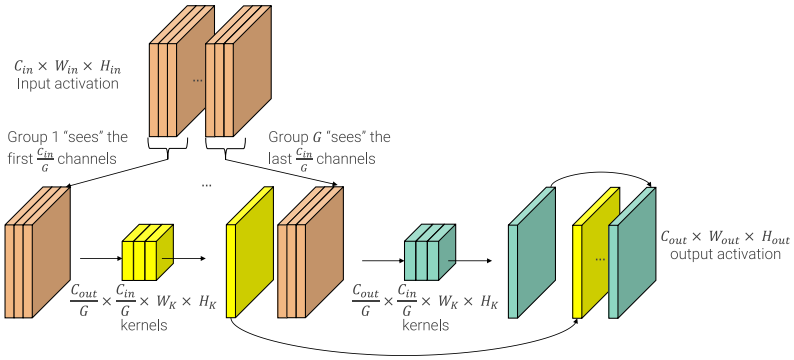
2.3 ResNeXt

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.

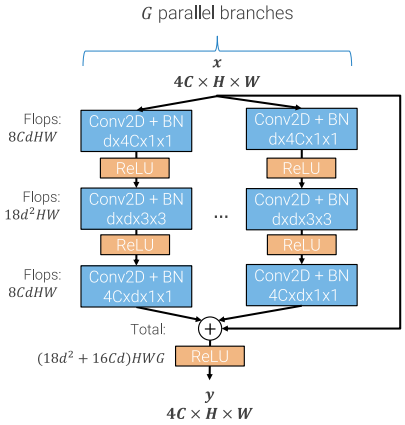


By processing the input in smaller chunks, there are the following gains:

- 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.

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:

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

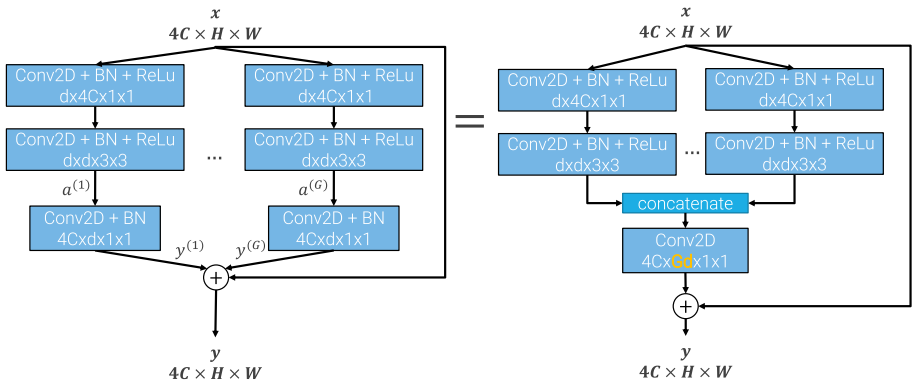
where:

- $*$ 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:

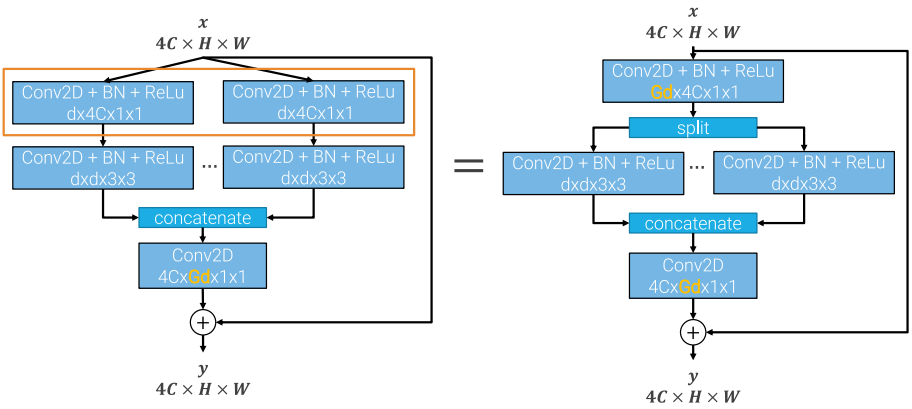
$$\begin{aligned}\mathbf{y}_k(j,i) &= \mathbf{w}_k^{(1)} \cdot \mathbf{a}^{(1)}(j,i) + \mathbf{w}_k^{(2)} \cdot \mathbf{a}^{(2)}(j,i) + \mathbf{x}_k \\ &= \underbrace{\left[\mathbf{w}_k^{(1)} \mathbf{w}_k^{(2)}\right]}_{\text{by stacking, this is a } 1\times 1 \text{ convolution with } 2d \text{ channels}}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

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.

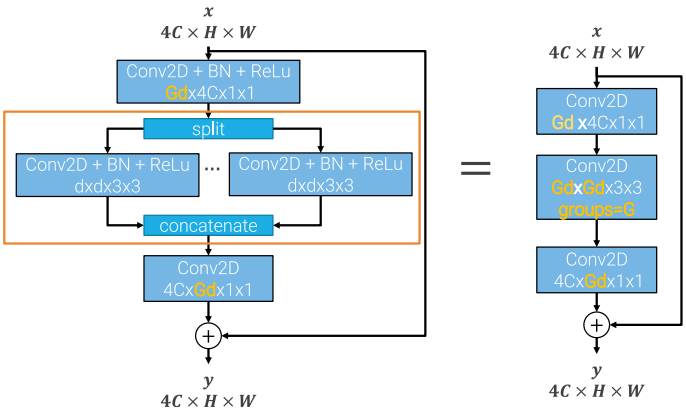


First 1×1 convolution The G 1×1 convolutions at the first layer of ResNeXt all process the same input $\mathbf{x}$. Trivially, this can also be represented using

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.

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

2.4 Squeeze-and-excitation network (SENet)

Squeeze-and-excitation module Block that weighs the channels of the input activation. Given the c-th channel of the input activation $\mathbf{x}_c$, the output $\tilde{\mathbf{x}}_c$ is computed as:

Squeeze-and-excitation module

$$\tilde{\mathbf{x}}_c = s_c \mathbf{x}_c$$

where $s_c \in [0, 1]$ is the scaling factor.

The two operations of a squeeze-and-excitation block are:

Squeeze Global average pooling to obtain a channel-wise vector.

Excitation Feed-forward network that first compresses the input channels by a ratio r (typically 16) and then restores them.

Squeeze-and-excitation network (SENet) Deep ResNet/ResNeXt with squeeze-and-excitation modules.

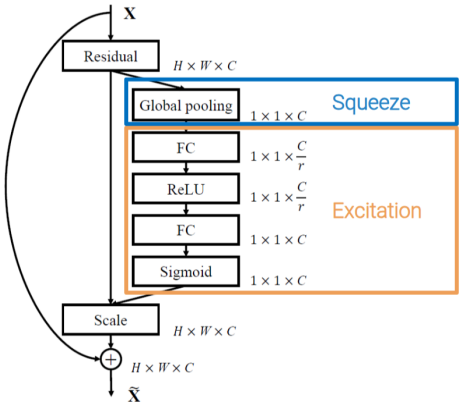


Figure 2.5: SE-ResNet module

2.5 MobileNetV2

Depth-wise separable convolution

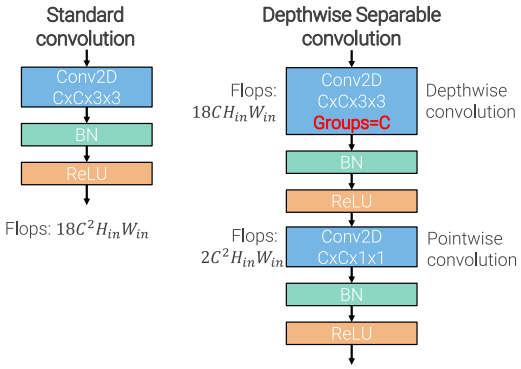
Depth-wise separable convolution Use grouped convolutions to reduce the computational cost of standard convolutions. The operations of filtering and combining features are split:

Depth-wise convolution Processes each channel in isolation. In other words, a grouped convolution with groups equal to the number of input channels is applied.

Context point-wise convolution 1×1 convolution applied after the depth-wise convolution to reproduce the channel-wide effect of standard convolutions.

Remark. The gain in computation is up to 10 times the FLOPs of normal convolutions.

Remark. Depth-wise convolutions are less expressive than normal convolutions.



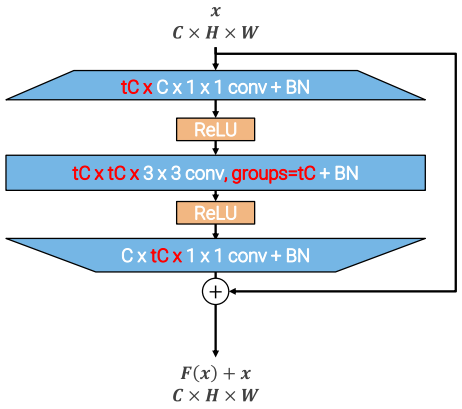
Remark. The 3×3 convolution in bottleneck residual blocks process a compressed version of the input activation, which might cause loss of information when passing through the ReLUs.

Inverted residual block Modified bottleneck block defined as follows:

Inverted residual block

1. A 1×1 convolution to expand the input channels by a factor of t.
2. A 3×3 depth-wise convolution.
3. A 1×1 convolution to compress the channels back to the original shape.

Moreover, non-linearity between residual blocks is removed as a result of theoretical studies.



MobileNetV2 Stack of inverted residual blocks.

MobileNetV2

- The number of channels grows slower compared to other architectures.
- The stem layer is lightweight due to the low number of intermediate channels.
- Due to the small number of channels, the number of channels in the activation are expanded before passing to the fully-connected layers.

Remark. Stride 2 is applied to the middle 3×3 convolution when downsampling is needed.

Table 2.1: Architecture of MobileNetV2 with expansion factor (t), number of channels (c), number of times a block is repeated (n), and stride (s).

Input	**Operator**	t	c	n	s
2242×3	`conv2d`	-	32	1	2
1122×32	`bottleneck`	1	16	1	1
1122×16	`bottleneck`	6	24	2	2
562×24	`bottleneck`	6	32	3	2
282×32	`bottleneck`	6	64	4	2
142×64	`bottleneck`	6	96	3	1
142×96	`bottleneck`	6	160	3	2
72×160	`bottleneck`	6	320	1	1
72×320	`conv2d` 1×1	-	1280	1	1
72×1280	`avgpool` 7×7	-	-	1	-
$1 \times 1 \times 1280$	`conv2d` 1×1	-	k	-	1

2.6 Model scaling

Single dimension scaling Scaling a baseline model by width, depth, or resolution. It generally always improve the accuracy.

Width scaling Increase the number of channels. — Width scaling

Depth scaling Increase the number of blocks. — Depth scaling

Resolution scaling Increase the spatial dimension of the activations. — Resolution scaling

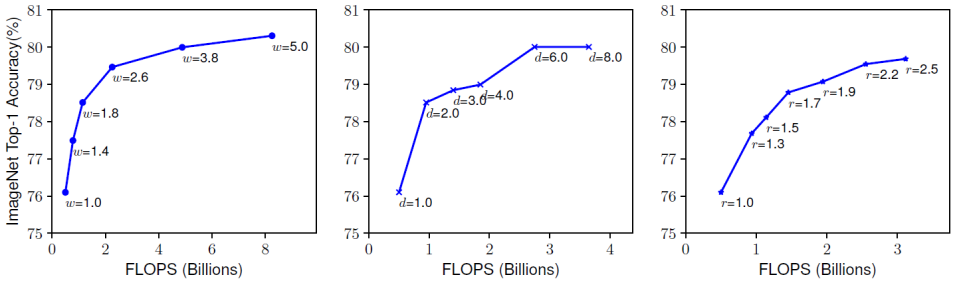


Figure 2.6: Top-1 accuracy variation with width, depth, and resolution scaling on EfficientNet

Compound scaling Scaling across multiple dimensions. — Compound scaling

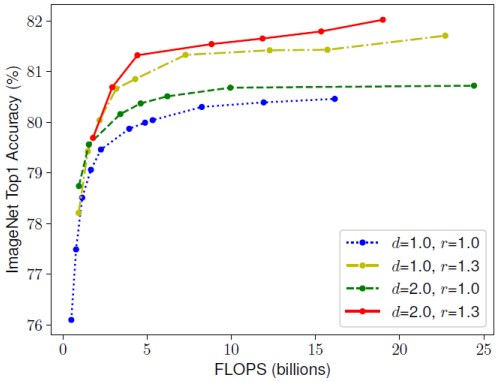


Figure 2.7: Width scaling for different fixed depths and resolutions

Compound scaling coefficient Use a compound coefficient ϕ to scale dimensions and systematically control the FLOPs increase.

| **Remark.** $\phi = 0$ represents the baseline model.

The multiplier for depth (d), width (w), and resolution (r) are determined as:

$$d = \alpha^\phi \qquad w = \beta^\phi \qquad r = \gamma^\phi$$

where α, β, and γ are subject to:

$$\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \qquad \text{with } \alpha, \beta, \gamma \geq 1$$

By enforcing this constraint, FLOPs will approximately grow by 2^ϕ (i.e., double) for each increase of ϕ.

In practice, α, β, and γ are determined through grid search.

Remark. The constraint is formulated in this way as FLOPS scales linearly with depth but quadratically with width and resolution.

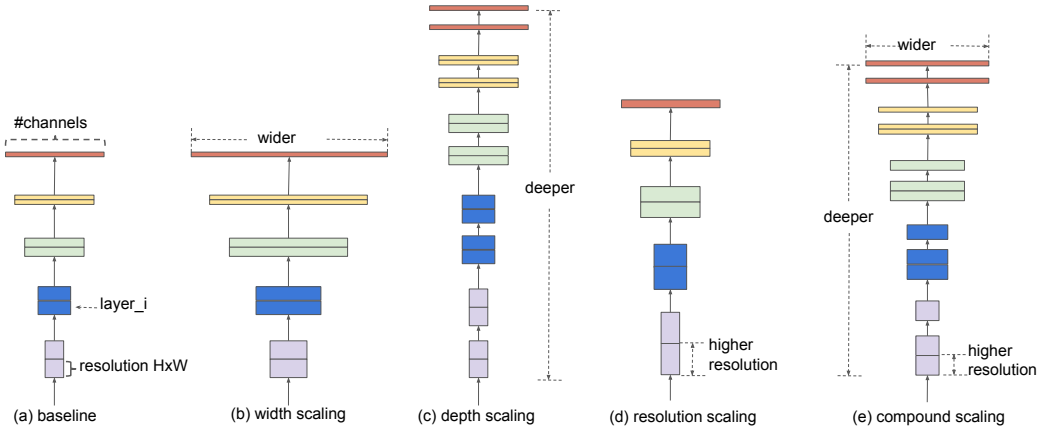
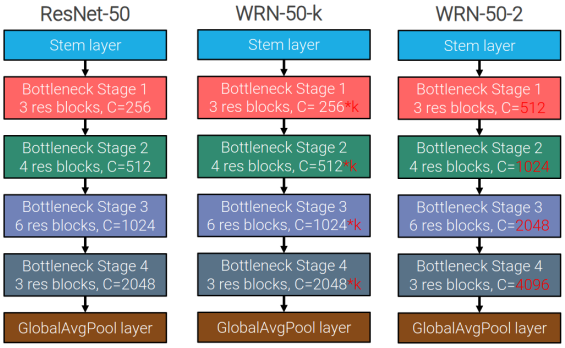


Figure 2.8: Model scaling approaches

2.6.1 Wide ResNet

Wide ResNet (WRN) ResNet scaled width-wise.

Wide ResNet (WRN)

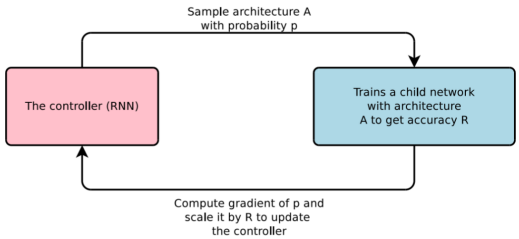


Remark. Wider layers are easier to parallelize on GPUs.

2.7 EfficientNet

Neural architecture search (NAS) Train a controller neural network using gradient policy to output network architectures.

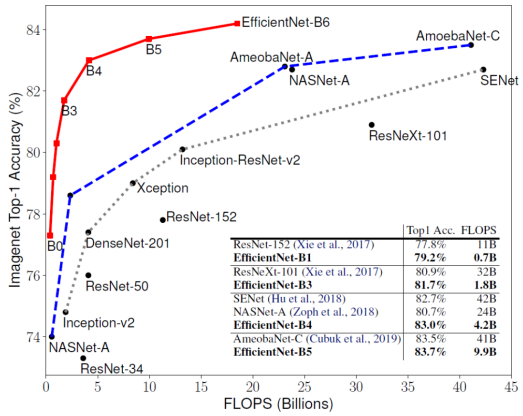
Neural architecture search (NAS)



Remark. Although effective, we usually cannot extract guiding principles from the architecture outputted by NAS.

EfficientNet-B0 Architecture obtained through neural architecture search starting from MobileNet. EfficientNet-B0

Scaling the baseline model (B0) allowed obtaining high accuracies with a controlled number of FLOPs.



	Top1 Acc.	FLOPS
ResNet-152 (Xie et al., 2017)	77.8%	11B
EfficientNet-B1	**79.2%**	**0.7B**
ResNeXt-101 (Xie et al., 2017)	80.9%	32B
EfficientNet-B3	**81.7%**	**1.8B**
SENet (Hu et al., 2018)	82.7%	42B
NASNet-A (Zoph et al., 2018)	80.7%	24B
EfficientNet-B4	**83.0%**	**4.2B**
AmoebaNet-C (Cubuk et al., 2019)	83.5%	41B
EfficientNet-B5	**83.7%**	**9.9B**

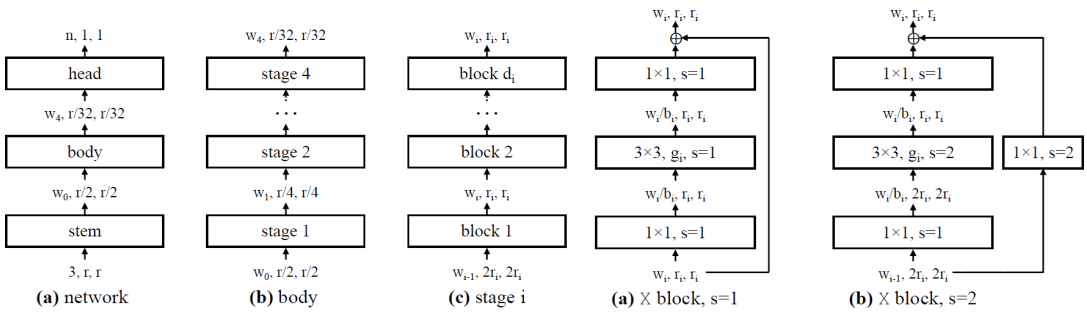
2.8 RegNet

Design space Space of a parametrized population of neural network architectures. By sampling networks from a design space, it is possible to determine a distribution and evaluate it using statistical tools. Design space

> **Remark.** Comparing distributions is more robust than searching for a single well performing architecture (as in NAS).

RegNet Classic stem-body-head architecture (similar to ResNeXt with fewer constraints) with four stages. Each stage i has the following parameters: RegNet

- Number of blocks (i.e., depth) d_i.
- Width of the blocks w_i (so each stage does not necessarily double the number of channels).
- Number of groups of each block g_i.
- Bottleneck ratio of each block b_i.



In other words, RegNet defines a 16-dimensional design space. To evaluate the architectures, the following is done:

1. Sample $n = 500$ models from the design space and train them on a low-epoch training regime.

2. Determine the error empirical cumulative distribution function F computed as the fraction of models with an error less than e:

$$F(e) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}[e_i < e]$$

3. Evaluate the design space by plotting F.

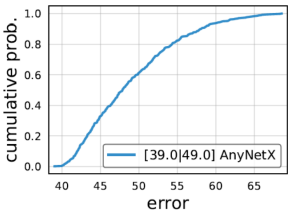


Figure 2.9: Example of cumulative distribution

Remark. Similarly to the ROC curve, the plot of the perfect design space is a straight line at 1.0 probability starting from 0% error rate.

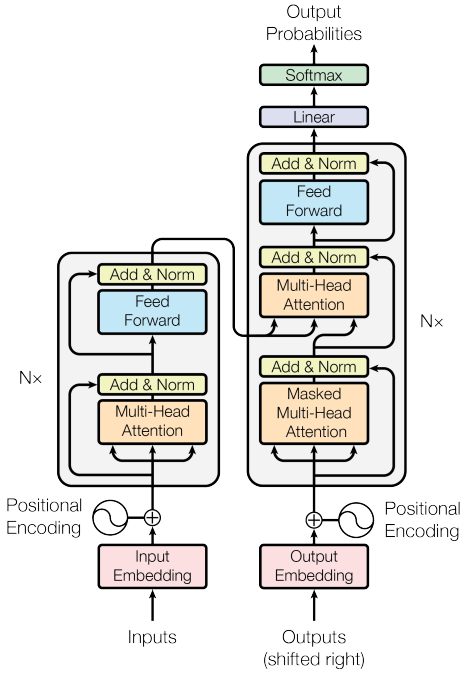
4. Repeat by fixing or finding relationships between parameters (i.e., try to reduce the search space).

Remark. In the original paper, RegNet outperformed EfficientNet. However, results were computed by retraining EfficientNet using the same hyperparameter configuration of RegNet, while the original paper of EfficientNet explicitly tuned its hyperparameters to maximize the results.

3 Transformers in computer vision

3.1 Transformer

Transformer Neural architecture designed for NLP sequence-to-sequence tasks. It heavily relies on the attention mechanism. Transformer



Autoregressive generation A transformer generates the output sequence progressively given the input sequence and the past outputted tokens. At the beginning, the first token provided as the past output is a special start-of-sequence token (`<SoS>`). Generation is terminated when a special end-of-sequence token (`<EoS>`) is generated. Autoregressive generation

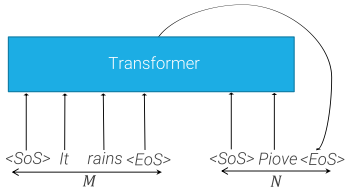


Figure 3.1: Example of autoregressive generation

3.1.1 Attention mechanism

Traditional attention Matrix computed by a neural network to weigh each token of a sequence against the tokens of another one. Traditional attention

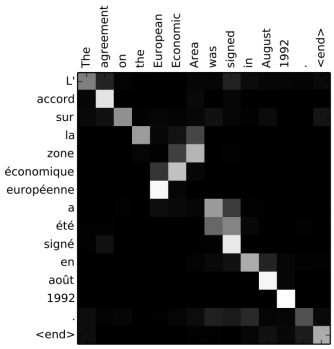


Figure 3.2: Attention weights for machine translation

Remark. Before attention, for tasks such as machine translation, the whole input sequence was mapped into an embedding that is used to influence the generation of the output.

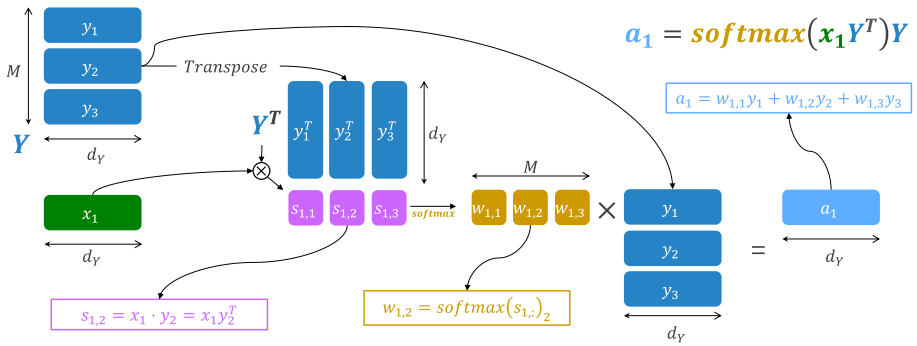
Remark. This is not the same attention of transformers as they do not directly compute attention weights between inputs and outputs.

Dot-product attention

Dot-product attention Given M input tokens $\boldsymbol{Y} \in \mathbb{R}^{M \times d_Y}$ and a vector $\mathbf{x}_1 \in \mathbb{R}^{d_Y}$, dot-product attention computes a linear combination of $\boldsymbol{Y}$ where each component is weighted based on a similarity score between $\boldsymbol{Y}$ and $\mathbf{x}_1$.

This is done as follows:

1. Determine the similarity scores of the inputs as the dot-product between $\mathbf{x}_1$ and $\boldsymbol{Y}^T$:
$$\mathbf{x}_1 \boldsymbol{Y}^T \in \mathbb{R}^M$$
2. Compute the attention weights by applying the `softmax` function on the similarity scores:
$$\texttt{softmax}(\mathbf{x}_1 \boldsymbol{Y}^T) \in \mathbb{R}^M$$
3. Determine the output activation $\mathbf{a}_1$ as the dot-product between the attention weights and the input $\boldsymbol{Y}$:
$$\mathbb{R}^{d_Y} \ni \mathbf{a}_1 = \texttt{softmax}(\mathbf{x}_1 \boldsymbol{Y}^T)\boldsymbol{Y}$$



Scaled dot-product attention

Scaled dot-product attention To add more flexibility, a linear transformation can be applied on the inputs $\boldsymbol{Y}$ and $\mathbf{x}_1$ to obtain:

Keys With the projection $\boldsymbol{W}_K \in \mathbb{R}^{d_Y \times d_K}$ such that $\mathbb{R}^{M \times d_K} \ni \boldsymbol{K} = \boldsymbol{Y}\boldsymbol{W}_K$, where d_K is the dimension of the keys.

Query With the projection $\boldsymbol{W}_Q \in \mathbb{R}^{d_X \times d_K}$ such that $\mathbb{R}^{d_K} \ni \mathbf{q}_1 = \boldsymbol{Y}\boldsymbol{W}_X$, where d_X is the length of $\mathbf{x}_1$ that is no longer required to be d_Y as there is a projection.

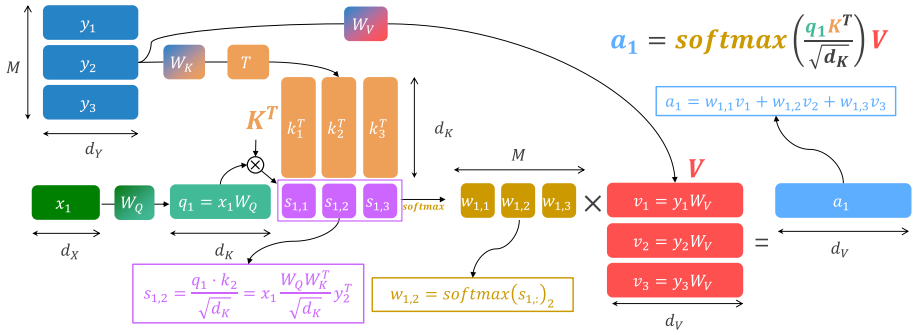
Values With the projection $\boldsymbol{W}_V \in \mathbb{R}^{d_Y \times d_V}$ such that $\mathbb{R}^{M \times d_V} \ni \boldsymbol{V} = \boldsymbol{Y}\boldsymbol{W}_K$, where d_V is the dimension of the values.

The attention mechanism is then defined as:

$$\mathbf{a}_1 = \texttt{softmax}(\mathbf{q}_1\boldsymbol{K}^T)\boldsymbol{V}$$

To obtain smoother attention weights when working with high-dimensional activations (i.e., avoid a one-hot vector from `softmax`), a temperature of $\sqrt{d_K}$ is applied to the similarity scores:

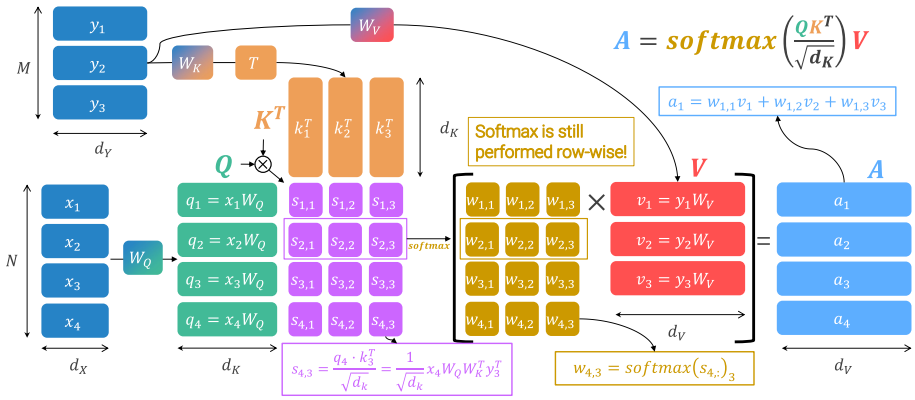
$$\mathbf{a}_1 = \texttt{softmax}\left(\frac{\mathbf{q}_1\boldsymbol{K}^T}{\sqrt{d_K}}\right)\boldsymbol{V}$$



Finally, due to the linear projections, instead of a single vector there can be an arbitrary number N of inputs $\boldsymbol{X} \in \mathbb{R}^{N \times d_X}$ to compute the queries $\mathbb{R}^{N \times d_K} \ni \boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q$. This change affects the similarity scores $\boldsymbol{Q}\boldsymbol{K}^T \in \mathbb{R}^{N \times M}$ and the output activations $\boldsymbol{A} \in \mathbb{R}^{N \times d_V}$.

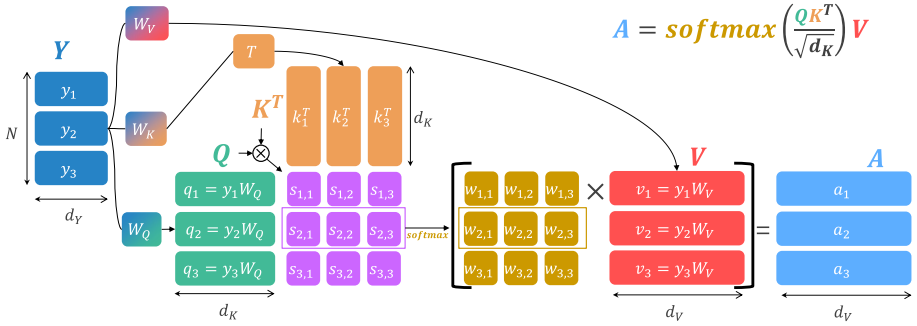
The overall attention mechanism can be defined as:

$$\boldsymbol{A} = \texttt{softmax}_{\texttt{row-wise}}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_K}}\right)\boldsymbol{V}$$



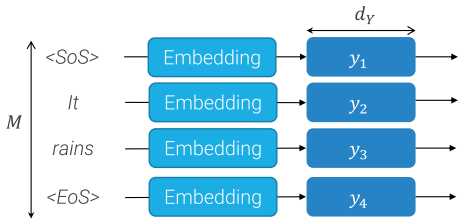
Self-attention Scaled dot-product attention mechanism where the inputs to compute keys, queries, and values are the same. Self-attention

Given an input $\boldsymbol{Y} \in \mathbb{R}^{N \times d_Y}$, the shape of each component is: $\boldsymbol{K} \in \mathbb{R}^{N \times d_K}$, $\boldsymbol{Q} \in \mathbb{R}^{N \times d_K}$, $\boldsymbol{V} \in \mathbb{R}^{N \times d_V}$, and $\boldsymbol{A} \in \mathbb{R}^{N \times d_V}$.



3.1.2 Embeddings

Embedding layer Converts input tokens into their corresponding learned embeddings of shape d_Y (usually denoted as d_{model}). Embedding layer



3.1.3 Encoder

Encoder components A transformer encoder is composed of:

Multi-head self-attention (`MHSA`) Given an input $\boldsymbol{Y} \in \mathbb{R}^{M \times d_Y}$, a `MHSA` block parallelly passes it through h different self-attention blocks to obtain the activations $\boldsymbol{A}^{(1)}, \ldots, \boldsymbol{A}^{(h)}$. The output $\boldsymbol{A}$ of the block is obtained as a linear projection of the column-wise concatenation of the activations $\boldsymbol{A}^{(i)}$: Multi-head self-attention

$$\mathbb{R}^{M \times d_Y} \ni \boldsymbol{A} = \left[A^{(1)} | \ldots | A^{(h)}\right] \boldsymbol{W}_O$$

where $\boldsymbol{W}_O \in \mathbb{R}^{hd_V \times d_Y}$ is the projection matrix.

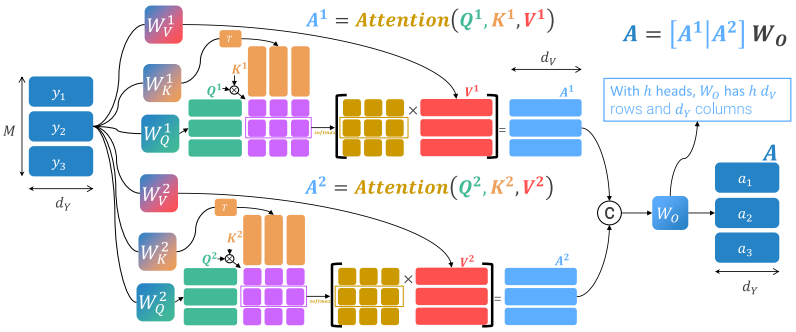


Figure 3.3: `MHSA` with two heads

Remark. The idea of multiple attention heads is to allow the model to attend to information from different representation subspaces.

Remark. Even though they can be freely set, the dimensions for queries, keys, and values of each attention head usually are $d_K = d_V = d_Y/h$.

Layer normalization (`LN`) Normalize each input activation independently to have zero mean and unit variance, regardless of the other activations in the batch. Layer normalization

Given B activations $\mathbf{a}^{(i)} \in \mathbb{R}^D$, mean and variance of each activation $i = 1, \dots, B$ are computed as:

$$\mu^{(i)} = \frac{1}{D}\sum_{j=1}^{D} \mathbf{a}_j^{(i)} \qquad v^{(i)} = \frac{1}{D}\sum_{j=1}^{D} \left(\mathbf{a}_j^{(i)} - \mu^{(i)}\right)^2$$

Each component j of the normalized activation $\hat{\mathbf{a}}^{(i)}$ is computed as:

$$\hat{\mathbf{a}}_j^{(i)} = \frac{\mathbf{a}_j^{(i)} - \mu^{(i)}}{\sqrt{v^{(i)} + \varepsilon}}$$

As in batch normalization, the actual output activation $\mathbf{s}^{(i)}$ of each input $\mathbf{a}^{(i)}$ is scaled and offset by learned values:

$$\mathbf{s}_j^{(i)} = \boldsymbol{\gamma}_j \hat{\mathbf{a}}_j^{(i)} + \boldsymbol{\beta}_j$$

Remark. Differently from computer vision, in NLP the input is not always of the same length and padding is needed. Therefore, batch normalization do not always work well.

Remark. Layer normalization is easier to distribute on multiple computation units and has the same behavior at both train and inference time.

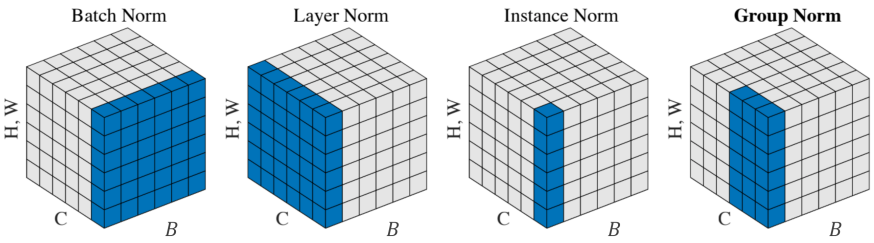


Figure 3.4: Affected axis of normalization methods

Feed-forward network (`FFN`) MLP with one hidden layer applied to each token independently. ReLU or one of its variants are used as activation function: Feed-forward network

$$\texttt{FFN}(\mathbf{x}) = \texttt{relu}(\mathbf{x}\boldsymbol{W}_1 + \mathbf{b}_1)\boldsymbol{W}_2 + \mathbf{b}_2$$

Remark. It can be implemented using two 1D convolutions with kernel size 1.

Residual connection Around the `MHSA` and `FFN` modules. Residual connection

Encoder stack Composed of L encoder layers. Encoder stack

Encoder layer Layer to compute a higher level representation of each input token while maintaining the same length of d_Y. Encoder layer

Given the input tokens $\boldsymbol{H}^{(i)} = [\mathbf{h}_1^{(i)}, \dots, \mathbf{h}_N^{(i)}]$, depending on the position of layer normalization, an encoder layer computes the following:

Post-norm transformer Normalization is done after the residual connection:

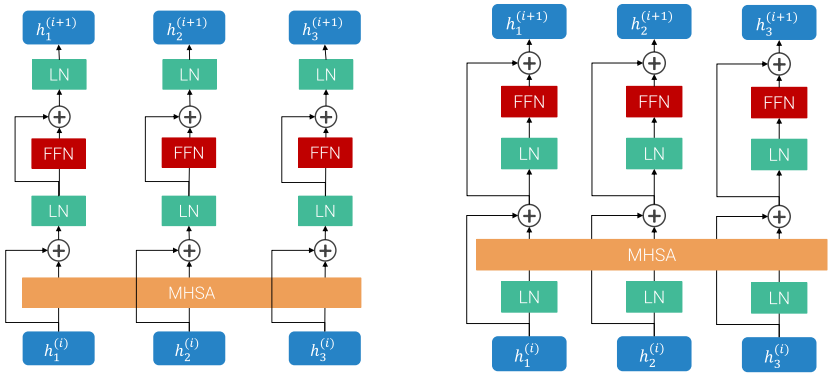
$$\bar{\mathbf{h}}_j^{(i)} = \texttt{LN}\left(\mathbf{h}_j^{(i)} + \texttt{MHSA}_{\boldsymbol{H}^{(i)}}(\mathbf{h}_j^{(i)})\right)$$
$$\mathbf{h}_j^{(i+1)} = \texttt{LN}\left(\bar{\mathbf{h}}_j^{(i)} + \texttt{FNN}(\bar{\mathbf{h}}_j^{(i)})\right)$$

Remark. In post-norm transformers, residual connections are "disrupted" by layer normalization.

Pre-norm transformer Normalization is done inside the residual connection:

$$\bar{\mathbf{h}}_j^{(i)} = \mathbf{h}_j^{(i)} + \texttt{MHSA}_{\boldsymbol{H}^{(i)}}\left(\texttt{LN}(\mathbf{h}_j^{(i)})\right)$$
$$\mathbf{h}_j^{(i+1)} = \bar{\mathbf{h}}_j^{(i)} + \texttt{FNN}\left(\texttt{LN}(\bar{\mathbf{h}}_j^{(i)})\right)$$

Remark. In practice, with pre-norm transformer training is more stable.



(a) Encoder in post-norm transformer (b) Encoder in pre-norm transformer

Remark. Of all the components in an encoder, attention heads are the only one that allow interaction between tokens.

3.1.4 Decoder

Decoder stack Composed of L decoder layers. Decoder stack

Decoder layer Layer to autoregressively generate tokens. Decoder layer

Its main components are:

Multi-head self-attention Processes the input tokens.

Encoder-decoder multi-head attention/Cross-attention Uses as query the output of the previous `MHSA` layer, and as keys and values the output of the encoder stack. In other words, it allows the tokens passed through the decoder to attend the input sequence. Cross-attention

Remark. The output of cross-attention can be seen as an additive delta to improve the activations $\mathbf{z}_j^{(i)}$ obtained from the first MHSA layer.

Remark. As queries are independent to each other, and keys and values are constants coming from the encoder, cross-attention works in a token-wise fashion.

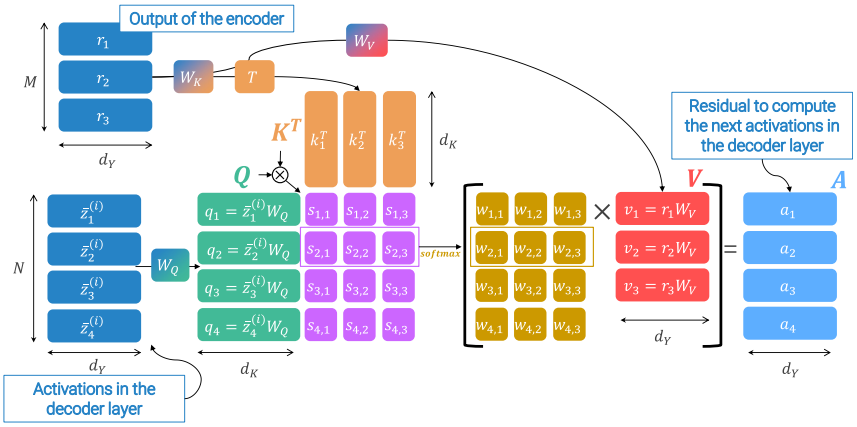


Figure 3.6: Cross-attention data flow

Feed-forward network MLP applied after cross-attention.

Remark. As for the encoder, there is a post-norm and pre-norm formulation.

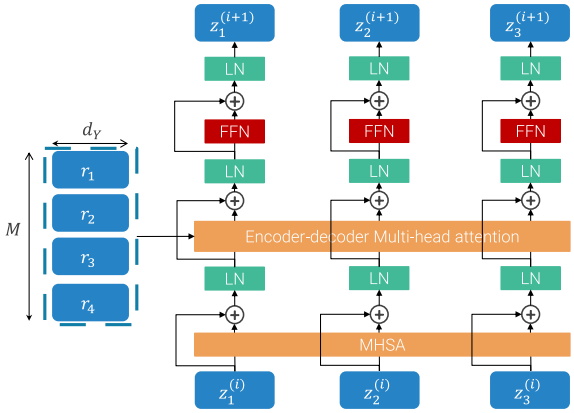


Figure 3.7: Decoder in post-norm transformer

Parallel training When training, as the ground truth is known, it is possible to train all decoder outputs in a single pass. Given a target sequence $[\texttt{<SoS>}, t_1, \ldots, t_n, \texttt{<EoS>}]$, it is processed by the decoder in the following way:

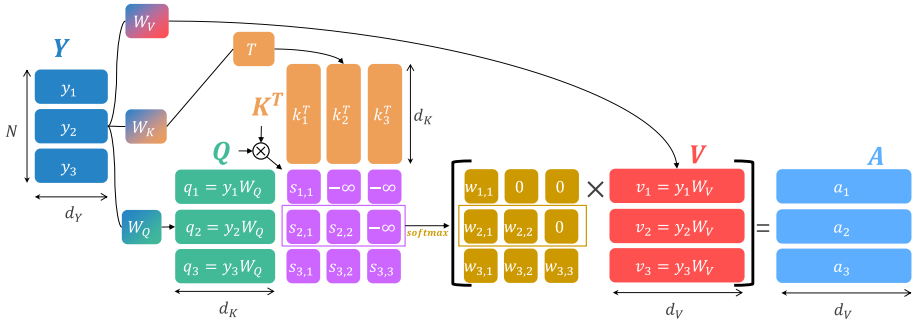
- The input is $[\texttt{<SoS>}, t_1, \ldots, t_n]$ (i.e., without end-of-sequence token).
- The expected output $[t_1, \ldots, t_n, \texttt{<EoS>}]$ (i.e., without start-of-sequence token).

In other words, with a single pass, it is expected that each input token generates the correct output token.

Remark. Without changes to the self-attention layer, a token at position i in the input is able to attend to future tokens at position $\geq i+1$. This causes a data leak as, during inference, autoregressive generation do not have access to future tokens.

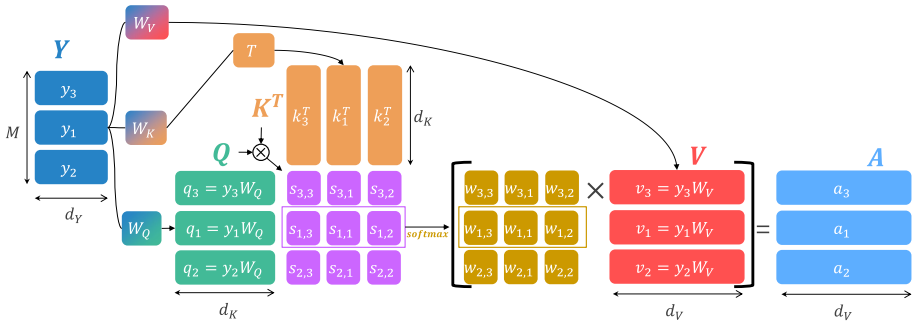
Masked self-attention Modification to self-attention to prevent tokens to attend at future positions (i.e., at their right). This can be done by either setting the similarity scores with future tokens to $-\infty$ or directly setting the corresponding attention weights to 0 (i.e., make the attention weights a triangular matrix).

Masked self-attention



3.1.5 Positional encoding

Remark (Self-attention equivariance to permutation)**.** By permuting the input sequence of a self-attention layer, the corresponding outputs will be the same as if it were the original sequence, but it is affected by the same permutation. Therefore, self-attention alone does not have information on the ordering of the tokens.



Positional encoding Vector of shape d_Y added to the embeddings to encode positional information. Positional encoding can be:

Positional encoding

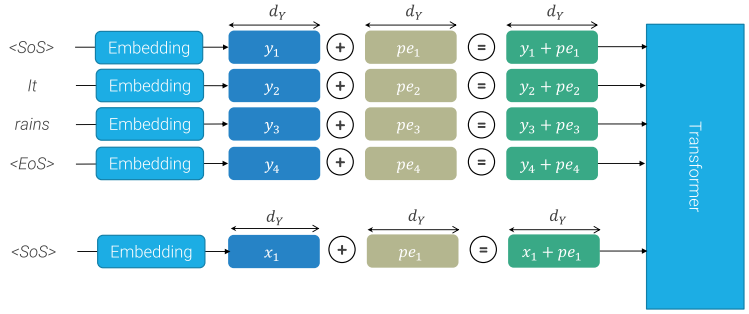
Fixed The vector associated to each position is fixed and known before training.

Example. The original transformer paper proposed the following encoding:

$$\texttt{pe}_{\texttt{pos},2i} = \sin\left(\frac{\texttt{pos}}{10000^{2i/d_Y}}\right) \qquad \texttt{pe}_{\texttt{pos},2i+1} = \cos\left(\frac{\texttt{pos}}{10000^{2i/d_Y}}\right)$$

where pos indicates the position of the token and i is the dimension of the position encoding vector (i.e., even indexes use sin and odd indexes use cos).

Learned The vector for position encoding is learned alongside the other parameters.



Remark (Transformer vs recurrent neural networks)**.** Given a sequence of n tokens with d-dimensional embeddings, self-attention and RNN can be compared as follows:

- The computational complexity of self-attention is $O(n^2 \cdot d)$ whereas for RNN is $O(n \cdot d^2)$. Depending on the task, n might be a big value.
- The number of sequential operations for training is $O(1)$ for self-attention (parallel training) and $O(n)$ for RNN (not parallelizable).
- The maximum path length (i.e., maximum number of operations before a token can attend to all the others) is $O(1)$ for self-attention (through the multi-head self-attention layer) and $O(n)$ for RNN (it needs to process each token individually while maintaining a memory).

3.2 Vision transformer

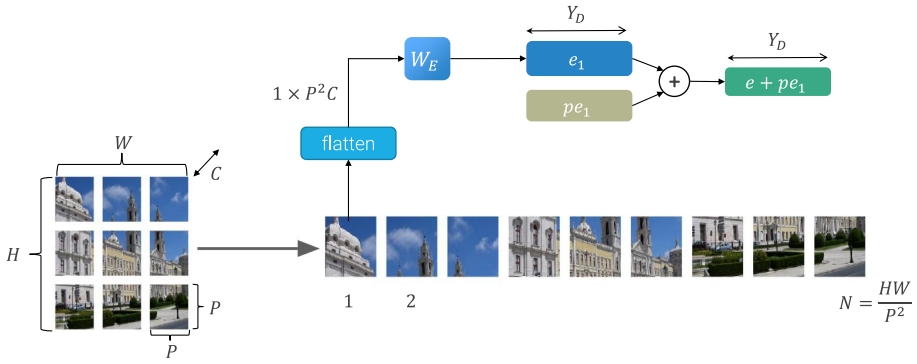
Remark. Using single pixels as tokens is unfeasible due to the complexity scaling of transformers as an $H \times W$ image results in an attention matrix of $(HW)^2$ entries.

Example. Consider an ImageNet image with shape 224×224. The attention weights will have $(224^2)^2 = 2.5$ bln entries which would require 5 GB to store them in half-precision. A classic 12 layers with 8 heads transformer would require 483 GB of memory to only store all attention matrices.

Remark. Compared to text, image pixels are more redundant and less semantically rich. Therefore, processing all of them together is not strictly necessary.

Patch Given an image of size $C \times H \times W$, it is divided into patches of size $P \times P$ along the spatial dimension. Each patch is converted into a Y_D-dimensional embedding for a transformer as follows: Patch

1. Flatten the patch into a P^2C vector.
2. Linearly transform it through a learned projection matrix $W_E \in \mathbb{R}^{P^2C \times Y_D}$.
3. Add positional information.



Vision transformer (ViT) Transformer encoder that processes embedded patches. A special classification token (`[CLS]`, as in BERT) is appended at the beginning of the sequence to encode the image representation and its embedding is passed through a traditional classifier to obtain the logits.

Vision transformer (ViT)

> **Remark.** The (pre-norm) transformer encoder used in vision is the same one as in NLP.