UNIVERSITÉ LAVAL

GLO-4030/7030
APPRENTISSAGE PAR RÉSEAUX DE NEURONES PROFONDS

Régularisation

Philippe Giguère

Régularisation

- Training set n'est qu'un échantillon de la vraie distribution p_{data}

Mastigoproctus giganteus dans ImageNet

- Cherche à bien performer en dehors de ce *training set*
- Régularisation peut réduire de l'erreur de test, et parfois augmenter l'erreur empirique (*train*)
- Grande capacité des réseaux profonds : capables d'apprendre par cœur le *train set* (on en reparle plus loin)

Stratégies générales

- Contraintes sur le modèle :
  - valeurs des paramètres θ
  - souvent encode une préférence pour des modèles plus « simples » (rasoir d'Ockham*)
- Cherche à réduire la variance sur l'estimé des paramètres optimaux θ…
- … sans trop induire de biais
- Conseil pratique : préférable d'avoir un réseau plus gros avec bonne régularisation que régulariser en réduisant la taille du réseau

*les hypothèses suffisantes les plus simples sont les plus vraisemblables

Principales stratégies

1. Pénalités sur la norme des poids (Sec. 7.1)
2. Data augmentation (Sec. 7.4)
3. Robustesse au bruit (Sec. 7.5)
4. Apprentissage semi-supervisé (Sec. 7.6)
5. Apprentissage multi-tâche (Sec. 7.7)
6. Early stopping (Sec. 7.8)
7. Parameter tying/sharing (Sec. 7.9)
8. Représentation sparse (Sec. 7.10)
9. Méthodes par ensemble (Sec. 7.11)
10. Dropout (et autres variantes) (Sec. 7.12)
11. Autres

Principales stratégies

- Pas toujours clair lesquelles sont les plus efficaces, pour un problème donné
- En pratique, on en combine plusieurs
- On les souhaite les plus orthogonales possible

1. Pénalités sur les poids

- Cherche à limiter la capacité du modèle
- Ajoute une pénalité à la fonction de coût :

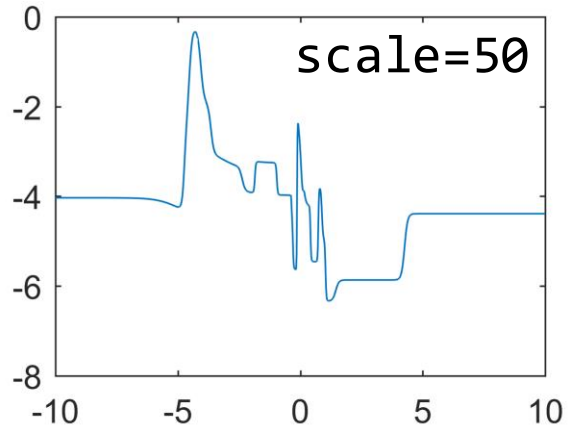
$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \lambda\Omega(\theta)$$

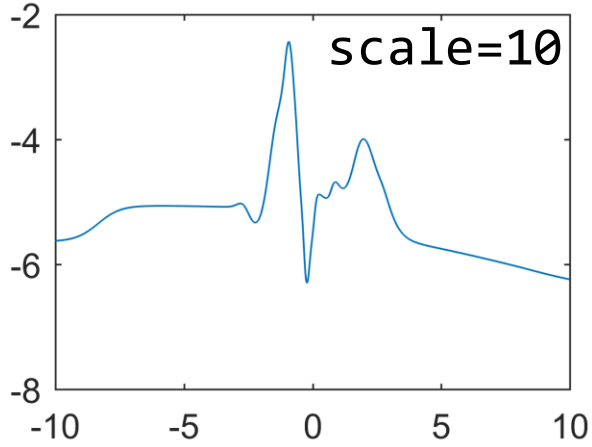
- Plusieurs choix possibles pour $\Omega(\bullet)$
- Pratique généralisée : pénalités appliquées seulement sur les poids *w* et non les biais*
- Possible d'avoir une fonction $\Omega(\bullet)$ différente à chaque couche, mais fait exploser le nombre d'hyperparamètre

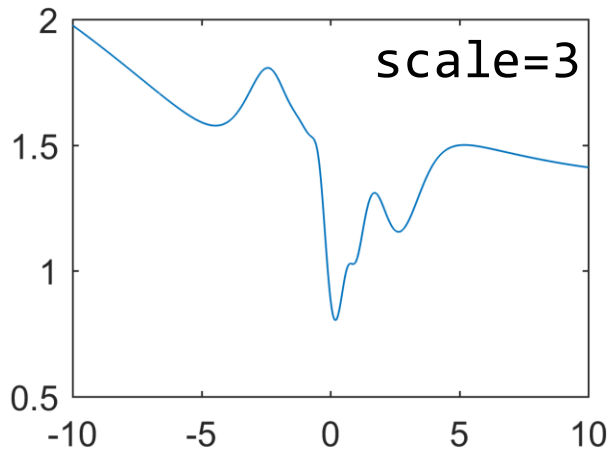
*régulariser les biais aura tendance à underfitter.

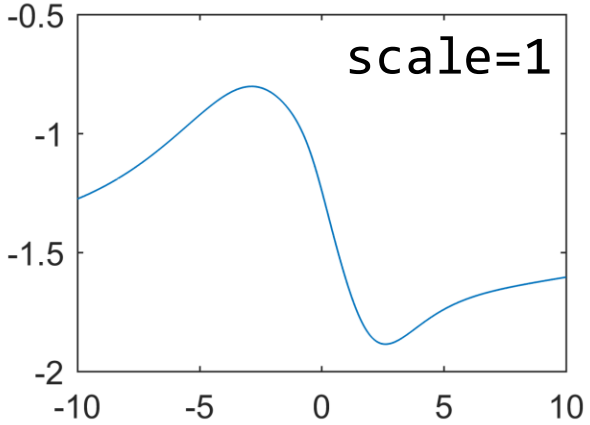
Exemple réseau sigmoïde

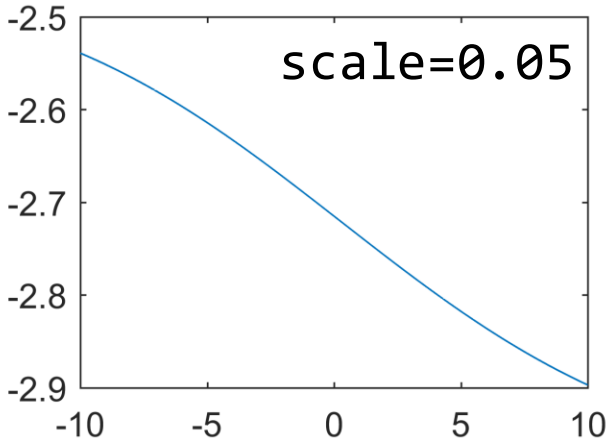
Initialisés au hasard












```
W1 = scale*randn(30,1);
W2 = randn(1,30);
b1 = scale*randn(30,1);
b2 = randn(1,1);
```

Poids *w* faibles → dégénère vers réseau linéaire

1. Pénalité $\Omega = L^2$ norm

- Autre nom : ***weight decay***
- La plus commune
- Cherche à rapprocher les poids *w* vers l'origine
- Son gradient :

$$\Omega(\theta) = \frac{1}{2}\|w\|_2^2$$

somme des entrées de w au carré

$$\frac{\partial \Omega}{\partial w}$$

$$\nabla_w \tilde{J}(w; X, y) = \lambda w + \nabla_w J(w; X, y)$$

$$w \leftarrow w - \grave{o}\nabla_w \tilde{J}(w; X, y)$$

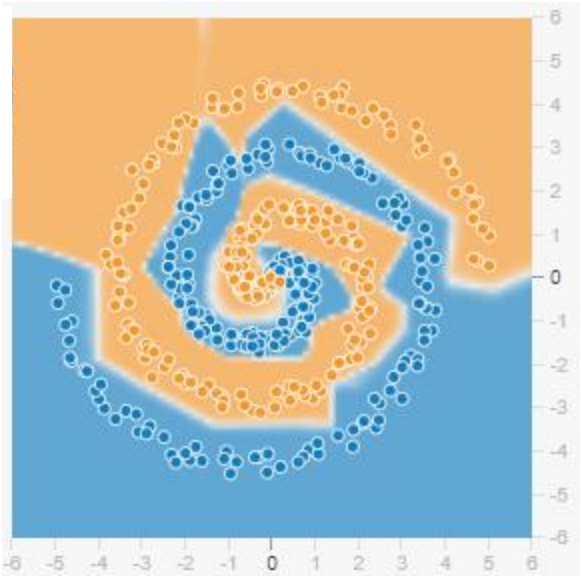
$$w \leftarrow (1 - \grave{o}\lambda) w - \grave{o}\nabla_w J(w; X, y)$$

cherche à éroder les poids à chaque update

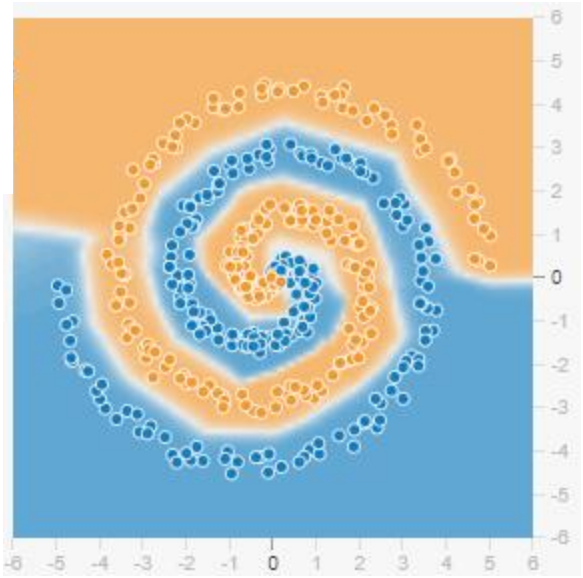
- Affectera peu les dimensions de *w* qui ont un grand gradient

Exemples

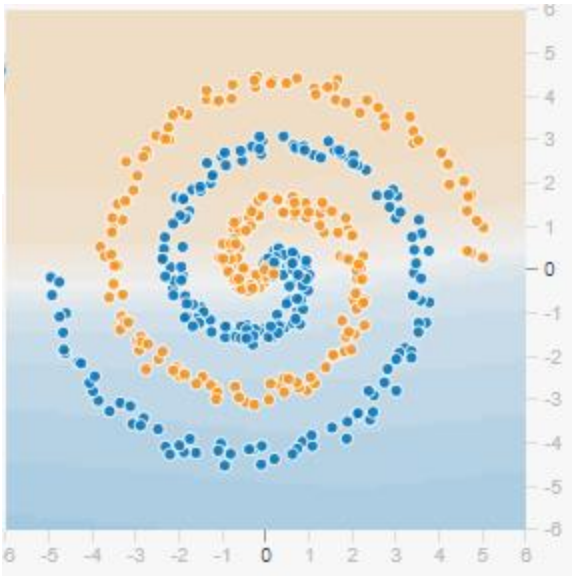
$\lambda=0$



$\lambda=0.001$



$\lambda=0.01$



1. Pénalité $\Omega = L^1$ norm

- Moins commune $\Omega(\theta) = \frac{1}{2}\|w\|_1 = \sum_i |w_i|$

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \lambda\Omega(\theta)$$

$$\nabla_w \tilde{J}(w; X, y) = \nabla_w J(w; X, y) + \lambda sign(w)$$

$$sign(w) = \begin{cases} -1 & \text{si } x < 0 \\ 0 & \text{si } x = 0 \\ 1 & \text{si } x > 0 \end{cases}$$

le changement de poids à l'update est indépendant de la magnitude de *w*

- Va induire des solutions *sparse* (poids w_i à 0) pour λ suffisamment grand
- Indique quels *features* sont utiles (*feature selection*)

1. Max norm constraint

- Limite sur la norme absolue des poids d'un neurone $\|w\|_2 < c$
- Valeur typique* de c est 3 à 4
- Empêche le réseau « d'exploser »
- Approche moins connue, mais utilisée dans [1]

[1] Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al. JMLR 2014.

*cs231n.github.io/neural-networks-2/

2. Data Augmentation

- Le meilleur régularisateur c'est d'avoir plus de données
- Pas toujours possible d'en collecter plus
- Créer des ***fausses données*** à partir des vraies
- Fonctionne particulièrement bien pour classification visuelle, car on y cherche l'invariance à certaines transformations :
  - Rotation
  - Translation
  - Réflexion horizontale ou verticale
  - Échelle
  - Intensité lumineuse, etc…
- Comprendre le processus (physique) de création des données

2. Data Augmentation

Horizontal flip

Vertical flip

Random crop

Random scale

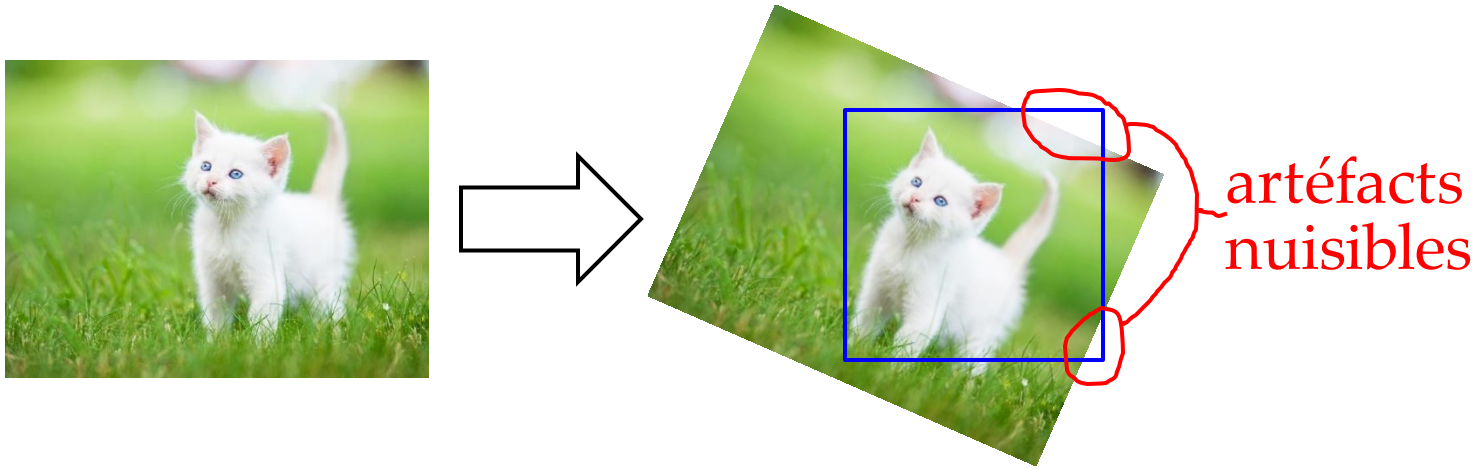
Random rotation

Combinaison de ces opérations

Attention à ne pas changer un 6 en 9…

Note pratico-pratique

- Le data augmentation se fait souvent sur le CPU, conjointement avec lecture disque multithread
- Rotation est coûteuse, fonction de la taille d'image
  - crop → rotation → crop
- Attention aux effets de bord des rotations



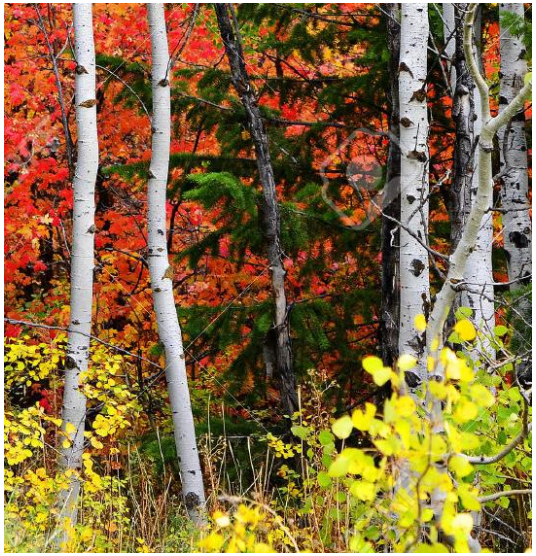
- De préférence, les versions SIMD (`pillow-simd`)
- Doit estimer la fourchette des paramètres de transformation (e.g. angle entre ±10°) (encore plus d'hyperparamètres ☹)

Exemples dans `torchvision`

- `RandomCrop`
- `RandomHorizontalFlip`
- `RandomVerticalFlip`
- `RandomResizedCrop`
- `RandomSizedCrop`
- `ColorJitter`

https://github.com/pytorch/vision/blob/master/torchvision/transforms.py

transforms.ColorJitter

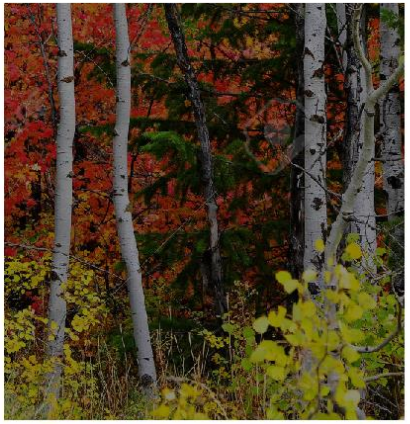


Brightness

Trial #0

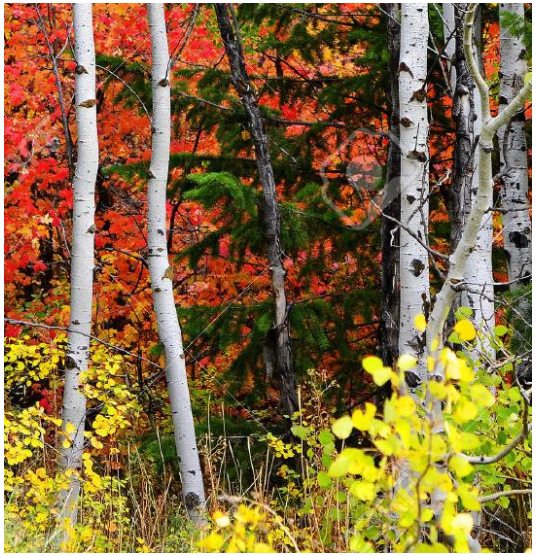
Trial #1

Trial #2



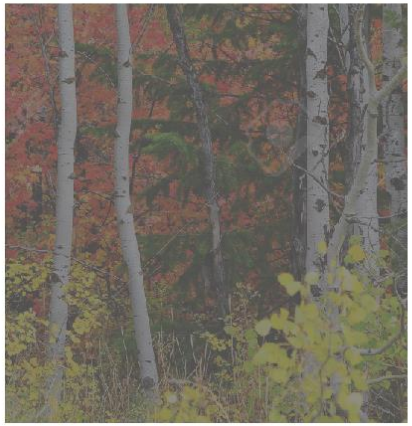
Trial #3

transforms.ColorJitter



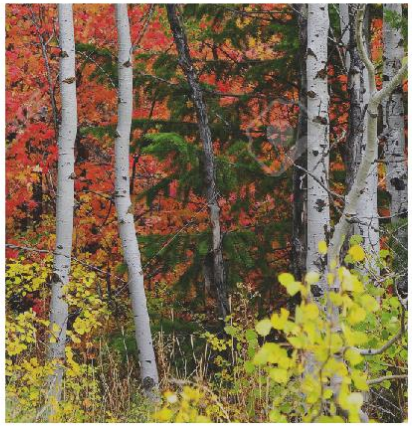
Contrast

Trial #0



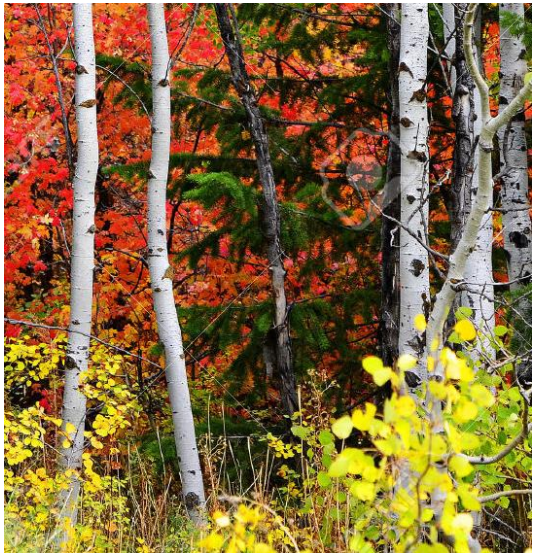
Trial #1

Trial #2



Trial #3

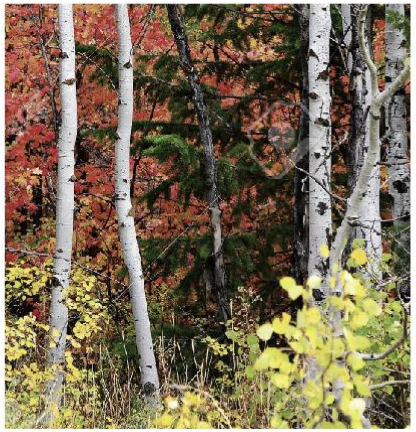
transforms.ColorJitter



Saturation

Trial #0

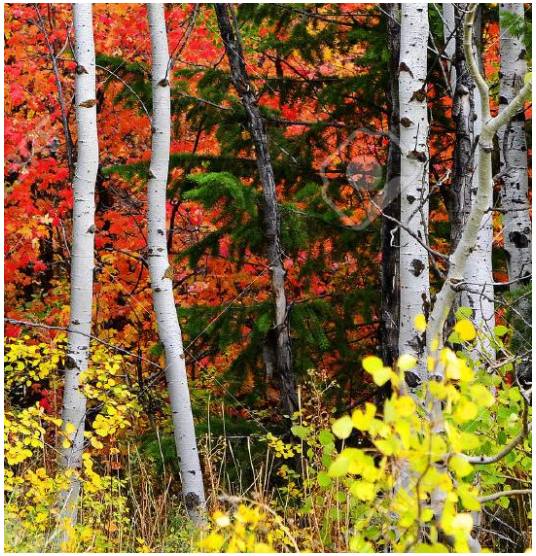
Trial #1



Trial #2

Trial #3

transforms.ColorJitter



Hue

Essai #0

Essai #1

Essai #2

Essai #3

3. Robustesse au bruit

- Possible aussi d'ajouter du bruit sur :
  - L'entrée $\boldsymbol{x}$
    - Similarité avec *denoising autoencoders*
  - Les unités cachées h_i
    - Simule variabilité des *features* de plus haut niveau d'abstraction
    - Similarité avec Dropout (dans quelques acétates)
  - Les poids w (moins courant)
    - Indique qu'on cherche un modèle moins sensible aux paramètres (sections plus planes du profil de coût J)
    - Parfois utilisé dans les RNN
- Bien sélectionner la magnitude du bruit

4. Apprentissage semi-supervisé

- Possède des données étiquetées $\{x_l, y\}$ et non-étiquetées $\{x_u\}$
- Avec données non-étiquetées x_u, peut apprendre une représentation $h = f(x_u)$
- Idéalement, des exemples x appartenant à la même classe y sont proches dans h
- Similaire à l'esprit de PCA (principal component analysis), embedding
- Utilisation d'autoencodeur, par exemple

4. Apprentissage semi-supervisé

Pseudo-étiquetage [1]

- Entraîne sur les données étiquetées $\{X_l, Y_l\} \rightarrow \theta_l$
- Applique ce réseau *f* sur les données non-étiquetées $Y_u = f(X_u; \theta_l)$
- Crée un nouveau jeu de données en concaténant le training set avec les données pseudo-label : $\{X_l + X_l,\ Y_l + Y_u\}$
- Règle du pouce : pas plus 1/4 à 1/3 de pseudo-label dans une mini-batch

[1] Dong-Hyun Lee, Pseudo-Label : The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks, 2013.

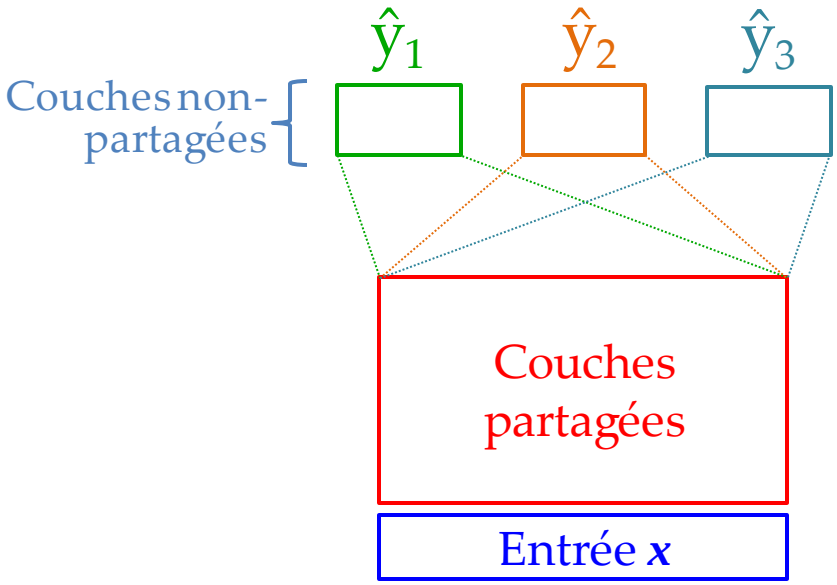
5. Apprentissage multitâche

- Si on partage des paramètres pour des tâches (reliées), réseau plus contraint

- Souvent augmente la généralisation

5. Apprentissage multitâche

Exemple typique

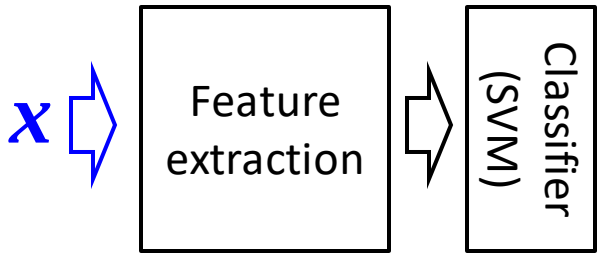


Augmente le nombre effectif d'exemples x pour lesquels une tâche a accès (meilleure généralisation)

- Doit y avoir un certain lien entre les tâches
- Si perte multitâche, difficulté de choisir leur importance (choix des λ)

5. Réseau comme *feature extraction*

- Réseaux pré-entrainés ont démontré qu'ils fonctionnent bien comme *feature extractor*

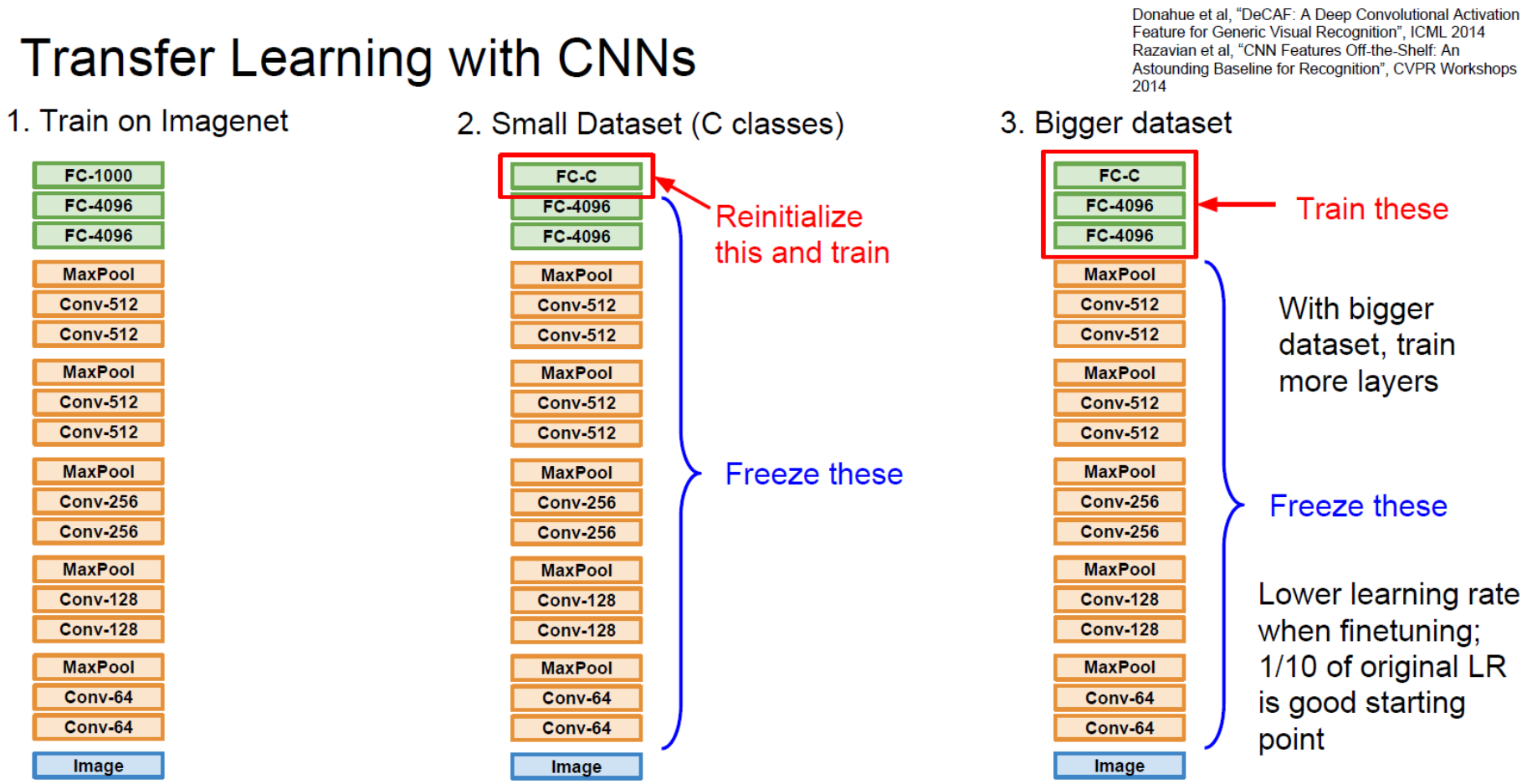


Razaviann et al., CNN Features off-the-shelf: an Astounding Baseline for Recognition, CVPR 2014.

- Aussi pour les tâches de *visual place recognition*
  - Réseaux pré-entraînés sur le sémantique fonctionnent mieux pour les changements de saisons que classification

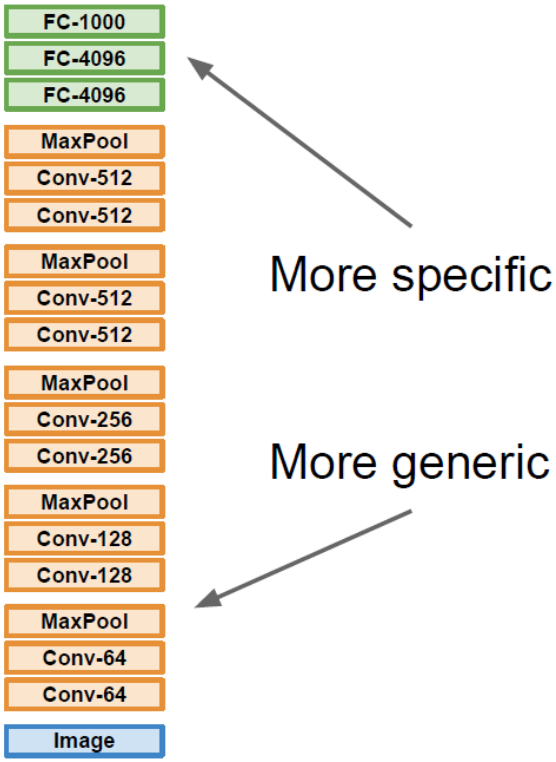
5. Finetuning

Exemple multitâche, mais sous forme séquentiel



tire de : cs231n

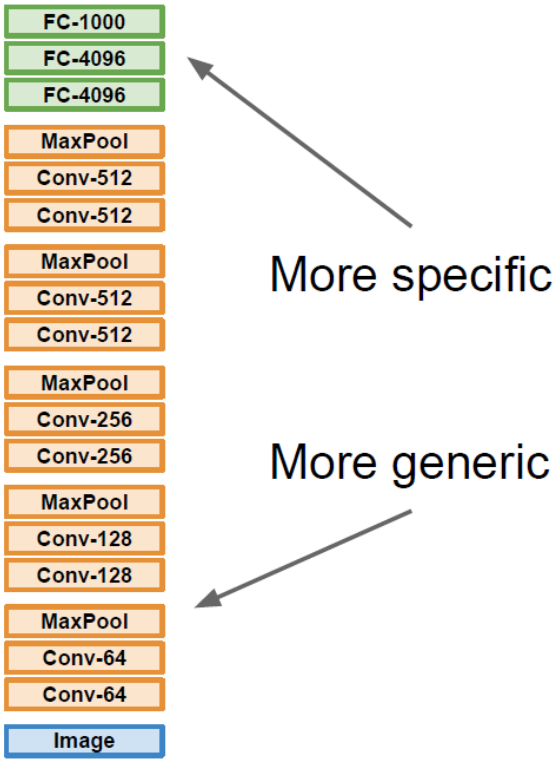
5. Finetuning



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	
quite a lot of data		

tire de : cs231n

5. Finetuning

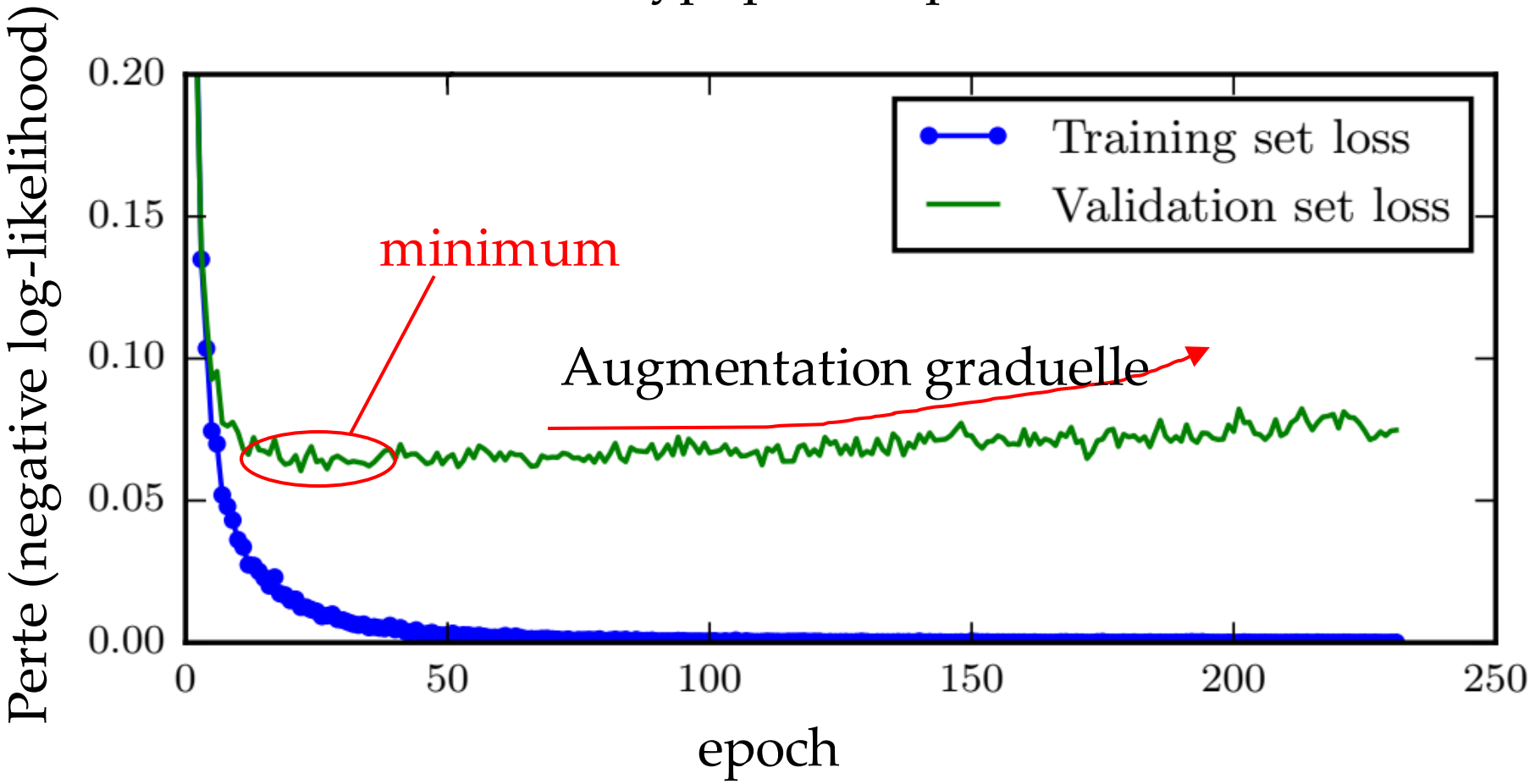


	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

tire de : cs231n

6. *Early stopping*

Forme typique des pertes



6. *Early stopping*

- Durand l'entraînement, on conserve les paramètres θ^* qui ont la perte minimale sur l'ensemble de validation
- Après toutes les epochs, on retourne θ^*
- Nécessite un ensemble de validation (moins de données pour l'entrainement)
- Après, pour utiliser toutes les données :
  - **A** réinitialiser le réseau, et faire un nouvel entrainement avec, pour le nombre d'epoch identifié préalablement
  - **B** poursuivre l'entrainement à partir de θ^*, pour un certain nombre d'epoch (mais combien?)

6. *Early stopping*

- Très utilisé (et facile d'utilisation!)
- Peut être vu comme une forme de sélection de l'hyper-paramètre du nombre d'epoch
- Contrôle la capacité effective du réseau, via le nombre d'epoch
- Coût additionnel de faire la validation
  - mais l'inférence pure est plus rapide que forward+backward+update

model.`train()` vs. *model*.`eval()`

- Transparent : pas de changement d'architecture, méthode d'entraînement, fonction objective, poids permis, etc…

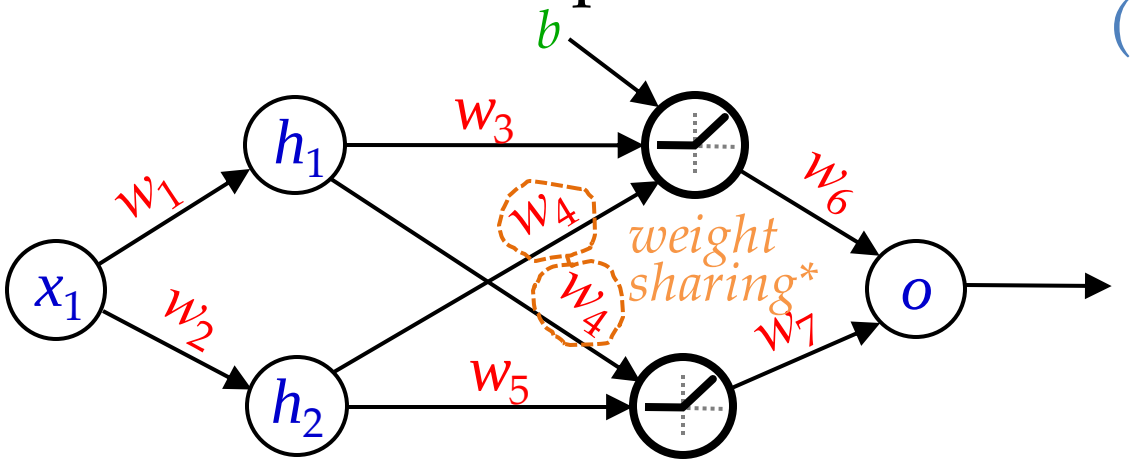
6. *Early stopping*

- Que dit la théorie?
- Régularisation se fait car on limite la distance parcourue dans l'espace des paramètres θ, autour de l'initialisation
- Sous certaines simplifications, on peut démontrer que early stopping est similaire au *weight decay* L^2 (p. 242-245)

7. *Parameter tying/sharing*

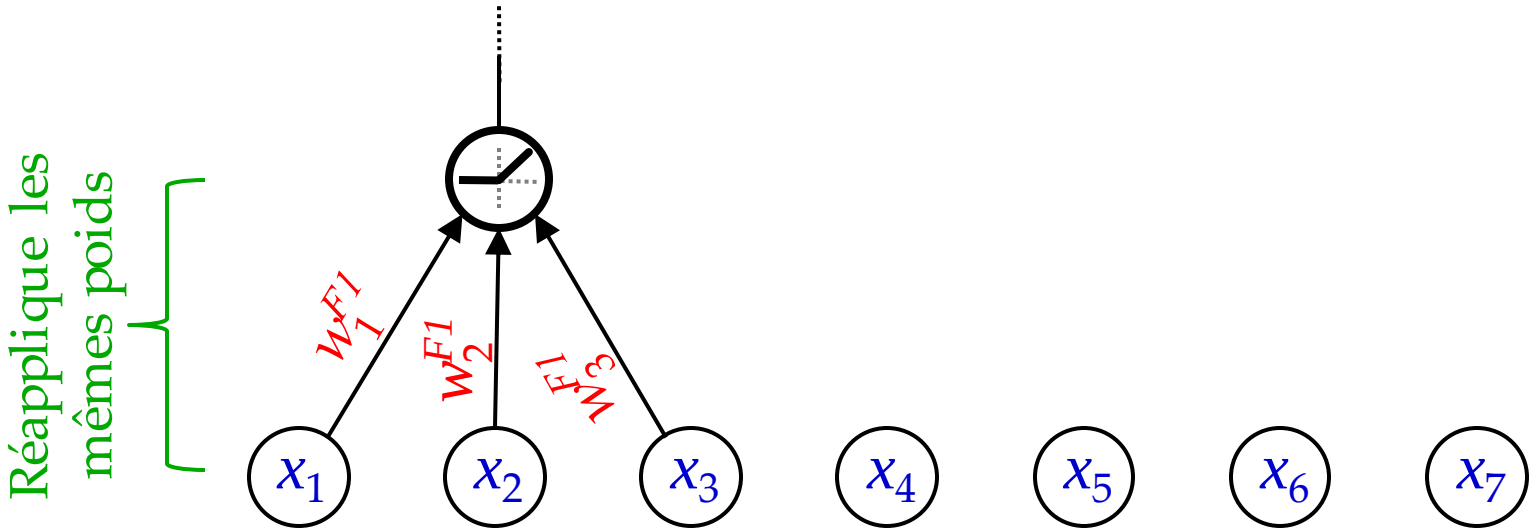
- Réduction du nombre de paramètres θ réduit la capacité du réseau
- Façon d'induire un *prior* sur les solutions appropriées, via des interdépendances dans les paramètres θ du modèle
- Comment :
  - 2 paramètres doivent prendre valeur similaire *(tying)*
  - 2 paramètres doivent prendre valeur identique *(sharing)*

+commun



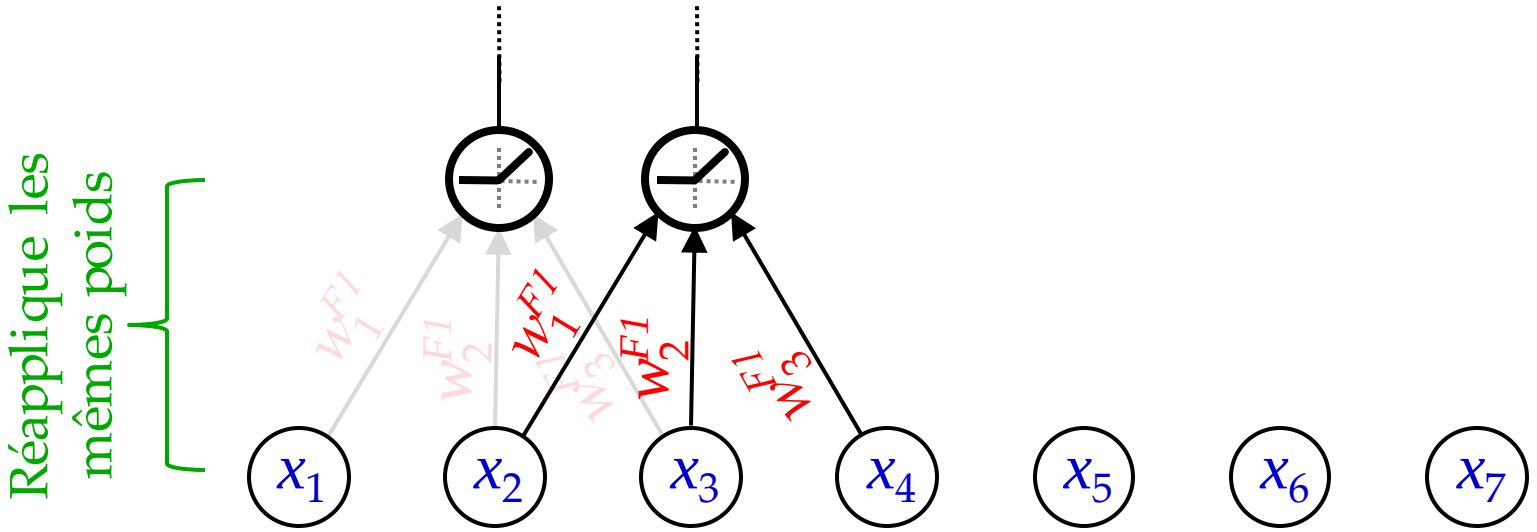
7. *Parameter sharing*

Convolutional Neural Network (CNN)



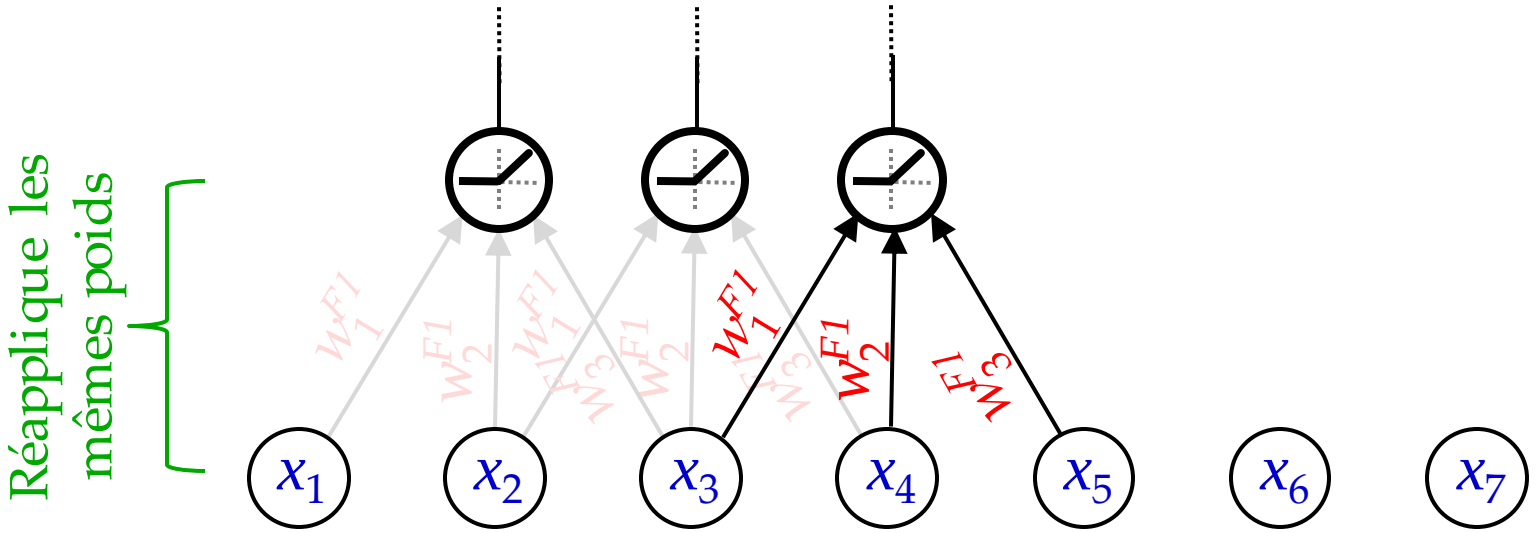
7. *Parameter sharing*

Convolutional Neural Network (CNN)



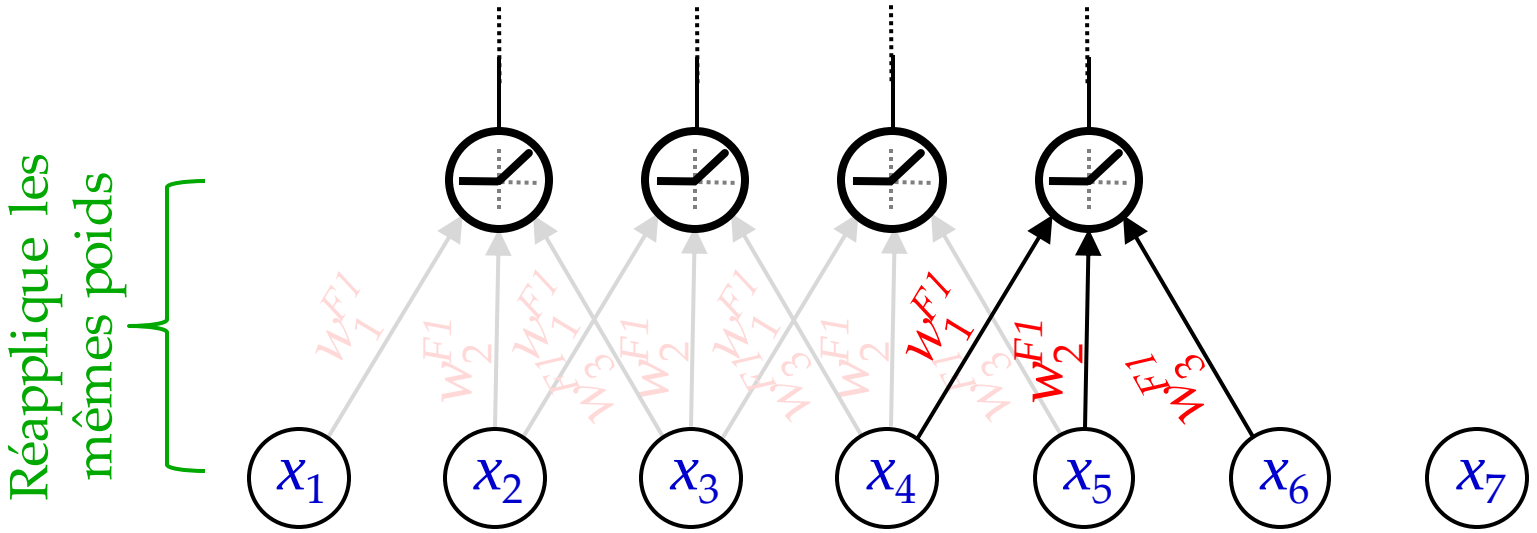
7. *Parameter sharing*

Convolutional Neural Network (CNN)



7. *Parameter sharing*

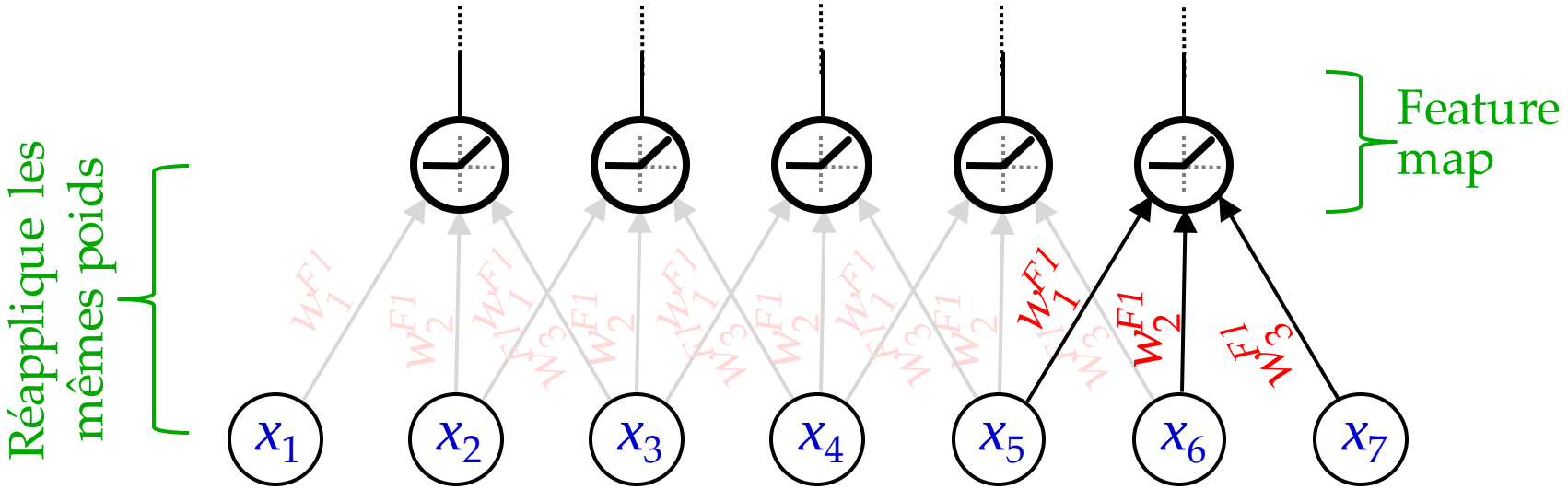
Convolutional Neural Network (CNN)



7. *Parameter sharing*

Convolutional Neural Network (CNN)

- Les poids d'un filtre (ici *F1*) sont utilisés à plusieurs reprises

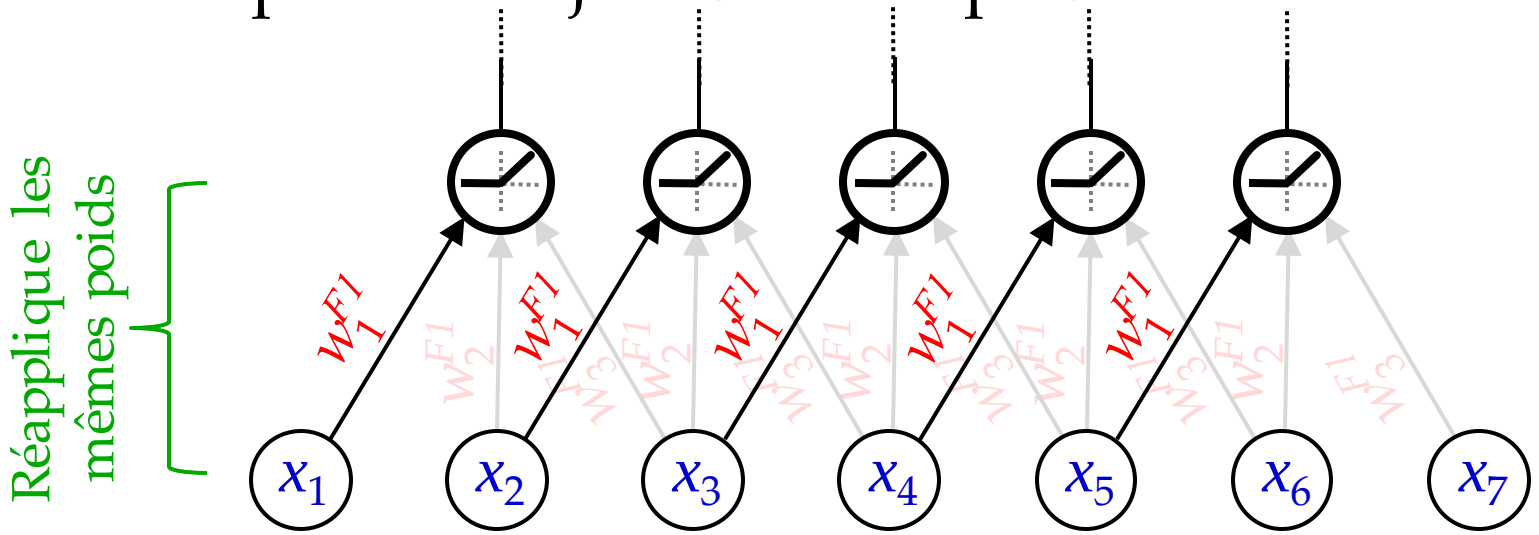


7. *Parameter sharing*

Convolutional Neural Network (CNN)

- Les poids d'un filtre (ici *F1*) sont utilisés à plusieurs reprises
- Équivalent à imposer que des poids d'un *fully connected* soient identiques
- Aussi, forte économie de mémoire
- Exemple de l'injection d'un prior via architecture

(Implémentable via graphe de calcul)



8. Représentation *sparse*

- Pénalité directement sur l'activation d'un neurone, et non ses poids
  - Perte L1 : encourage sparsité sur les poids

Pénalité type L1

$$\underset{\boldsymbol{y} \in \mathbb{R}^m}{\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix}} = \underbrace{\underset{\boldsymbol{\theta} \in \mathbb{R}^{m \times n}}{\begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix}}}_{\text{poids sparse}} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}}$$

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \boxed{\lambda \Omega(\theta)}$$

$$\underset{\boldsymbol{y} \in \mathbb{R}^m}{\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix}} = \underset{\boldsymbol{B} \in \mathbb{R}^{m \times n}}{\begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}} \underbrace{\underset{\boldsymbol{h} \in \mathbb{R}^n}{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}}}_{\text{représentation sparse}}$$

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \boxed{\lambda \Omega(h)}$$

9. Ensemble

- Entraîne un nombre de modèle
- Fait un vote / moyenne des sorties (softmax)
- Fonctionne sous l'hypothèse que les modèles font des erreurs différentes
- Pour architecture identique, variations dues à l'initialisation, mini-batch aléatoire
- Donne un gain de 1-2% « gratuit »
- Souvent découragé pour *benchmarking*

9. Ensemble

Réseau	year	# crops	# modèles	Gain Top-1 (Erreur)	Gain Top-5 (Erreur)
AlexNet	2012		7		2.8% (15.4)
VGGnET	2014	-	2	0.7% (23.7%)	0% (6.8%)
GoogLeNet	2013	144	7	-	1.22% (6.7%)
BatchNorm-Inception	2015	144	6	1.9% (20.1%)	0.92% (4.9%)
Inception-v3	2015	144	4	1.57% (17.2%)	0.62% (3.6%)
Inception v4 + 3x Inception-ResNet-v2	2016	144	4	1.3% (16.5%)	0.6% (3.1%)

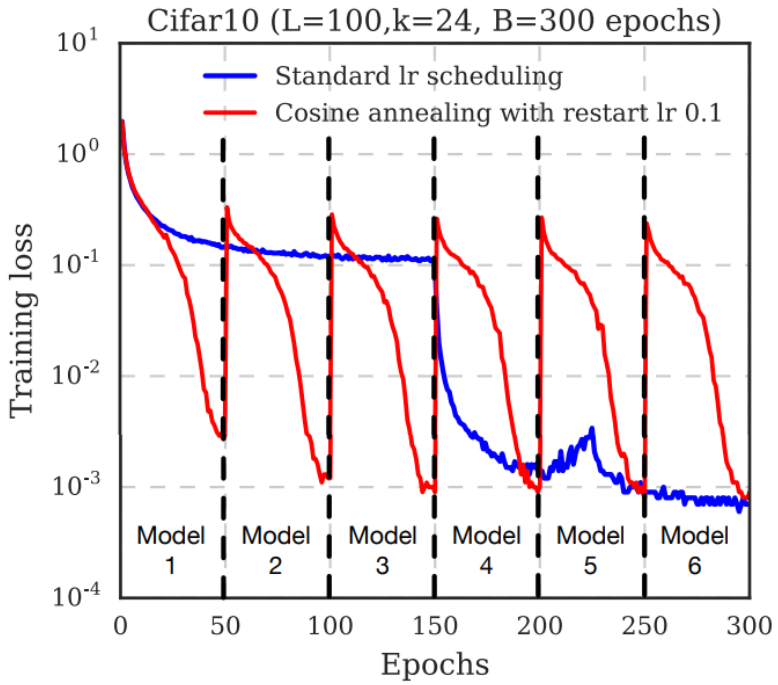
(La valeur entre parenthèse est le taux d'erreur, après ensemble)

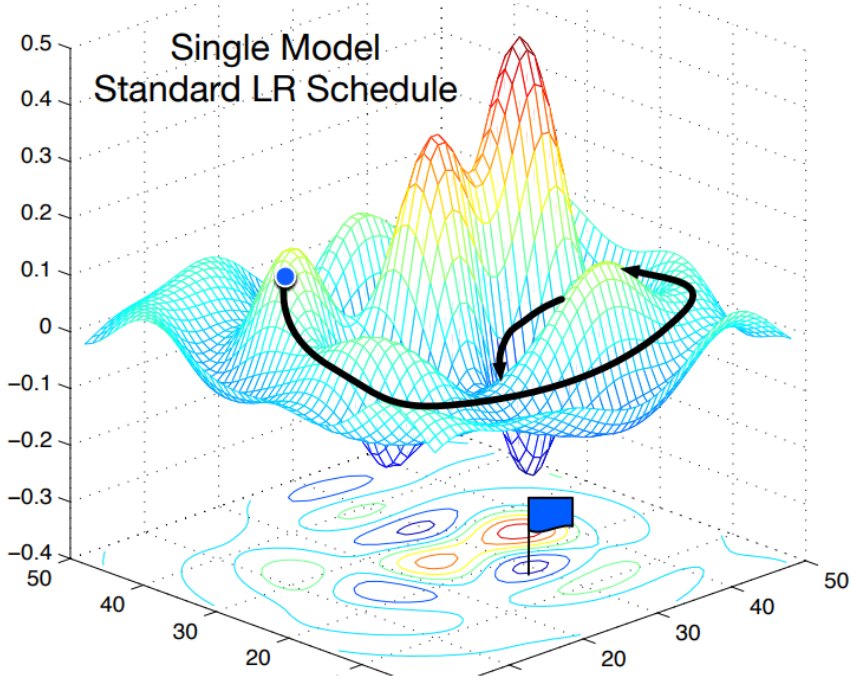
Szegedy et al., Rethinking the Inception Architecture for Computer Vision, CPVR 2015.

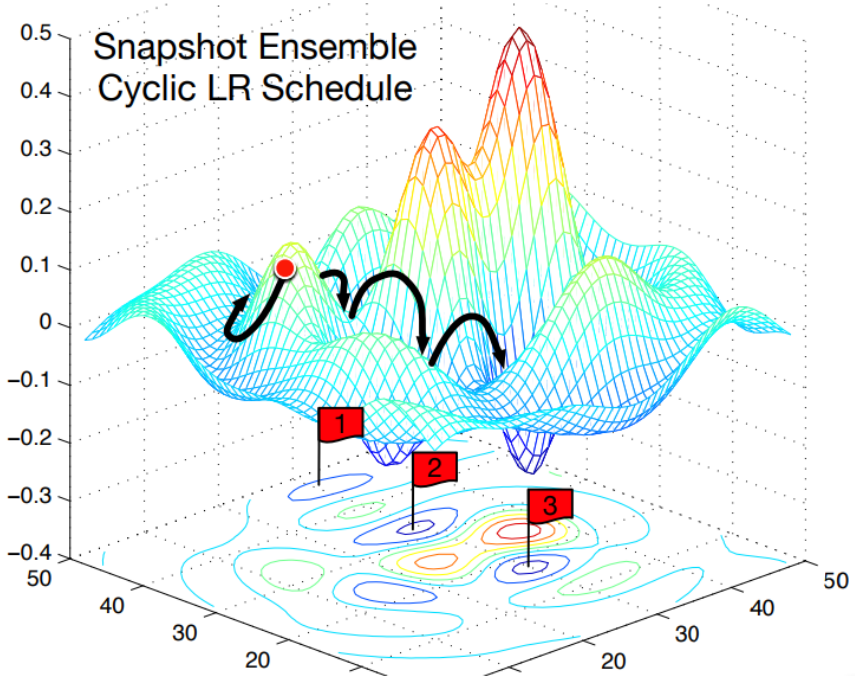
9. Ensemble

Huang et al., Snapshot ensembles: train 1, get M for free, ICLR 2017

- Comme **refaire** du fine-tuning sur le même modèle
- Learning rate continuellement décroissant dans un cycle
- M cycles

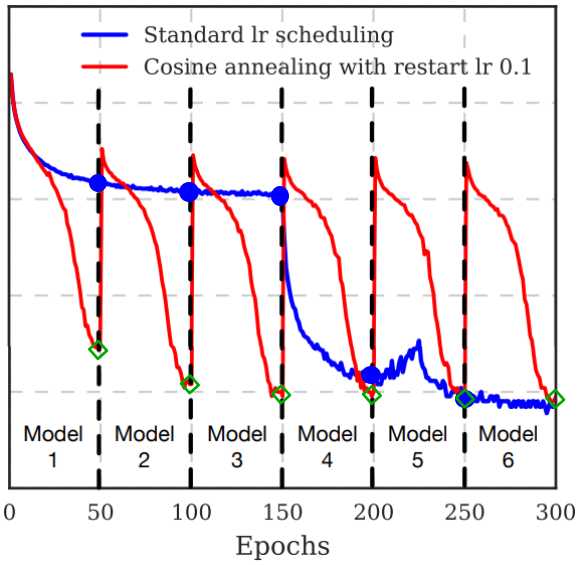






9. Ensemble

Réinitialise au début de chaque cycle



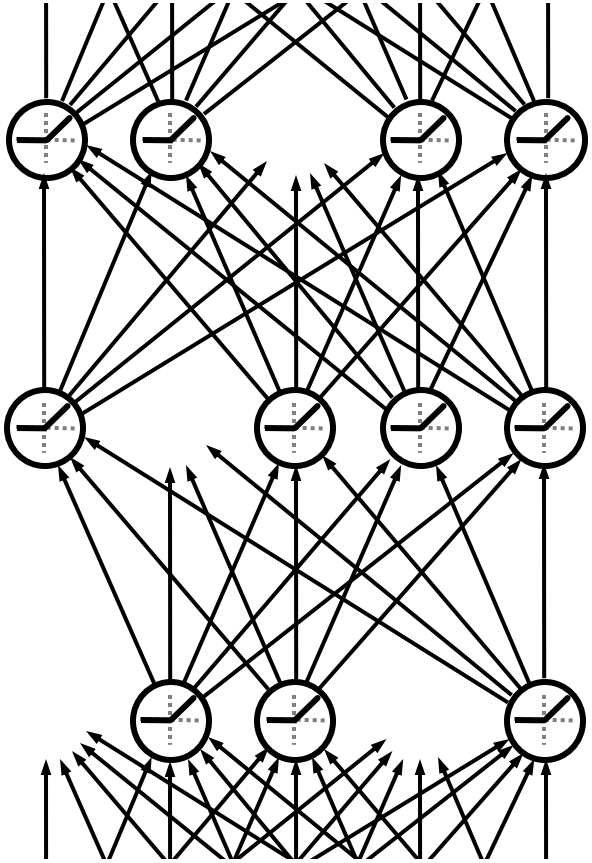
	Method	C10	C100	SVHN	Tiny ImageNet
ResNet-110	Single model	5.52	28.02	1.96	46.50
	NoCycle Snapshot Ensemble	5.49	26.97	1.78	43.69
	SingleCycle Ensembles	6.66	24.54	1.74	42.60
	Snapshot Ensemble ($\alpha_0 = 0.1$)	5.73	25.55	**1.63**	40.54
	Snapshot Ensemble ($\alpha_0 = 0.2$)	**5.32**	**24.19**	1.66	**39.40**
Wide-ResNet-32	Single model	5.43	23.55	1.90	39.63
	Dropout	4.68	22.82	1.81	36.58
	NoCycle Snapshot Ensemble	5.18	22.81	1.81	38.64
	SingleCycle Ensembles	5.95	21.38	1.65	35.53
	Snapshot Ensemble ($\alpha_0 = 0.1$)	**4.41**	**21.26**	1.64	35.45
	Snapshot Ensemble ($\alpha_0 = 0.2$)	4.73	21.56	**1.51**	**32.90**
DenseNet-40	Single model	5.24*	24.42*	1.77	39.09
	Dropout	6.08	25.79	1.79*	39.68
	NoCycle Snapshot Ensemble	5.20	24.63	1.80	38.51
	SingleCycle Ensembles	5.43	22.51	1.87	38.00
	Snapshot Ensemble ($\alpha_0 = 0.1$)	4.99	23.34	**1.64**	37.25
	Snapshot Ensemble ($\alpha_0 = 0.2$)	**4.84**	**21.93**	1.73	**36.61**
DenseNet-100	Single model	3.74*	19.25*	-	-
	Dropout	3.65	18.77	-	-
	NoCycle Snapshot Ensemble	3.80	19.30	-	-
	SingleCycle Ensembles	4.52	18.38		
	Snapshot Ensemble ($\alpha_0 = 0.1$)	3.57	18.12	-	-
	Snapshot Ensemble ($\alpha_0 = 0.2$)	**3.44**	**17.41**	-	-

Huang et al., Snapshot ensembles: train 1, get M for free, ICLR 2017.

10. Dropout

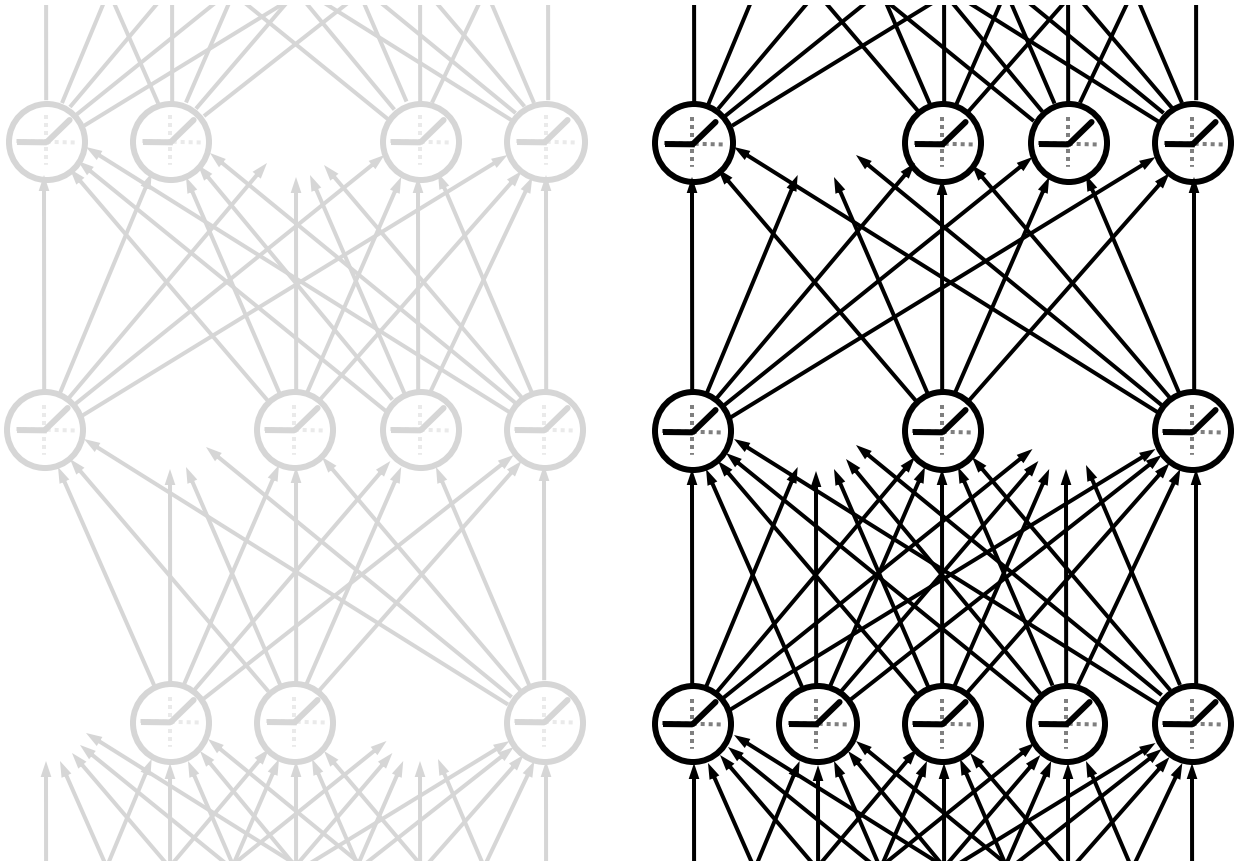
- Retirer aléatoirement et temporairement des neurones du réseau à chaque exemple
- Peut se faire aisément en multipliant leur sortie par 0

10. Dropout



Exemple m

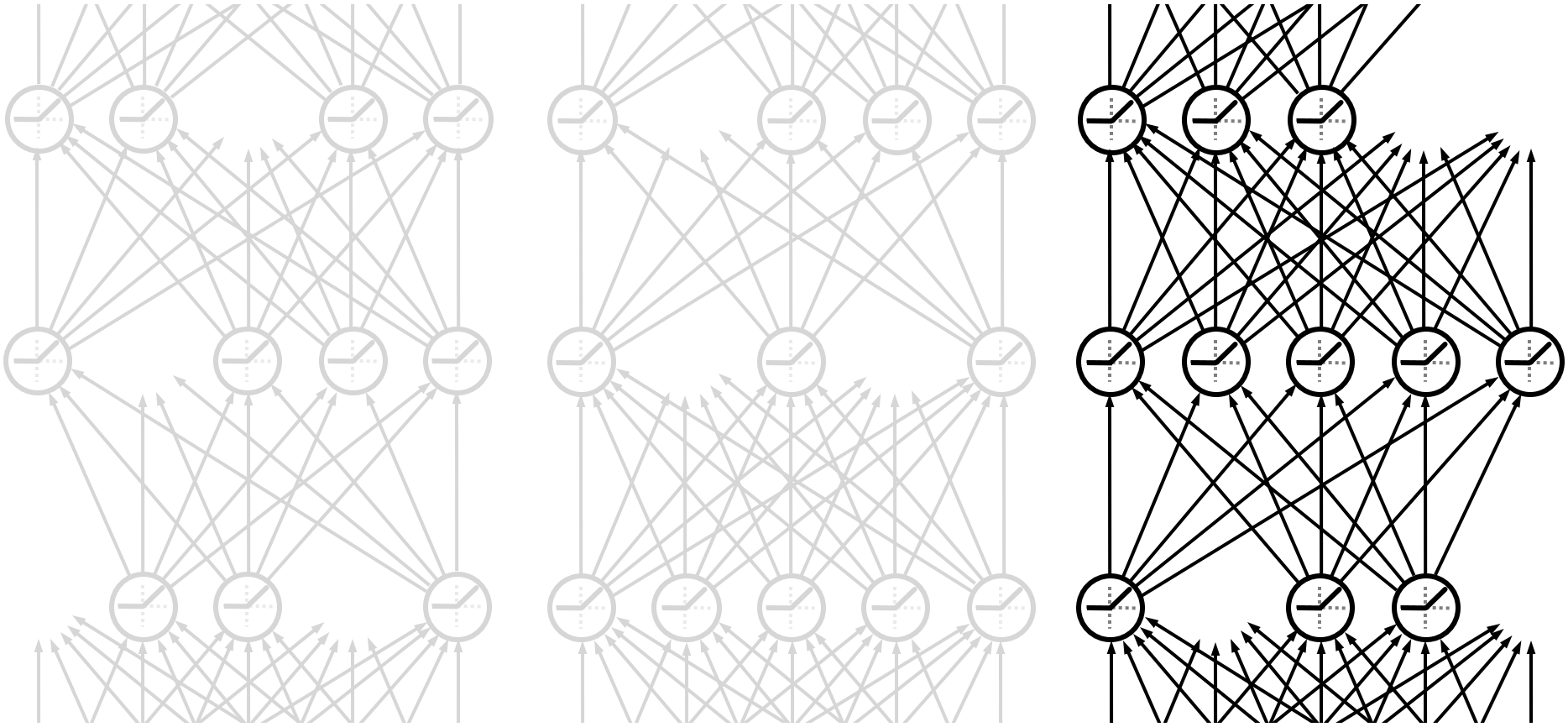
10. Dropout



Exemple *m*

Exemple *m+1*

10. Dropout



Exemple *m*

Exemple *m+1*

Exemple *m+2*

10. Dropout

- Retirer aléatoirement et temporairement des neurones du réseau à chaque exemple
- Peut se faire aisément en multipliant leur sortie par 0
- Était très populaire, un peu moins maintenant
- Peut être vu comme un modèle par ensemble, pour un nombre **exponentiel** de modèle (mais avec une partie de *weight sharing*)

10. Dropout

- Choisir un taux d'inclusion :
  - par couche (p_{inc}=0.8 en entrée, p_{inc}=0.5 ailleurs)
  - pour le réseau entier (sauf sortie)
- Limite la coadaptation des neurones : un *feature* (*hidden unit*) doit être bon en plusieurs contextes de sous-*features*
- Orthogonal à d'autres approches
- Fonctionne sur à peu près toutes les architectures (RBM, DNN, RNN)
- Doit parfois compenser la perte de capacité en augmentation la taille du réseau ☹

10. Dropout

- Surtout utilisé dans la partie *fully-connected* (classifieur)
  - moins dans la partie CNN, feature extraction
- À l'inférence, deux stratégies possibles :

  A On sélectionne 10-20 masques de dropout et l'on fait la moyenne des prédictions

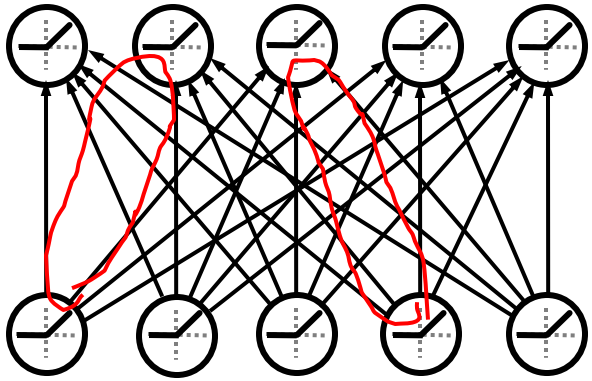
  +commun **B** On conserve tous les neurones, mais on pondère les connections sortantes par p_{inc}

Note : PyTorch fait **B**, mais en pondérant par $1/(1-p_{inc})$ durant l'entrainement.

10. Dropout

- Autres approches :
  - Drop-connect

    Enlève des connections aléatoirement et temporairement

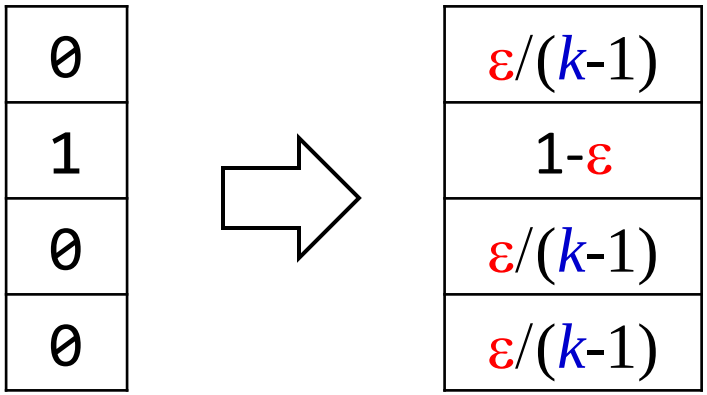


  - Stochastic Depth
    - G. Huang et al., Deep Networks with Stochastic Depth, ECCV,2016.
  - Drop-path
    - FractalNet, CrescendoNet

Autres astuces pour booster les résultats

Label smoothing

- On considère qu'il y a du bruit sur les étiquettes (erreurs d'étiquetage)
- En cas d'erreur d'étiquette, l'entrainement sur le *1-hot vector* est néfaste
- Spécifie un **taux d'erreur ε**
- Cible devient (sur k sorties softmax)



Label smoothing

- Empêche les classificateurs de devenir sur-confiants
- Permet à softmax + crossentropy-loss de converger à une valeur finale (scores ne seront pas poussés infiniment loin)
- Pour en savoir plus :

[1] Szegedy et al., Rethinking the Inception Architecture for Computer Vision, CPVR 2015.

[2] Pereyra et al., Regularizing Neural Networks by Penalizing Confident Output Distributions, ICLR 2017.

Taille de la mini-batch

- Certaines indications que la taille de la batch a une influence sur le type de solutions trouvées
- Petite taille batch = descente gradient bruitée, minimum de J plus plats
- Peu exploité en ce moment, encore à l'étude : restez à l'écoute

Optimal Brain Damage

- Modifier l'architecture durant l'entraînement
- Recherche active de neurones contribuant peu au résultat final
- Conceptuellement, cherche le « gradient » de la délétion sur le coût

$$\frac{\partial J}{\partial \text{délétion}}$$

- Après délétion, on poursuit l'entraînement

Le Cun et al., Optimal Brain Damage, NIPS 1990, cité 3 000+ fois.

Autres

- *Batch Norm* : tend à régulariser les réseaux, rendant parfois inutile le dropout
- Fractional Max-Pooling
- Stochastic Pooling

Verra dans les CNN

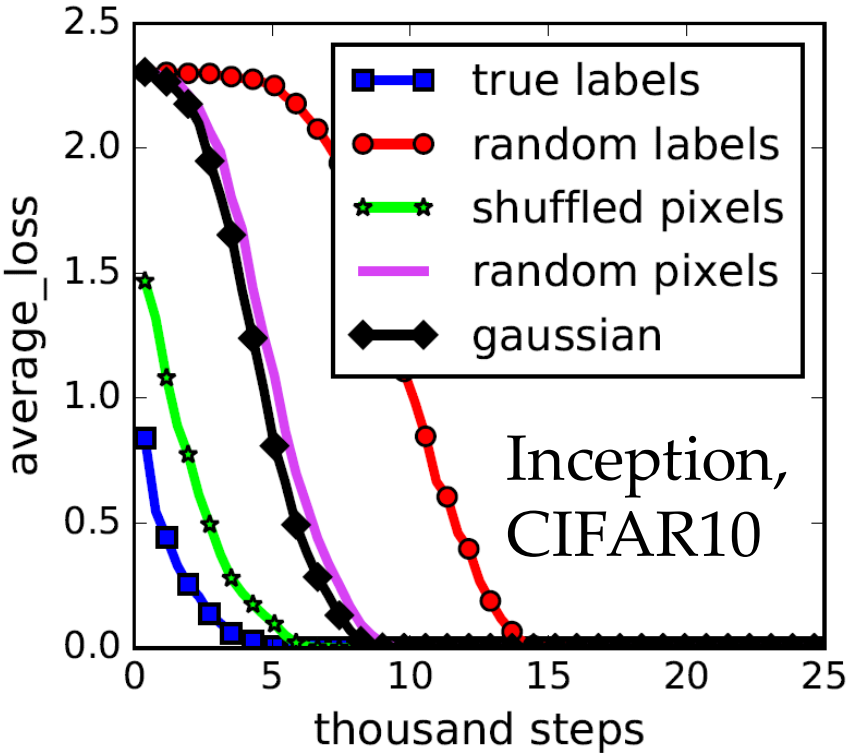
Rethinking generalization

- Démontre qu'un réseau profond peut apprendre facilement par cœur un jeu de donnée, donc *overfit* parfait
  - (preuve quand # de paramètres > # de données)
- Ces même réseaux entraînés **SANS régularisation** arrivent à généraliser assez bien
- Défie la sagesse conventionnelle (que je viens de vous enseigner!)

Zhang et al., Understanding Deep Learning Requires Rethinking Generalization, ICLR 2017.

Rethinking generalization

- Aucun changement sur les hyperparamètres d'entrainement
- Peu importe la randomisation, un réseau de neurone profond a suffisamment **de capacité pour apprendre par cœur** les données : *overfit* parfait
- Dynamique similaire :
  - pente juste avant convergence
  - relativement facile d'apprendre tous les cas



Rethinking generalization

- Régularisation AIDE, mais n'est pas nécessaire
- Émettent l'hypothèse que SGD est un régularisateur implicite

model	# params	random crop	weight decay	train accuracy	test accuracy
CIFAR 10		yes	yes	100.0	89.05
Inception	1,649,402	yes	no	100.0	89.31
		no	yes	100.0	86.03
		no	no	100.0	85.75
(fitting random labels)		no	no	100.0	9.78

Rethinking generalization

- Régularisation AIDE, mais n'est pas nécessaire
- Émettent l'hypothèse que SGD est un régularisateur implicite

model	# params	random crop	weight decay	train accuracy	test accuracy
CIFAR 10					
Inception	1,649,402	yes	yes	100.0	89.05
		yes	no	100.0	89.31
		no	yes	100.0	86.03
		no	no	100.0	85.75
(fitting random labels)		no	no	100.0	9.78
Inception w/o BatchNorm	1,649,402	no	yes	100.0	83.00
		no	no	100.0	82.00
(fitting random labels)		no	no	100.0	10.12

- Batch Norm semble offrir une régularisation (3-4%)
- Voir aussi : Mandt et al., Stochastic Gradient Descent as Approximate Bayesian Inference, *JMLR* 2017