Vanishing et exploding gradients

Julien Romero

Rappels

- On met à jour les poids d'un réseau de neurones à l'aide la descente de gradient
- Le calcul des gradients se fait grâce à la backpropagation
  - En partant de la fin du réseau, on peut propager le gradient vers le début du réseau
  - Basé sur la dérivation chaînée.

Backpropagation pour un réseau profond

$$\frac{\partial L}{\partial W_l} = \frac{\partial L}{\partial \sigma(Z_l)} \frac{\partial \sigma(Z_l)}{\partial Z_l} \frac{\partial Z_l}{\partial W_l}$$

Fonction d'activation

Backpropagation pour un réseau profond

$$\frac{\partial L}{\partial W_{l-1}} = \frac{\partial L}{\partial \sigma(Z_l)} \frac{\partial \sigma(Z_l)}{\partial Z_l} \frac{\partial Z_l}{\partial \sigma(Z_{l-1})} \frac{\partial \sigma(Z_{l-1})}{\partial Z_{l-1}} \frac{\partial Z_{l-1}}{\partial W_{l-1}}$$

Backpropagation pour un réseau profond

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \sigma(Z_l)} \prod_{i=2}^{l} \left(\frac{\partial \sigma(Z_i)}{\partial Z_i} \frac{\partial Z_i}{\partial \sigma(Z_{i-1})} \right) \frac{\partial \sigma(Z_1)}{\partial Z_1} \frac{\partial Z_1}{\partial W_1}$$

Convergence des produits infinis

$$\prod_{n=0}^{\infty} a_n = l \Rightarrow \lim_{n \to +\infty} a_n = 1$$

$$\forall n, |a_n| < 1 \Rightarrow \prod_{n=0}^{\infty} a_n = 0$$

$$\forall n, a_n > 1 \Rightarrow \prod_{n=0}^{\infty} a_n = \infty$$

Les gradients en pratique

Plus on rajoute de couches, plus il est difficile de garder des gradients raisonnables :

- **Vanishing gradients** : Les gradients sont très proches de 0
- **Exploding gradients** : Les gradients deviennent immenses

Que faire ?

Solution à l'exploding gradient - Gradient Clipping

Idée : On va fixer une valeur maximale du gradient et réduire les gradients en fonction

```
total_norm = torch.sum(grad * grad)
clip_coef = max_norm / (total_norm + 1e-6)
grad *= clip_coef
```

> torch.nn.utils.clip_grad_norm_(*parameters*, *max_norm*, *norm_type=2.0*, *error_if_nonfinite=False*, *foreach=None*) [SOURCE]
>
> Clip the gradient norm of an iterable of parameters.
>
> The norm is computed over the norms of the individual gradients of all parameters, as if the norms of the individual gradients were concatenated into a single vector. Gradients are modified in-place.

Solution aux deux problèmes - Initialisation des poids

- Dans un MLP, si tous les poids et biais valent 0, que valent les sorties ?
- Quand peut-on dire sur les gradients ?

Solution aux deux problèmes - Initialisation des poids

Première possibilité : Utiliser une constante

- Dans un MLP, si tous les poids et biais valent 0, que valent les sorties ?
  - Toutes les sorties valent 0
- Quand peut-on dire sur les gradients ?
  - Tous les gradients sont les mêmes (et donc toutes les mises à jours aussi).
- On ne casse pas la symétrie
  - Les poids resteront identiques

MAUVAISE IDÉE

Solution aux deux problèmes - Initialisation des poids

Deuxième possibilité : Utiliser de petits nombres proches de 0 (gaussienne centrée sur 0)

$$W \sim \mathcal{N}(0, 0.01)$$

Ici, la std vaut 0.01.

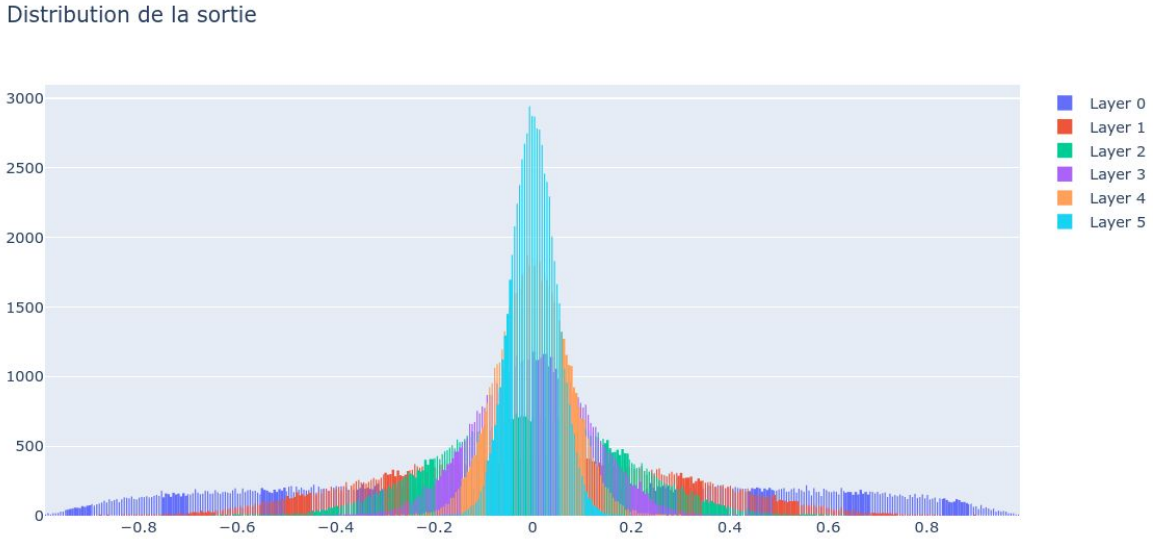
Marche pour des petits réseaux, mais pas pour des réseaux profonds.

Solution aux deux problèmes - Initialisation des poids

Deuxième possibilité : Utiliser de petits nombres proches de 0 (gaussienne centrée sur 0)

On a un MLP avec 6 couches et une fonction d'activation **tanh**.

L'entrée suit une loi normale de std 1, centrée sur 0.



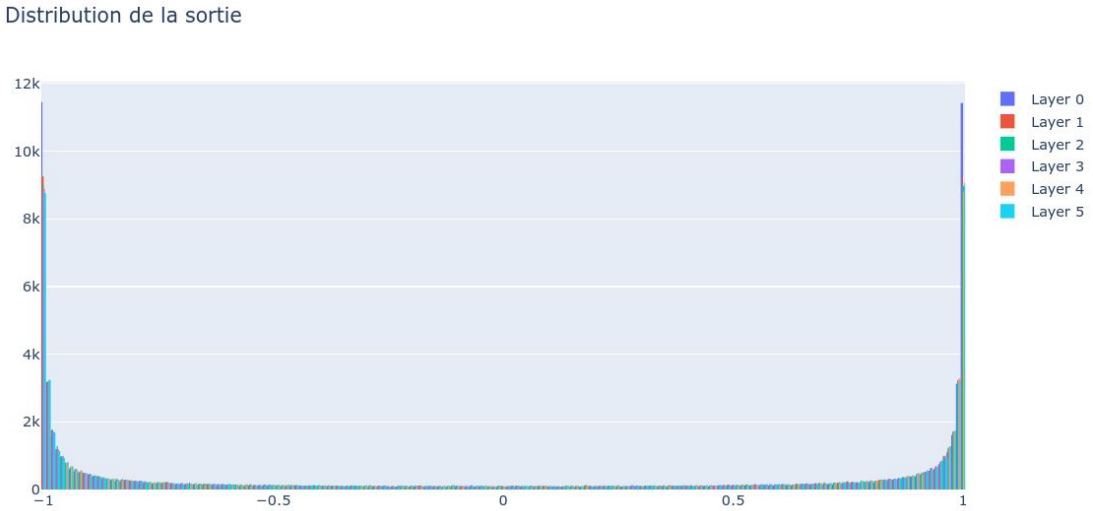
Les sorties se rapprochent de 0, donc le gradient local reste proche de 0 => on n'apprend pas

Solution aux deux problèmes - Initialisation des poids

Deuxième possibilité : Utiliser de petits nombres proches de 0 (gaussienne centrée sur 0)

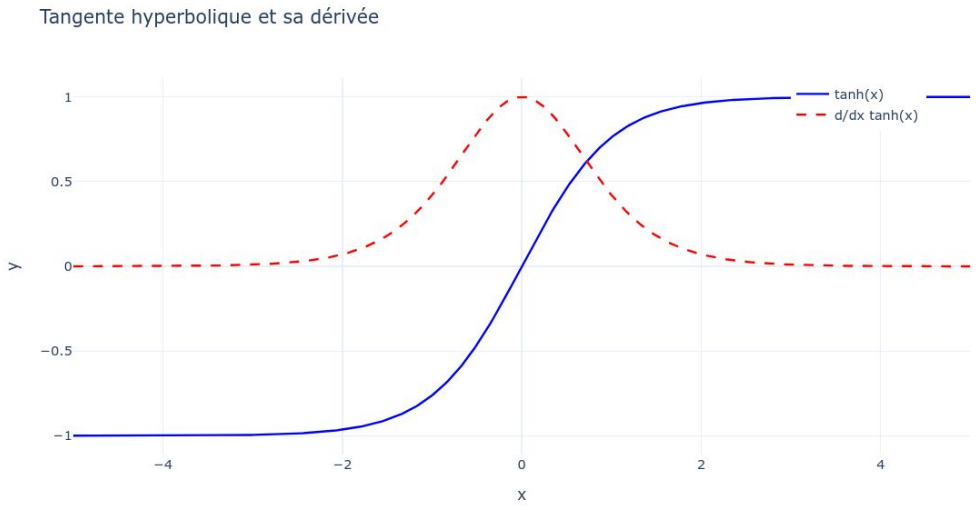
On augmente la std de W à 0.05

$$W \sim \mathcal{N}(0, 0.05)$$



Les sorties se rapprochent de -1 et 1 (saturation), donc le gradient de la fonction d'activation reste proche de 0 => on n'apprend pas

Rappel sur la tangente hyperbolique



Résultats assez similaires avec la sigmoid

Solution aux deux problèmes - Initialisation des poids

Deuxième possibilité (mieux) : Utiliser la distribution de Xavier (uniforme ou normale)

$$W \sim \mathcal{N}(0, \frac{1}{D_{in}})$$

Din est la dimension de l'entrée avant W.

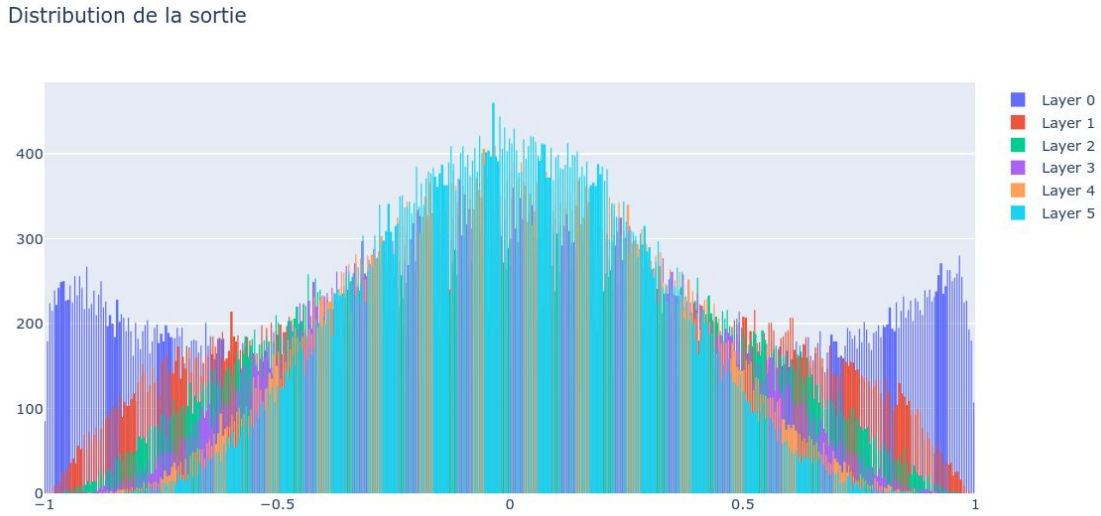
Dans un réseau convolutif, Din = kernel_size ² * input_channels

On suppose les **distributions centrées sur 0** . Ne marche pas avec ReLU.

- Kaiming/MSRA : $\sigma = \sqrt{\frac{2}{D_{in}}}$

Solution aux deux problèmes - Initialisation des poids

Deuxième possibilité (mieux) : Utiliser la distribution de Xavier



Solution aux deux problèmes - Initialisation des poids

- De nombreuses autres méthodes d'initialisation en fonction de l'architecture.
- Dans Pytorch, chaque module a une initialisation par défaut que l'on peut changer.

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Variables

- **weight** (*torch.Tensor*) – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Solution aux deux problèmes - Initialisation des poids

- De nombreuses autres méthodes d'initialisation en fonction de l'architecture.
- Dans Pytorch, chaque module a une initialisation par défaut que l'on peut changer.

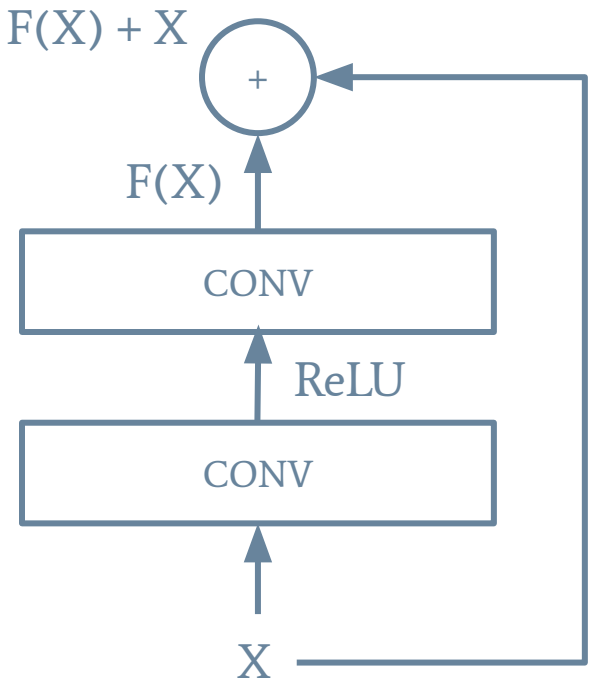
CLASS torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

- **weight** (*Tensor*) – the learnable weights of the module of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$. The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$

Solution aux deux problèmes - Initialisation des poids

- De nombreuses autres méthodes d'initialisation en fonction de l'architecture.
- Dans Pytorch, chaque module a une initialisation par défaut que l'on peut changer.
- On peut changer l'initialisation avec le module *torch.nn.init*

Solution au vanishing gradient - Réseau résiduels



Idée : Ajouter un raccourci pour permettre au gradient de circuler.

Imaginons que l'on ait L couches résiduelles

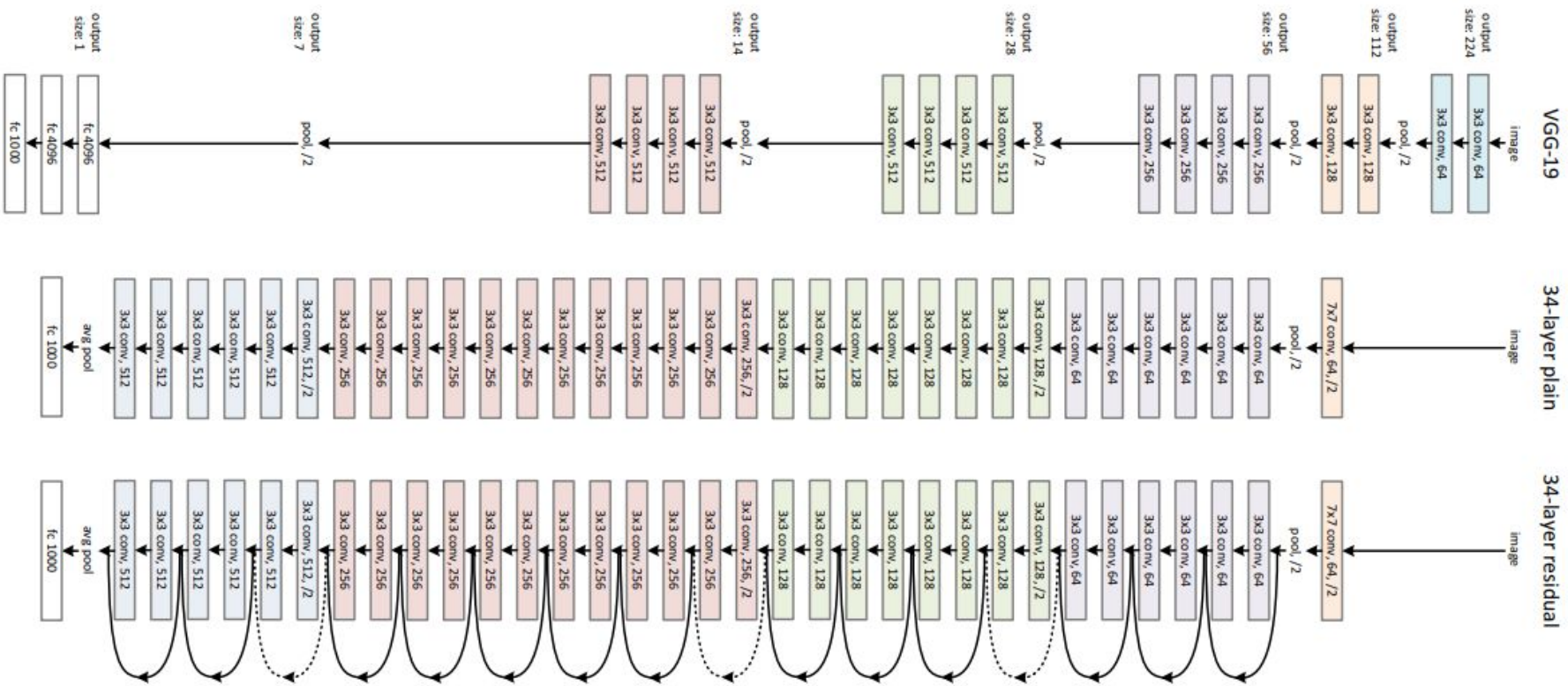
$$x_{l+1} = F_l(x_l) + x_l$$

$$x_L = \sum_{i=0}^{L-1} F_i(x_i) + x_0$$

$$\frac{\partial L}{\partial x_0} = \frac{\partial L}{\partial x_L}\frac{\partial x_L}{\partial x_0} = \frac{\partial L}{\partial x_L}\left(1 + \frac{\partial}{\partial x_0}\sum_{i=0}^{L} F_i(x_i)\right)$$

Le gradient après les résidu est conservé = plus difficile à faire disparaître

ResNet


VGG-19
image
3x3 conv, 64
3x3 conv, 64
pool, /2
3x3 conv, 128
3x3 conv, 128
pool, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
fc 4096
fc 4096
fc 1000
output size: 224
output size: 112
output size: 56
output size: 28
output size: 14
output size: 7
output size: 1
34-layer plain
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000
34-layer residual
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000

Solution aux deux problèmes - Batch/Layer Normalisation

Idée : On veut normaliser la sortie d'une couche pour que la moyenne soit 0 et l'écart-type 1 selon chaque dimension.

On doit donc apprendre :

- La moyenne suivant chaque dimension
- L'écart-type suivant chaque dimension

En pratique, on apprend :

- Une moyenne glissante
- Un écart-type glissant
- Un facteur d'échelle
- Un biais

Solution aux deux problèmes - Batch/Layer Normalisation

Pendant l'entraînement, on calcule les valeurs glissantes qui seront utilisées à l'inférence!

On reçoit en entrée 1 batch X de M éléments de dimension D. On peut calculer les moyennes sur ce batch pour chaque dimension i :

$$\mu_i = \frac{1}{M}\sum_{j=1}^{M} x_i^j \qquad \sigma_i = \sqrt{\frac{1}{M}\sum_{j=1}^{M}(x_i^j - \mu_i)^2}$$

Comme on ne peut pas obtenir la moyenne/écart-type sur tous les points, et comme la moyenne sur un batch est bruitée, on préfère les moyennes et écart-types glissants (alpha est un hyperparamètre) :

$$\mu_{mov_i} = \alpha\mu_{mov_i} * (1-\alpha)\mu_i \qquad \sigma_{mov_i} = \alpha\sigma_{mov_i} * (1-\alpha)\sigma_i$$

On met à jour uniquement pendant l'entraînement !

Solution aux deux problèmes - Batch/Layer Normalisation

On normalise ensuite la sortie :

Pendant l'entraînement :

$$\hat{x}_i = \frac{x_i - \mu}{\sigma}$$

Pendant l'inférence :

$$\hat{x}_i = \frac{x_i - \mu_{mov_i}}{\sigma_{mov_i}}$$

On apprend deux vecteurs gamma et beta de dimension D changeant l'échelle et faisant une translation :

$$Out_i = \gamma \odot \hat{x}_i + \beta$$

Solution aux deux problèmes - Batch/Layer Normalisation

BatchNorm1d

CLASS torch.nn.BatchNorm1d(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *device=None*, *dtype=None*) [SOURCE]

BatchNorm2d

CLASS torch.nn.BatchNorm2d(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *device=None*, *dtype=None*) [SOURCE]

2D = Normalise par channel

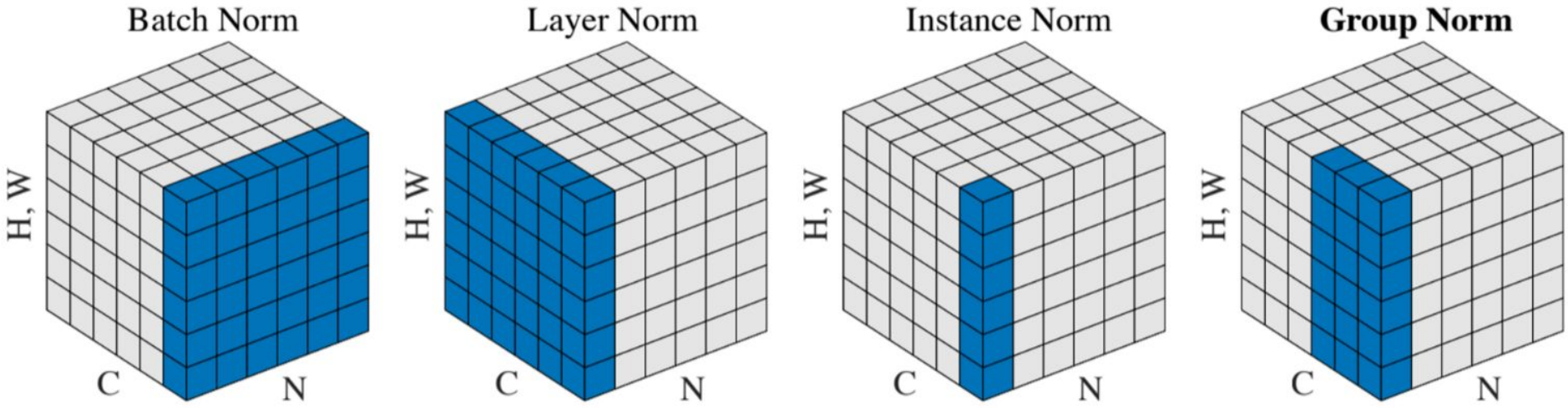
Solution aux deux problèmes - Batch/Layer Normalisation

- En général, après une couche linéaire ou une convolution et avant la fonction d'activation

Avantages :

- Plus facile d'entraîner des réseaux profonds
- Permet des taux d'apprentissage plus élevés, donc une convergence plus rapide
- Réseaux plus robustes à l'initialisation
- Participe à la régularisation
- Très léger à l'inférence

Variantes de la Batch Normalization



Wu and He, “Group Normalization”, ECCV 2018

Autres techniques

- Régularisation
- Changer la fonction d'activation
  - En général ReLU est meilleures que tanh et sigmoid.
  - Essayer des variantes plus avancées : Leaky ReLU, ELU, GELU, ...
- Utiliser un autre optimiseur
  - Adam

En résumé

- Vanishing/Exploding Gradient
- Gradient clipping
- Initialisation des poids
- Réseaux résiduels
- Batch normalisation et variantes
- Régularisation, fonction d'activation, optimiseur