

# Compte rendu RDFIA TME 1-2-3

RIABI arij 3702151

## Partie 1 - SIFT

1) Le calcul des gradients  $I_x$ ,  $I_y$  de l'image est effectué à l'aide des filtres de Sobel, qui sont de taille  $3 \times 3$ .

Dans notre cas, chaque filtre est **décomposable en deux vecteurs**  $h_x$ ,  $h_y$ :  $[-1, 0, 1]$ ,  $[1, 2, 1]$ .

2) Cette séparation du filtre est **utile d'un point de vue calculatoire** : on travaille avec deux vecteurs de taille 3, au lieu de travailler avec deux matrices  $3 \times 3$ .

L'un des vecteur s'occupe de **détecter les contours** et l'autre **permet de flouter**.

3) On utilise un masque gaussien pour lisser l'image afin de **réduire le bruit** et donner plus **d'importance** au centre et de **passer graduellement des pixels pris en compte aux pixels ignorés**.

4) L'orientation des gradients est discrétisée pour que l'on puisse créer des **histogrammes comparables** et pour rendre notre modèle **plus robuste aux rotations**.

5) On remplace les descripteurs dont la norme est inférieure à un certain seuil par des vecteurs nuls : ils ne sont **pas intéressants car ils n'ont pas assez de contraste**.

Ensuite, on normalise, seuil, puis re-normalise l'image afin de rendre le descripteur **invariant aux changements de luminosité**.

6) L'avantage du **SIFT** est qu'il est **robuste aux transformations géométriques comme luminosité, de cadrage, d'angle et de zoom**. Ainsi on peut l'utiliser pour comparer des images, même si ces dernières ont été prises dans des **conditions différentes**.

7) Les **images de gradient** décrivent la **variation d'intensité des pixels dans l'image**. L'**histogramme des SIFTs** est **difficile à interpréter**, le nombre de pics dans l'histogramme des SIFTs est à peu près proportionnel au **nombre de parties dans l'image**.

## Partie 2 – DICTIONNAIRE VISUEL

8) Le dictionnaire sert à identifier des "**patterns**" **récurrents** (identifiés dans plusieurs images). De plus, cela peut permettre de **réduire l'espace de représentation**, en particulier si le nombre de mots du dictionnaire est petit (fini). On cherche ensuite à rapprocher chaque SIFT d'un de ces patterns.

9) L'équation (4) correspond à de l'optimisation strictement convexe, on résout donc l'optimisation par annulation du gradient.

On a :

$$\text{grad} = -2 \sum (xi - c)$$

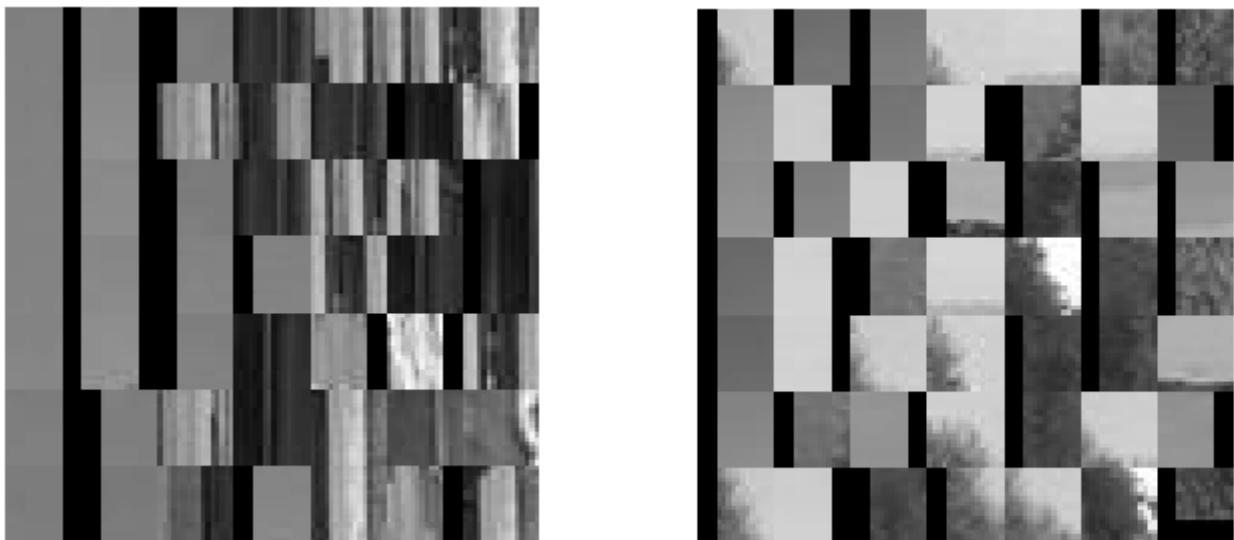
Donc :

$$\text{grad} = 0 \Leftrightarrow \sum (xi - c) = 0 \Leftrightarrow c = \frac{1}{n} \sum (x)$$

10) On peut choisir le nombre de cluster idéal en faisant un “**grid search**”, ce qui consiste à essayer différentes valeurs pour ce paramètre et prendre celle qui convient le mieux. Cependant, le **temps d'exécution** de l'algorithme est élevé et **augmente avec le nombre de clusters**. En pratique, on choisira le **nombre de cluster maximal sans avoir à trop attendre**.

11) **On ne peut pas visualiser directement les éléments du dictionnaire.** On cherche donc les **patchs dont les représentations sont les plus proches de ces vecteurs** pour les visualiser.

12) Pour visualiser le dictionnaire visuel obtenu, on choisit des clusters aléatoires et on affiche des patchs qui leur sont proches.



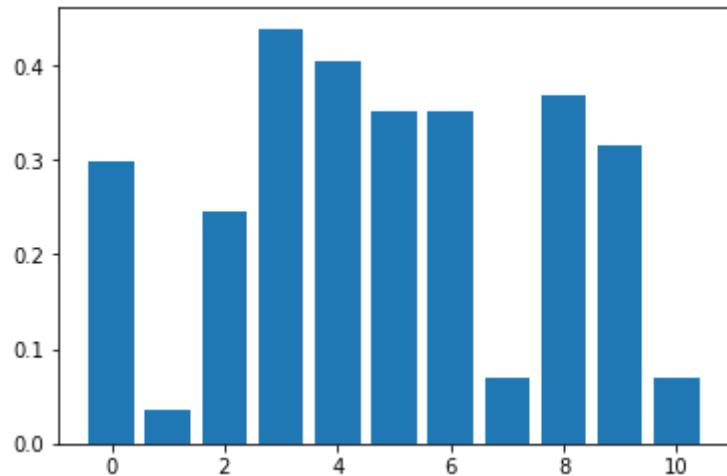
*Régions proches de deux centres de cluster choisis aléatoirement*

Le cluster de gauche correspond à des **lignes verticales**, alors que celui de droite représente à peu près le fait de contenir une **forme arrondie**.

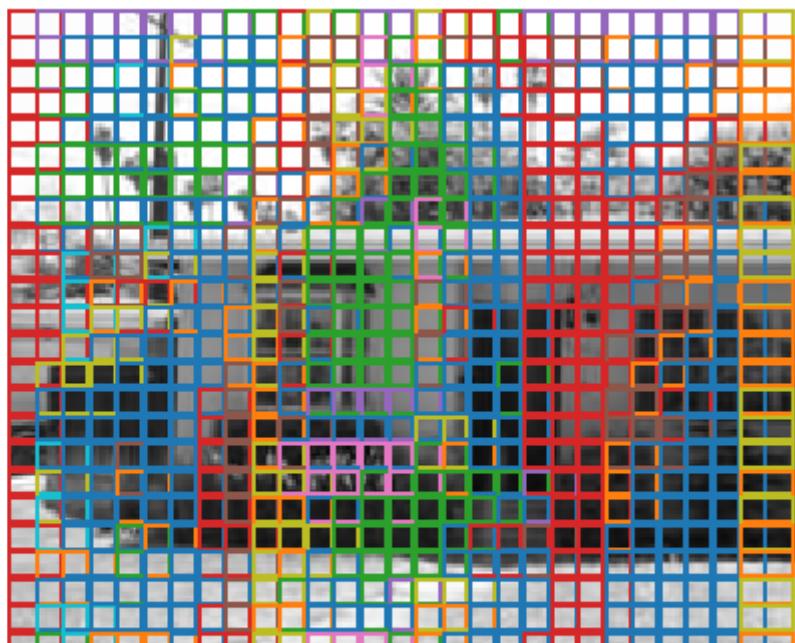
## Partie 3 – BAG OF WORDS

13)  $z$  est une **représentation de notre image**. Ici on utilise un coding au plus proche voisin puis un pooling somme, il s'agit donc de la **fréquence des clusters parmi les patchs**, en rapprochant chaque patch du centre de le plus proche.

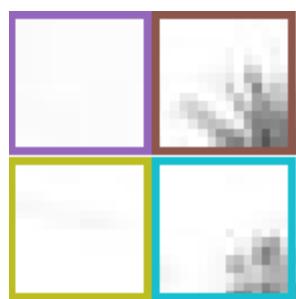
14) Créer le dictionnaire visuel est coûteux à cause de la complexité de l'algorithme k-means. On a donc choisi ici de n'utiliser que 10 clusters afin d'accélérer la vitesse d'exécution.



*Histogramme de la répartition des zones de l'image dans les différents clusters*



*Association de chaque zone de l'image à son cluster*



*Représentation de quelques clusters : le cluster violet représente une zone uniforme, le cluster bleu représente les régions possédant un objet dans le coin inférieur droit*

15) Le codage ‘au plus proche voisin’ est celui permettant l’implémentation la plus efficace, notamment grâce à l’utilisation de **kd-tree** (qui, de plus, offrent la possibilité d’augmenter encore

la rapidité d'exécution au coût d'une perte de précision de la solution trouvée) et il aussi robuste aux petites changements dans l'espace.

Un autre codage pourrait être de faire un “**soft assignement**” : une distribution d'appartenance de l'exemple à toutes les classes ce qui offre une meilleure **expressivité** et une meilleure **stabilité**.

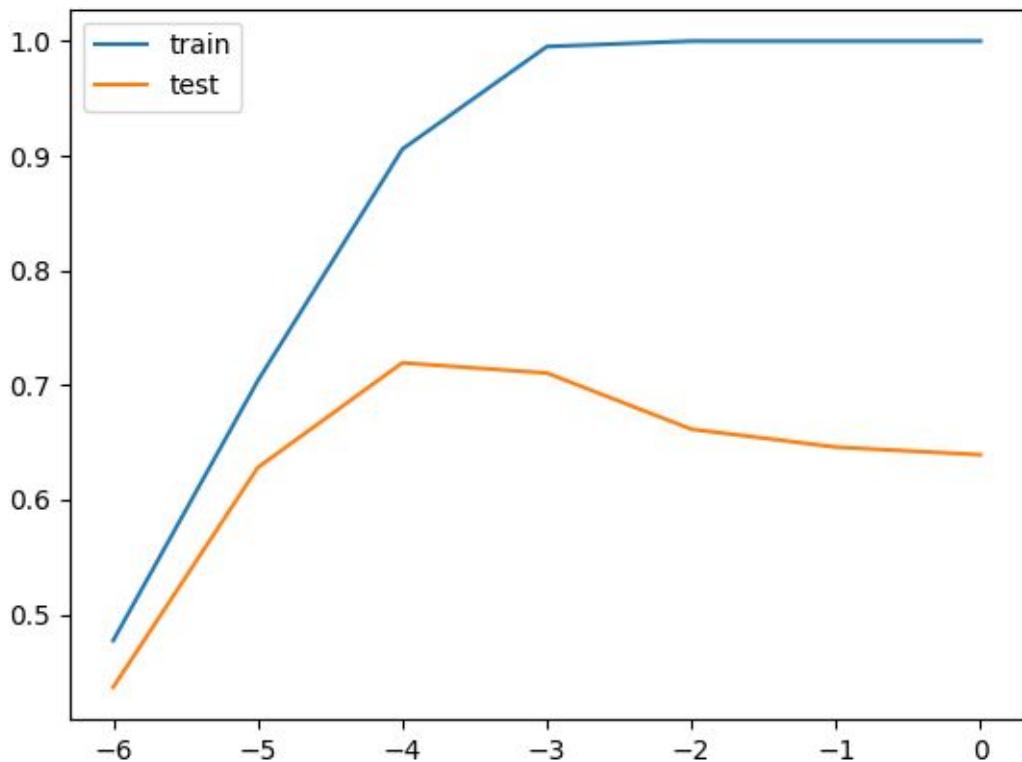
16) Le ‘pooling somme’ est celui traditionnellement utilisé pour du ‘hard coding’. Il a pour avantage de **favoriser les mots visuels apparaissant beaucoup dans l'image**.

Une alternative au pooling somme est le **max pooling** qui résume les activations par la valeur des activations la plus élevée.

17) On utilise la normalisation L2 pour pouvoir comparer les représentations des images, on obtient un **vecteur unitaire** donc pour lequel **seule la direction importe**. Avec une autre norme, comme la norme L1, on n'obtiendrait pas cette propriété.

## Partie 4 – CLASSIFICATION

18) L'erreur en train est inférieure à l'erreur en test. Par **grid search**, la valeur de C optimale trouvée est  $10^{-3}$ . Au delà de ce seuil, le modèle est **moins tolérant aux fautes sur l'ensemble de train** et commence à **sur-apprendre** : la **précision en train augmente** et tend même vers 1 mais la **précision en test diminue**.



*Courbes de précision en fonction de la valeur de l'hyper-paramètre C (échelle logarithmique)*

Une fois la valeur optimale de C trouvée, on réentraîne un SVM sur l'ensemble des données d'apprentissage et de validation et on mesure ses performances sur le jeu de test. On obtient une **précision de 70 %** environ (soit un taux d'erreur de 30%). Cette précision est largement supérieure à l'aléatoire car avec 15 classes, en classifiant aléatoirement les images on aurait une

précision de  $1/15 = 7\%$  (soit un taux d'erreur de 93%). Le classifieur entraîné a donc un **taux d'erreur trois fois inférieur à celui de l'aléatoire**.

19) On ne peut pas utiliser l'ensemble de test pour le grid search car utiliser l'ensemble de test pour l'apprentissage **baisserait les scores obtenus**.

20) Étant donné une nouvelle image on utilise la chaîne de traitement suivante :

1 - Déterminer les **points d'intérêt** en trouvant les angles de l'image (détecteur de **HARRIS**) (on ne l'a pas fait dans ce TP)

2 - On **extrait les caractéristiques** de l'image au niveau des points d'intérêt avec des **SIFTs**. On obtient une représentation en **Bag Of Features**

3 - En utilisant le **dictionnaire de mots visuels** déjà **calculé sur la base de données** d'apprentissage, on détermine la Représentation **Bag Of (Visual) Words** en affectant les sifts de l'image aux mots du dictionnaire dont il est proche. Dans notre cas, on fait du "hard coding" puis du "sum pooling" mais il existe d'autres méthodes

4 - On utilise ensuite notre **SVM préentraîné** afin de déterminer la **classe de l'image**. Le cas multi-classe est traité en "ovr", c'est à dire en "Un Contre Tous"

21) On peut améliorer notre chaîne de traitement de différentes manières,

- Pour l'extraction de features, on pourrait

- déterminer les **points d'intérêt** de l'image

- rendre nos SIFT "**scale invariant**"

- faire un "**soft assignation**" au lieu d'un "hard coding" afin de mieux gérer les points à égale distance de deux mots du dictionnaire par exemple.

- Pour la classification, on pourrait

- essayer **d'autres modèles**, par exemple un "**Random Forest**" car ce type de modèle gère bien le **multi-classe** ou un réseau de neurones.

# compte rendu TP 4-5-6-7

Arij Riabi 3702151

## TP 4-5 :

### 1.1 Jeu de données

1)

- L'ensemble de train sert à entraîner le modèle.
- L'ensemble de validation sert à optimiser les hyper-paramètres pour prendre ceux qui donnent un meilleure score (sans toucher au test) en faisant par exemple un grid search sur l'ensemble des paramètres.
- L'ensemble de test est utilisé à la fin pour évaluer le modèle. **Aucun exemple de test ne doit être vu en train** pour ne pas biaiser les résultats; on cherche à avoir un modèle qui généralise le plus nos données.

2) Un nombre plus grand d'exemples va permettre au modèle de **mieux généraliser** et nous **évite le risque de sur-apprentissage**.

### 1.2 Architecture du réseau (phase forward)

3) **Sans faire d'activation, accumuler des couches linéaires revient à avoir toujours un modèle linéaire** qui n'est pas capable de traiter des données non linéairement séparables (dans le cas d'une classification), c'est la fonction d'activation qui nous permet d'introduire la non-linéarité et d'avoir un modèle plus complexe.

4) on a :

- $n_x$  : Dimensions des entrées. Sur le schéma,  $n_x = 2$ .
- $n_h$  : Nombre de neurones de la couche cachée. Sur le schéma,  $n_h = 4$ .
- $n_y$  : Nombre de classes. Sur le schéma,  $n_y = 2$ .

**$n_x$  et  $n_y$  sont spécifiques aux données et au problème** on ne les choisit pas, mais  $n_h$  est un **hyper-paramètre** qu'on peut optimiser par grid search par exemple.

5)  $\hat{y}$  est la prédiction du modèle et  $y$  est la valeur réelle. **La différence entre  $y$  et  $\hat{y}$  définit l'erreur faite par le modèle.**

6) On utilise une fonction SoftMax en sortie pour transformer la sortie de la dernière couche en nombres compris entre 0 et 1 et dont la somme est égale à 1 : cela donne ainsi une **probabilité par classe** ce qui est utile pour la classification. De plus, à la différence de la fonction 'max', le 'SoftMax' est une **fonction continue et dérivable**, cela permet d'utiliser l'**algorithme de backpropagation**.

7)  $\hat{h} = W_h * x + b_h$   
 $h = \tanh(\hat{h})$   
 $\tilde{y} = W_y * h + b_y$   
 $\hat{y} = \text{SoftMax}(\tilde{y})$

## 1.3 Fonction de coût

8) La MSE et la cross entropy sont des **fonctions de coût convexes** par rapport à leurs entrées.

En appliquant l'algorithme de descente de gradient, on atteindra alors le **minimum global**.

- Pour l'erreur quadratique, la dérivée de la loss est  $\partial L / \partial \hat{y} = 2(y - \hat{y})$ . Au final, pour faire diminuer l'erreur il faut que  $\hat{y}$  soit proche de  $y$ .
- Pour la cross entropy, la dérivée de la loss est  $\partial L / \partial \hat{y}_i = -y_i / \hat{y}_i$ . Pour faire diminuer l'erreur il faut aussi que  $\hat{y}$  soit proche de  $y$ .

9) Le coût **MSE** est le meilleur pour les **problèmes de régression** dans l'ensemble des nombres réels: la fonction est strictement convexe, facile à calculer, et, le fait que le gradient de l'erreur soit proportionnel à l'erreur semble être un bon choix.

Pour les problèmes de classification, la MSE est quasiment impossible à faire converger.

L'**entropie croisée** ou la divergence de Kullback-Leibler sont plus adaptées car elles sont **faites pour mesurer des différences de probabilités**.

## 1.4 Méthode d'apprentissage

10) Selon la quantité de données utilisées pour calculer le gradient on trouve :

-> **Classique**: Le calcul de gradient est réalisé sur tout le dataset pour faire une seule mise à jour, ainsi, cela est peut être trop lent.

->**stochastic gradient descent**: calcule le gradient sur un exemple à la fois avec une mise à jour à chaque fois ce qui rend l'apprentissage plus rapide et permet un apprentissage en ligne. Mais l'algorithme peut avoir du mal à converger.

->**Mini-batch gradient descent**: Le calcul de gradient est fait sur un lot d'exemples ce qui **réduit la variance** des mises à jour des paramètres pour avoir une convergence plus stable.

11) La convergence dépend du choix de  $\eta$  : si il est trop petit, le modèle va être **long à entraîner**, si il est trop grand, le modèle **peut ne pas arriver à converger**. En pratique, on fait souvent **décroître  $\eta$  au fil des itérations**.

12) Dans l'algorithme de backpropagation, en partant de la fin du réseau, chaque couche calcule sa dérivée en fonction de son entrée et de ses poids et transmet à la couche précédente l'erreur à corriger. On obtient une **complexité proportionnelle au nombre de couches** du réseau.

Avec l'approche naïve, on calcule les dérivées de toutes les couches de manière indépendante, on calcule ainsi  $n$  fois la dérivée de la dernière couche ce qui est extrêmement inefficace. Au final, on obtient une **complexité proportionnelle au carré du nombre de couches** du réseau.

**13)** L'algorithme de backpropagation **nécessite que de tout le réseau soit dérivable**. On ne peut par exemple pas utiliser de SVM dedans.

**14)**

$$\begin{aligned}
 l &= -\sum_{i=0}^{n_y} y_i \log \hat{y}_i \\
 &= -\sum_{i=0}^{n_y} y_i \log \frac{\exp \tilde{y}_i}{\sum_k \exp \tilde{y}_k} \\
 &= -\sum_{i=0}^{n_y} y_i \left( \log (\exp \tilde{y}_i) - \log \left( \sum_k \exp \tilde{y}_k \right) \right) \\
 &= -\sum_{i=0}^{n_y} \left( y_i \tilde{y}_i - y_i \log \left( \sum_k \exp \tilde{y}_k \right) \right) \\
 &= -\sum_{i=0}^{n_y} y_i \tilde{y}_i + \sum_{i=0}^{n_y} y_i \log \left( \sum_k \exp \tilde{y}_k \right)
 \end{aligned}$$

Comme  $y$  est un vecteur one-hot on a

$$\begin{aligned}
 l &= -\sum_{i=0}^{n_y} y_i \tilde{y}_i + \log \sum_k \exp \tilde{y}_k \\
 &= -\sum_i y_i \tilde{y}_i + \log \sum_i \exp \tilde{y}_i
 \end{aligned}$$

**15)**  $\forall i \in [1; n_y]$

$$\begin{aligned}
 \frac{\partial l}{\partial \tilde{y}_i} &= -y_i + \frac{\exp \tilde{y}_i}{\sum_j \exp \tilde{y}_j} \\
 &= \text{softmax}(\tilde{y}_i) - y_i \\
 &= \hat{y}_i - y_i
 \end{aligned}$$

**16) \***  $\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}}$

On sait que

$$\begin{aligned}
 \tilde{y}_k &= (W_y h + b_y)_k \\
 &= \sum_q W_{y,kq} h_q + b_{y,k}
 \end{aligned}$$

donc

$$\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \{h_j \text{ si } i = k \text{ 0 sinon.}$$

De plus, d'après

$$\frac{\partial l}{\partial \tilde{y}_k} = (\hat{y}_k - y_k)$$

d'où

$$\frac{\partial l}{\partial W_{y,ij}} = (\hat{y}_i - y_i) \cdot h_j$$

On obtient alors :

$$\nabla_{W_y} l = \nabla_{\tilde{y}} l \cdot h$$

$$\begin{aligned} * \frac{\partial l}{\partial b_{y,j}} &= \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial b_{y,j}} \\ &= \sum_k \frac{\partial l}{\partial \tilde{y}_k} \cdot 1 \\ &= \sum_k \frac{\partial l}{\partial \tilde{y}_k} \end{aligned}$$

d'où  $\nabla_{b_y} l = \nabla_{\tilde{y}} l$

17)

$$\frac{\partial l}{\partial h_i} = \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial h_i}$$

$$\frac{\partial l}{\partial h_k} = \sum_q \frac{\partial l}{\partial \tilde{y}_q} \frac{\partial \tilde{y}_q}{\partial h_k}$$

On sait que

$$\begin{aligned} \tilde{y}_q &= \sum_j W_{y,qj} h_j + b_{y,q} \\ \Rightarrow \frac{\partial \tilde{y}_q}{\partial h_k} &= W_{y,qk} \end{aligned}$$

et

$$h_k = \tanh(\tilde{h}_k)$$

$$\frac{\partial h_k}{\partial \tilde{h}_i} = \{1 - \tanh^2(\tilde{h}_i)\} \text{ si } i = k \text{ 0 sinon.}$$

ce qui donne

$$\begin{aligned}\frac{\partial l}{\partial \tilde{h}_i} &= \sum_q \frac{\partial l}{\partial \tilde{y}_q} \frac{\partial \tilde{y}_q}{\partial h_i} \frac{\partial h_i}{\partial \tilde{h}_i} \\ \Rightarrow \nabla_{\tilde{h}} l &= \nabla_{\tilde{y}} l \cdot \nabla_h \tilde{y} \cdot \nabla_{\tilde{h}} h \\ &= \nabla_{\tilde{y}} l \cdot W_y \cdot (1-h^2)\end{aligned}$$

Soient  $1 \leq i \leq n_h$  et  $1 \leq j \leq n_x$ . On a

$$\begin{aligned}\frac{\partial l}{\partial W_{h,ij}} &= \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{h,ij}} \\ &= \frac{\partial l}{\partial \tilde{h}_i} \frac{\partial h_i}{\partial W_{h,ij}} \\ &= \frac{\partial l}{\partial \tilde{h}_i} \cdot x_j \\ \Rightarrow \nabla_{W_h} l &= \nabla_{\tilde{h}} l \cdot x^\top \\ &= \nabla_{\tilde{y}} l \cdot W_y \cdot (1-h^2) \\ \cdot \text{ Soient } &1 \leq i \leq n_h \\ \frac{\partial l}{\partial b_{h,i}} &= \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_{h,i}} \\ &= \frac{\partial l}{\partial \tilde{h}_i} \cdot 1 \\ &= \frac{\partial l}{\partial \tilde{h}_i} \\ \Rightarrow \nabla_{b_h} l &= \nabla_{\tilde{h}} l\end{aligned}$$

## TP 6-7 :Réseaux convolutionnels pour l'image

1) Convolution:

- taille de sortie  $(x+2p-k)/s + 1, (y+2p-k)/s + 1$
- nb de poids à apprendre:  $k * k * z$
- nb de poids qu'il aurait fallu apprendre si une couche fully-connected devait produire une sortie de la même taille:  $x * y * z * ((x+2p-k) // s + 1) * ((y+2p-k) // s + 1)$

2) Avec des convolutions, on apprend **moins de poids** tout en **préservant la localité de l'information** dans l'image qu'avec des couches fully-connected.

La limite des convolutions est que le nombre de poids est quadratique avec la taille du kernel et surtout qu'elles n'ont accès **qu'aux informations locales**.

3) L'intérêt du **pooling spatial** est qu'on réduit la dimension de l'image en résumant son information et qu'on devient **légèrement invariant aux translations**.

4) Les **couches de convolution apprises peuvent être appliquées quelle que soit la taille des images en entrée**. C'est aussi le cas pour les couches d'activation et de pooling spatial mais ce n'est pas le cas pour les couches fully-connected. On peut donc utiliser toutes les premières couches du réseau tant qu'on n'utilise pas de couche fully-connected.

5) Une couche fully-connected ayant une sortie de taille  $n$  peut être représentée par  $n$  convolutions ayant une taille de kernel égale à la taille de l'entrée et un padding nul (et un stride quelconque).

6) Si toutes les couches sont des convolutions, on peut appliquer le réseau de neurones sur des images de taille quelconques. **La taille de la sortie du réseau dépend alors de la taille des images** sur lesquelles on l'utilise. Cependant, si on applique le réseau sur des images plus petites la sortie sera vide car les images ne seront pas assez grandes pour appliquer le filtre appris. Si les images sont plus grandes, la sortie du réseau ne sera plus qu'un simple nombre mais un tableau 2D.

7) Sur la sortie de la première couche de convolution, la **taille du champ récepteur des pixels est égale à la taille du filtre** de la convolution. En appliquant une deuxième convolution immédiatement, la taille du champ récepteur des pixels est  $(k_2 - 1) * s_1 + k_1$ . La taille du champ récepteur augmente avec la profondeur, cela signifie que **les premières couches latentes ont des features de bas niveau et les couches de la fin possèdent des features de plus haut niveau**.

8) Pour qu'une convolution conserve la taille des images, on utilise un stride de 1 et un padding en fonction de la taille du kernel  $k$  :

- Avec  $k$  impair, un **padding de taille  $(k - 1) / 2$**  convient.
- Avec  $k$  pair, il faudrait aussi utiliser un padding de taille  $(k - 1) / 2$  mais cela donne une taille de padding non entière. **C'est pour cela qu'en pratique on utilise des kernel de taille impaire.**

9) Pour qu'un max pooling réduise les dimensions spatiales de 2, on utilise un stride de 2, un kernel de taille 2 et un padding nul.

10)

Couche	Taille de sortie	Nombre de poids
entrée	32*32*3	0
conv1	32*32*32	$5*5*3*32 = 2400$
pool1	16*16*32	0
conv2	16*16*64	$5*5*32*64 = 51200$
pool2	8*8*64	0
conv3	8*8*64	$5*5*64*64 = 102400$
pool3	4*4*64	0
fc4	1000	$4*4*64*1000 = 1024000$
fc5	10	$1000*10 = 10000$
Nombre de paramètres total		1 190 000

11) Le nombre total de poids à apprendre est de 1 190 000, alors que dans notre base d'apprentissage on a 50 000 images ce qui est trop petit par rapport au nombre de poids ainsi on a un **risque de sur-apprentissage**.

12) Pour l'approche **BoW+SVM** utilisée précédemment avec un dictionnaire de 1000 SIFT de taille 128, on apprend de l'ordre de 128 000 paramètres. Il y a plusieurs hyperparamètres à fixer comme la taille du dictionnaire appris, la constante 'C' du SVM et le type de SIFT utilisé. Avec le **réseau convolutionnel** il y a **10 fois plus de poids à apprendre**. L'architecture neuronale possède aussi plusieurs hyperparamètres, mais on a un **apprentissage de bout-en-bout** ce qui permet **d'apprendre les extractions de caractéristiques** à effectuer.

14) Model.eval() change le fonctionnement de certaines couches ou les désactive comme les couches de dropout ou de batchNorm. Par ailleurs, l'erreur affichée en train correspond à une **moyenne effectuée sur les différents batch**, ce n'est donc pas l'erreur obtenue à la fin de l'époque (c'est le cas en test).

16) La convergence dépend du choix de  $\eta$  : si il est trop petit, le modèle va être **long à entraîner**, si il est trop grand, le modèle **peut ne pas arriver à converger**. En pratique, on fait souvent **décroître  $\eta$  au fil des itérations**.

Selon la quantité des données utilisées pour la calculer le gradient on trouve :

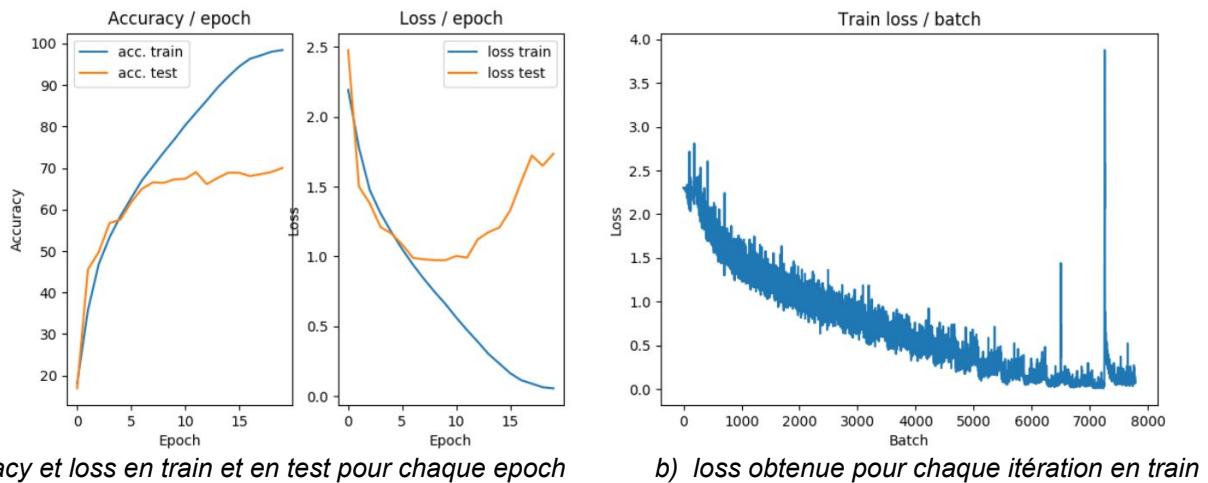
- **Batch gradient descent:** Le calcul du gradient est réalisé sur tout le dataset pour faire une seule mise à jour précise ainsi cela peut être trop lent.

- **stochastic gradient descent:** calcule le gradient sur un exemple à la fois avec une mise à jour à chaque fois ce que rend l'apprentissage plus rapide et permet un apprentissage en ligne. Mais le modèle peut avoir du mal à converger au minimum global si le pas de gradient ne décroît pas au fil des itérations.

- **Mini-batch gradient descent:** Le calcul de gradient est fait sur un lot d'exemples ce qui réduit la variance des mises à jour des paramètres pour avoir une convergence plus stable.

17) Au début de la première époque, le modèle n'a pas encore été entraîné, l'erreur obtenue est définie par l'**initialisation aléatoire** des poids du réseau. Cela fournit une sorte de **baseline**: en s'entraînant, le modèle doit faire baisser cette erreur, sinon cela signifie qu'il n'arrive pas à s'entraîner.

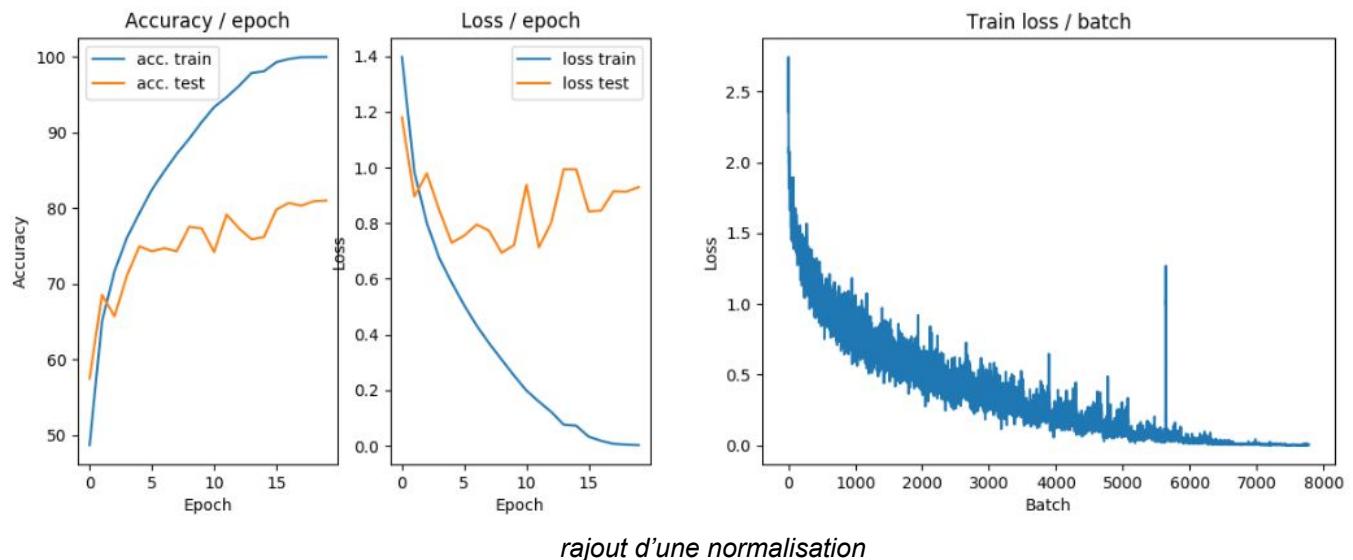
18)



En train, la loss diminue globalement vers 0, cela démontre que le réseau arrive à s'entraîner. Le fait que la loss oscille est dû à la variance des batchs: certains batchs sont plus faciles à traiter que d'autres.

Pendant les premières époques les loss en train et en test baissent, mais à partir d'une certaine époque la loss en train continue à baisser alors que la **loss en test augmente**. On fait face à une **situation de sur-apprentissage**. Sur les courbes d'accuracy, le phénomène observé n'est pas le même: quand la loss en test se met à augmenter, l'**accuracy en test se contente de stagner**, mais cela reste un **problème à corriger**.

19)



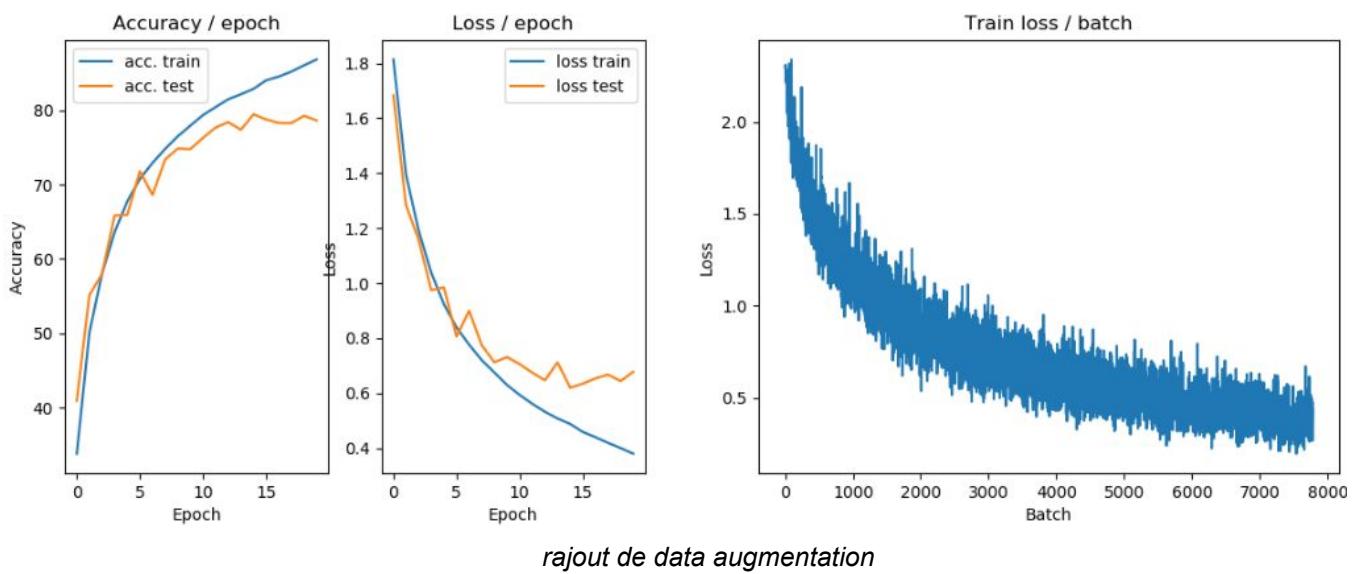
Normaliser les données par standardisation améliore le **conditionnement de l'apprentissage**. En pratique, cela améliore les performances obtenues en train et en test, la **précision en test passe**

**de 65% à 80%** environ. Les courbes en train et en test n'ont pas la même tendance: nous sommes encore dans un cas de sur-apprentissage.

20) Il ne faut pas apprendre les pré-traitements sur le dataset de test, car sinon les performances obtenues sont biaisées.

21) Il existe d'autres méthodes de normalisation. On peut borner l'intensité par un minimum et un maximum. On peut également utiliser un whitening: après avoir centré les données par soustraction de la moyenne, on calcule la matrice de covariance, puis on applique une PCA à cette matrice. Le nombre de composantes choisies permet de réduire la dimension, et donc de garder seulement les données qui contiennent le plus de variance.

22)



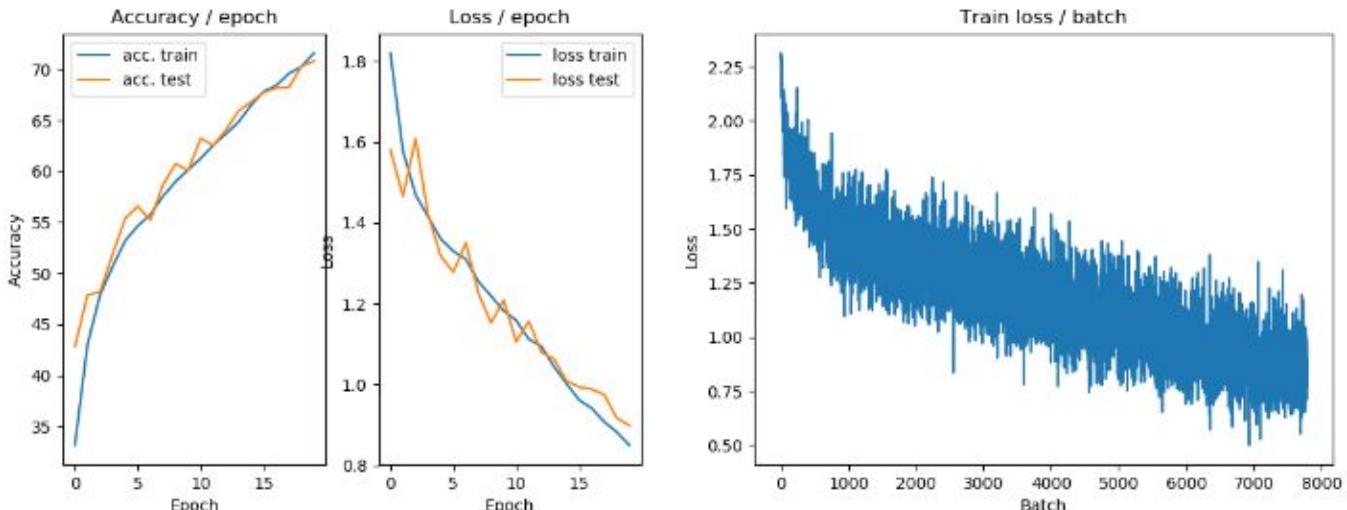
En rajoutant de la data augmentation, le sur-apprentissage arrive plus tardivement et est moins prononcé: les performances diffèrent moins entre le jeu de train et le jeu de test.

23) Certaines images ont une sémantique invariante par symétrie, mais ce n'est pas le cas de toutes les images. Par exemple, les images représentant des lettres ou des chiffres perdent tout leur sens si on prend leur symétrique.

24) Il est important que le dataset d'origine soit suffisamment représentatif pour que les images générées soient utiles à l'apprentissage. On n'oublie pas le risque de perte de sémantique mentionné au dessus, puisque certaines transformations peuvent ne pas être cohérentes et on ne retrouverait pas dans des données réelles.

25) On peut ajouter du bruit à l'image, faire des rotations d'angle alpha, faire des changements de luminosité, de saturation ou faire une suppression partielle d'une partie de l'image (cut out)

26)



*rajout d'un momentum et d'un learning rate scheduler*

On modifie l'optimiseur en rajoutant un **momentum** de 0.9 et en ajouter un **learning rate scheduler** avec une décroissance exponentielle de coefficient 0.95. Cela **améliore la stabilité de l'apprentissage**: les performances s'améliorent de manière plus constante. Au bout de 20 epochs, il n'y a toujours pas de sur-apprentissage mais le modèle n'a pas fini de converger. Cela est sûrement dû au fait que le learning rate scheduler permet de faire des optimisations plus fines mais **ralentit l'entraînement**.

*Note: Pour des raisons de temps de calcul, on ne cherche pas à obtenir les meilleures performances pour chaque expérience réalisée dans ce TP. Par la suite, entraîner le réseau sur 20 epoch seulement permettra néanmoins de voir certains impacts des ajouts effectués. Les expériences à réaliser étant les même que celles des autres groupes de TP, nous nous évitons cette peine et utilisons leurs résultats. Le modèle final sera lui entraîné plus longtemps afin de comparer l'ajout de toutes les modifications au modèle initial qui convergeait en 20 epochs, cela démontre notre capacité à mener les expériences.*

27) Le momentum est une méthode pour accélérer la convergence et réduire les oscillations quand les données ne varient pas de la même façon selon les axes. On l'effectue en ajoutant le gradient calculé à l'étape précédente pondéré par une valeur autour de 0.9 au nouveau gradient avant la mise à jour. Cela permet d'augmenter le pas quand on a la même direction et de **réduire le pas quand la direction change**. Ce qui accélère l'apprentissage.

Le learning rate scheduler permet de se déplacer rapidement au début afin d'**approcher rapidement une solution convenable**. Une fois que le modèle a convergé, baisser le learning rate permet de continuer l'entraînement en réalisant des **améliorations plus fines**.

Ces deux méthodes améliorent la vitesse d'apprentissage du modèle mais elles n'ont aucun impact sur le sur-apprentissage.

28) Plusieurs variantes de SGD existent, par exemple:

->**Adam** : cette méthode se base sur un pas d'apprentissage par paramètre et utilise la moyenne et la variance pour adapter le pas d'apprentissage; on divise  $\eta$  par la variance et à la place de multiplier par le gradient on multiplie par la moyenne des gradients.

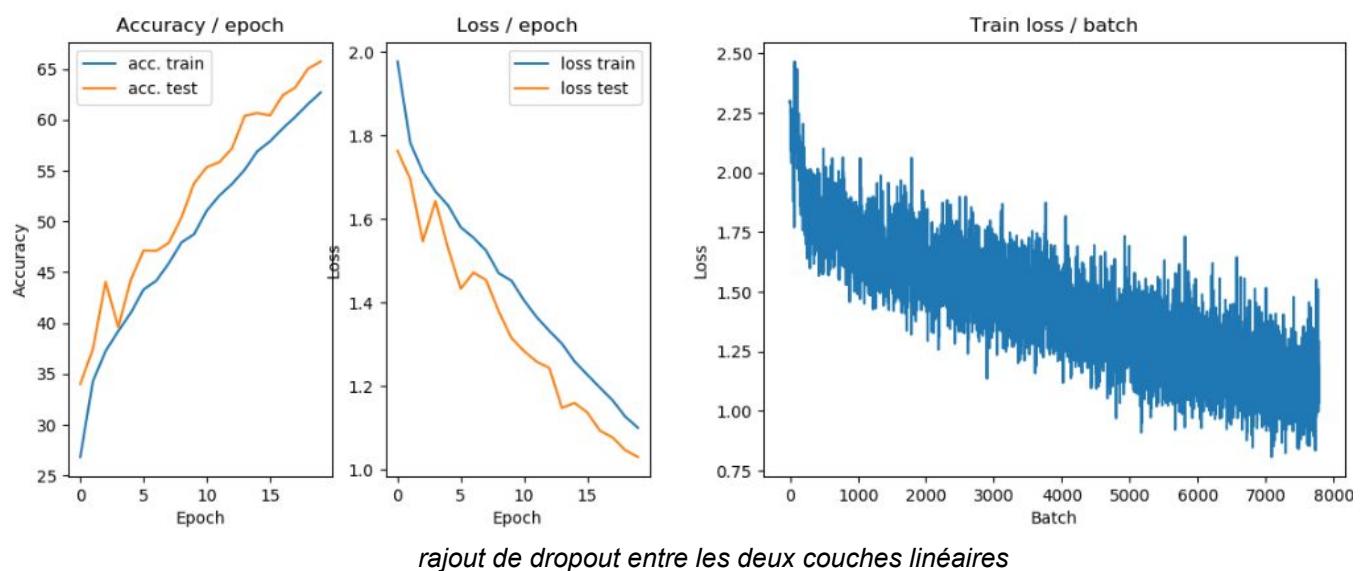
->**AdaMax** : Une variante d'Adam basée sur la norme de l'infini; à la place d'utiliser la norme l2 de la variance on calcule le max entre la variance estimée et la norme des gradients ce qui donne l'avantage d'être moins sensible au choix des hyper-paramètres .

->**Nadam** : C'est une combinaison entre Adam et NAG; on calcule les deux premiers moments comme dans Adam, mais quand on fait la mise à jour, on ajoute à la moyenne et aux gradients le gradient lui même multiplié par son pas d'apprentissage.

->**AMSGrad** : une amélioration de ADAM ; on prend la valeur maximale entre la variance estimée à cette étape et le maximum des variances précédentes.

Dans la suite de ce TP, on utilisera Adam.

29)



Il y a **beaucoup de poids** entre les deux couches linéaires du réseau, il y a donc un **grand risque de sur-apprentissage**, un ajout de régularisation est donc opportun. En rajoutant une couche de **dropout** (avec un facteur 0.5) entre les deux couches linéaires, nous arrivons dans une situation inverse au sur-apprentissage au sens où **les performances obtenues en test sont meilleures que celles obtenues en train**. Cela s'explique en partie par le fait que pour le jeu de données d'apprentissage, le réseau utilise le dropout mais pas pour le jeu de test. Mais cela reste à relativiser par le fait que les performances sont mesurées à la fin d'une epoch en test mais pendant l'epoch en train.

Les performances du modèle au bout de 20 epochs sont **inférieures au modèle sans dropout**. En effet, un réseau avec du dropout nécessite **plus d'epochs pour s'entraîner**, mais pour une même architecture, les epochs sont plus rapides à calculer car il y a moins de poids.

30) La régularisation est un **processus de pénalisation envers la complexité du modèle**. Cela permet de réduire le risque de sur-apprentissage.

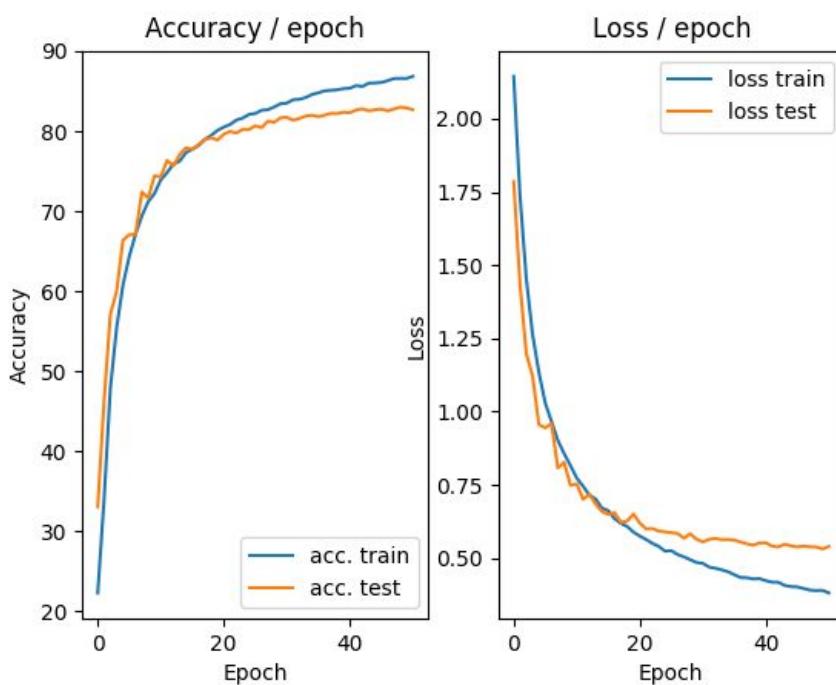
31) Le drop out consiste à "désactiver" certaines unités de notre réseau, et ainsi à forcer l'apprentissage des autres. Le gradient a tendance au cours de l'apprentissage à de plus en plus "prendre le même chemin", et ainsi donner plus d'importance à certaines parties du réseau plutôt qu'à d'autres. En désactivant ces parties le gradient devra réutiliser les autres parties du réseau.

Lors de l'utilisation de dropout le réseau perd de l'expressivité, il peut être utile d'augmenter la taille des couches du réseau de neurones.

32) Le paramètre  $p$  correspond à la probabilité d'un neurone d'être désactivé. Si cette probabilité est trop forte, nous risquons de sous-apprendre. En revanche, si elle est trop faible nous perdons les bénéfices du dropout.

33) En mode d'évaluation, le dropout est désactivé. Pour pallier au fait que l'intensité de la sortie est proportionnelle au nombre de neurones (non désactivés), on multiplie les sorties du réseau par le coefficient de dropout. Cela permet d'obtenir une espérance d'intensité égale à celle du mode d'apprentissage.

34)



*rajout de batch norm après chaque convolution (50 epochs)*

Rajouter de **la normalisation par batch accélère l'entraînement du modèle**: en 20 epochs, l'accuracy passe de 65% à 80%.

## Conclusion

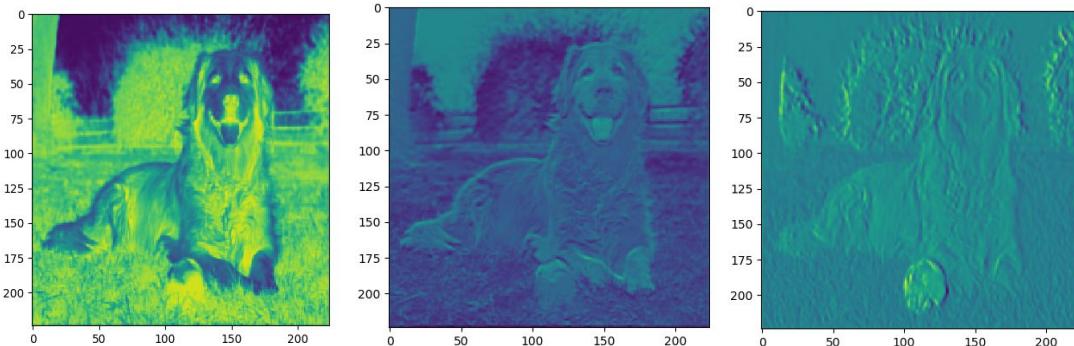
- L'architecture initiale du réseau fait face à un problème de sur-apprentissage et obtient une accuracy de 70%.
- En normalisant les données, le modèle obtient une accuracy de 80% mais les performances en train et en test restent très différentes.
- Avec de l'augmentation de données, bien que le nombre de poids à apprendre reste supérieur au nombre d'images de notre dataset, les courbes en train et en test obtiennent la même tendance, le sur-apprentissage est beaucoup moins marqué. Étonnamment, en pratique cela n'améliore pas vraiment les performances en test.
- Changer l'optimiseur et ajouter un learning rate scheduler n'améliore pas les performances en test.

- L'ajout de dropout n'améliore pas non plus les performances du modèle.
- En rajoutant des batch norm, l'entraînement du réseau est plus rapide mais les performances obtenues ne sont pas meilleures.

Au final, le meilleur modèle obtenu a une accuracy de 80% environ et une accuracy de 99% environ. Les dernières améliorations apportées n'améliorent pas les performances du modèle. Une piste envisageable pour obtenir de meilleures performances serait de travailler sur un dataset plus grand comme ImageNet.

## TP 8 - Classification d'images par CNN

- 1) Le nombre de poids d'une couche fully connected est le produit de la dimension d'entrée et de la dimension de sortie (en négligeant les biais). Cela nous donne  $7*7*4096 + 4096*4096 + 4096*1000 = \textbf{124M de paramètres}$  environ dans le réseau VGG16, cela représente 90% des poids du réseau (total=138 M)
- 2) La sortie du réseau est un vecteur de taille 1000, cela correspond à la **probabilité d'appartenance à chacune des 1000 classes** pour lesquelles ce réseau a été entraîné.
- 3) Le réseau fonctionne très bien si l'image en entrée correspond à une des classes apprises, mais il n'a **aucune chance de réussir pour une autre classe**. Le réseau se trompe parfois, surtout quand on lui présente des images de style différent.
- 4) On visualise l'**activation obtenue par l'application de différents filtres sur l'image**. La première couche est comparable aux filtres sobels par exemple au sens où elle donne des informations de bas niveau. L'utilité d'une carte particulière est difficilement interprétable par un humain. Les cartes correspondent à une mise en valeur de différentes features par le réseau.



- 5) Le dataset 15Scene est **beaucoup trop petit pour permettre d'apprendre un réseau contenant des millions de paramètres comme VGG**. Cela cause un risque de sur-apprentissage du dataset par le réseau.
- 6) On essaye ici de se servir de la partie feature extraction de VGG et de seulement réapprendre la partie finale. L'extraction de caractéristiques générales d'une image est la même quelle que soit la tâche à effectuer. On peut donc **se servir des très bonnes performances du réseau VGG**.
- 7) Le problème avec cette méthode est que VGG a été entraîné avec un type particulier d'image, si on lui fournit des images avec des **caractéristiques nouvelles**, par exemple des images en noir et blanc, elles ne pourront **pas être détectées**. Si les images possèdent une **autre dynamique** (par exemple si elles sont plus sombres), le réseau aura plus de mal à détecter les caractéristiques. Il faut donc que les deux tâches ne soient pas trop différentes.
- 8) Plus on s'éloigne de l'entrée et plus les feature maps sont de haut niveau, les premières convolutions détectent par exemple des contours, et les dernières des parties de chien ou de chat.
- 9) Les images de 15Scene sont en noir et blanc, pour pouvoir les faire passer par VGG, on duplique 3 fois leur channel.
- 10) En utilisant un SVM apprenant à classifier les images à partir d la feature map, on obtient un **taux d'erreur de 11%**. Dans le TP précédent, en utilisant un SVM sur les BoW, on avait obtenu un taux de 30%. Les résultats obtenus avec VGG sont 3 fois meilleurs, cela montre que les caractéristiques extraites par VGG sont de meilleure qualité.
- 11) **Utiliser un SVM bloque le fine-tuning**, on pourrait le remplacer simplement par une autre couche linéaire à la fin du réseau.

12) Remplacer le SVM par une simple couche neuronale permet la backpropagation des gradients dans VGG. Pour des raisons de temps de calcul, nous ne souhaitons pas fine-tuner les poids de VGG, cela ne peut donc pas nous apporter un gain de performances et nous fait perdre l'utilisation de l'hyperparamètre C.

Le paramètre C est très important pour les SVM, chercher une bonne valeur pour ce paramètre est donc pertinent. En essayant des valeurs de 0.001, 0.01, 0.1, .5, 1 et 10 le meilleur score en test est C=1 (comme précédemment).

Pour extraire les features de VGG, on utilise actuellement la partie ‘feature’ et le début de la partie ‘classification’. D’après le nom des parties du réseau de neurones, il est légitime de penser que pour classifier de nouvelles classes d’objets, il ne faut pas se servir de la partie ‘classification’ de VGG. On réalise donc une nouvelle fois les expériences en utilisant **seulement la partie ‘feature’**. Les vecteurs de caractéristique extraits sont plus gros que précédemment et sont extraits plus rapidement étant donné qu’on applique moins de couches de VGG. Les performances obtenues sont **nettement meilleures** : on atteint un **taux d’erreur de 1%** seulement, c’est à dire une différence d’un ordre de grandeur. **Utiliser la partie ‘feature’ uniquement est donc primordial.**

## TP 9 - Visualisation de réseaux de neurones

### A - Cartes de saillance

1) On obtient des cartes de saillance en affichant simplement la norme de la dérivée de la sortie du réseau de neurones par rapport à l'image. **Les cartes de saillance sont activées au niveau d’éléments caractéristiques** des objets de l’image. Par exemple au niveau de la tête des chiens ou des bottes de foin.



2) Les activations de la carte de saillance **ne recouvrent pas totalement les zones** qu'un humain caractérisait comme importantes (ex: les autres bottes de foin). Cette technique souffre d'une **forte**

**variance**: en changeant l'image, les activations peuvent changer fortement. Finalement, cette technique suppose la linéarité du réseau : le fait que le gradient soit élevé ne signifie pas forcément que la zone est importante.

3) Cette technique peut servir à la **détection d'objets**, à la **segmentation d'image** ou en encore à la modification d'image pour perturber les prédictions des réseaux de neurones.

4) L'utilisation de filtres 1x1 dans SqueezeNet fournit des cartes d'activation très pixélisées. Avec un réseau plus classique comme VGG, les cartes obtenues sont plus lisses et correspondent mieux à la carte d'activation pour un oeil humain.



## B - Images adverses

5) Quelle que soit l'image en entrée et quelle que soit la classe choisie, **on arrive à tromper le réseau de neurone sans qu'un utilisateur humain ne voit de réelle différence**. Par exemple on a réussi à construire une image d'oiseau que le réseau de neurone interprète comme un alligator. Les changements effectués à l'image ne sont pas naturels, nous créons une nouvelle dynamique d'image en rajoutant un bruit que le réseau n'est pas habitué à voir.



6) Nous venons de démontrer que l'utilisation de ces réseaux de neurones n'est pas fiable si nous avons accès au modèle (et au dataset). On peut par exemple penser à des algorithmes de détection automatique de panneaux de signalisation pour voiture autonome. En appliquant une modification difficilement perceptible pour les humains sur le panneau, on pourrait tromper les voitures.

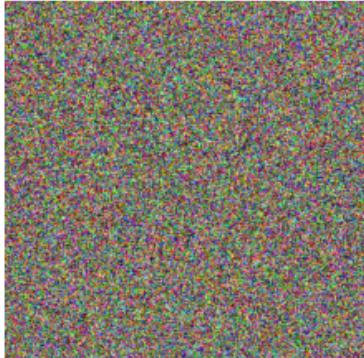
7) En pratique le modèle n'est pas souvent accessible mais ajouter un motif ayant des features très caractéristiques d'une classe permet d'avoir de très bonnes chances de tromper l'algorithme.

## C - Génération d'images

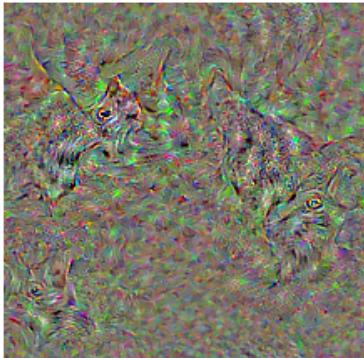
8) De la manière que précédemment, on modifie l'image en suivant le gradient. On ne cherche plus à tromper le réseau mais juste à créer une image appartenant à une classe.

En partant d'un bruit, cette technique fait apparaître des traits caractéristiques de la classe choisie. Dans l'exemple ci-dessous, des pelages de chat et des formes d'oreilles de chat apparaissent.

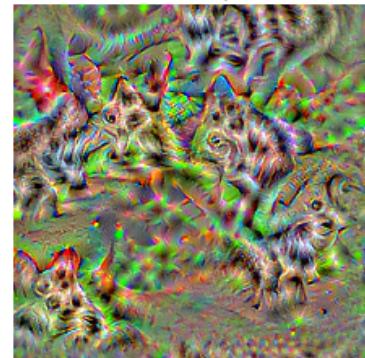
tabby, tabby cat  
Iteration 1 / 1000



tabby, tabby cat  
Iteration 50 / 1000



tabby, tabby cat  
Iteration 500 / 1000



9-10) Afin d'obtenir des images plus réalistes, on remplace le bruit initial par une image représentant un objet de la classe à illustrer. La technique est alors supposée amplifier les traits caractéristiques déjà présents dans l'image.

Tibetan mastiff  
Iteration 1 / 1000

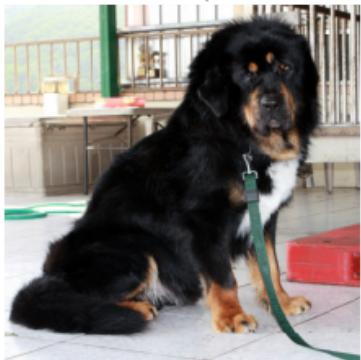


image en entrée

Tibetan mastiff  
Iteration 25 / 1000



après quelques epoch,  
certains traits caractéristiques  
sont accentués, des artefacts apparaissent déjà

Tibetan mastiff  
Iteration 525 / 1000



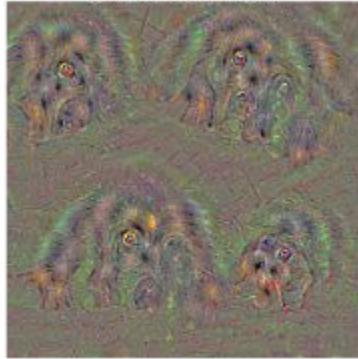
sans régularisation

Tibetan mastiff  
Iteration 200 / 1000



régularisation L2  
de la distance à l'image initiale

Tibetan mastiff  
Iteration 575 / 1000



régularisation L2  
de la norme de la valeur des pixels

Tibetan mastiff  
Iteration 300 / 1000

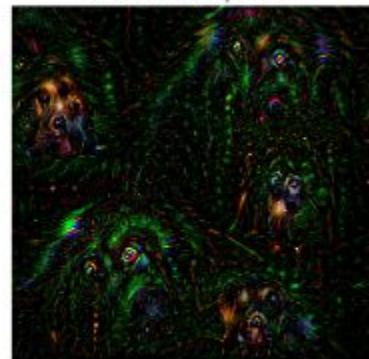


Image obtenue par une  
erreur d'implémentation

Sans régularisation sur la norme de la valeur des pixels, les images deviennent psychédéliques, si la régularisation est trop forte, les images deviennent ternes. Il n'y a pas d'entre deux qui permette de générer des images réalistes.

Augmenter le nombre d'epoch et faire baisser progressivement le learning rate n'a pas d'impact notable, l'image obtenue oscille autour d'optimums locaux.

Utiliser une norme L2 de la distance par rapport à l'image initiale pourrait permettre d'amplifier les traits sans trop s'éloigner de l'image originale. Mais pour un oeil humain, l'image obtenue en pratique **ressemble moins à la classe cible que l'image originale.**

## TP 10/11 - Generative Adversarial Networks

### A - Théorie

On cherche donc à optimiser le problème suivant :

$$\min_G \max_D \mathbb{E}_{\mathbf{x}^* \in \mathcal{D}_{\text{Data}}} [\log D(\mathbf{x}^*)] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))] \quad (5)$$

Concrètement, d'un point de vue pratique, on alterne entre l'apprentissage du générateur et du discriminateur, qui ont chacun une fonction objectif différente dérivée du problème ci-dessus :

$$\max_G \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} [\log D(G(\mathbf{z}))] \quad (6)$$

$$\max_D \mathbb{E}_{\mathbf{x}^* \in \mathcal{D}_{\text{Data}}} [\log D(\mathbf{x}^*)] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))] \quad (7)$$

1) Les équations correspondent aux objectifs des deux réseaux :

- Le réseau G veut maximiser la probabilité que D dise que les images générées soient vraies. La formalisation de son objectif ne considère pas directement les images réelles.
- le réseau D veut d'une part maximiser la probabilité de bien classer les images réelles et d'autre part de repérer les images générées par le réseau G.

**Les deux réseaux doivent progresser parallèlement.** Si nous ne maximisons que le générateur, nous aurions seulement à produire du bruit adverse pouvant certes tromper le discriminateur mais les images générées n'auraient pas de sens. Si nous n'améliorons que le discriminateur, le générateur ne sera pas en mesure de générer des images convaincantes

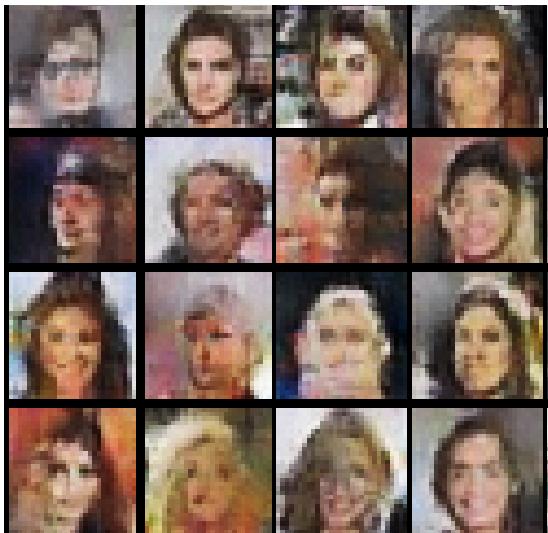
2) Idéalement, le réseau G apprend à **transformer la distribution  $P(z)$  en  $P(\text{Data})$ .** Cependant, la fonction objectif du réseau G contraint juste les données à être **plausibles selon  $P(\text{Data})$  et non de suivre  $P(\text{Data})$ .** Par exemple dans le cas de la génération de chiffres, une situation optimale pour G serait de ne générer que des '0'. On appelle cette situation le '**mode collapse**'.

3) La vraie équation pour G aurait dû être :

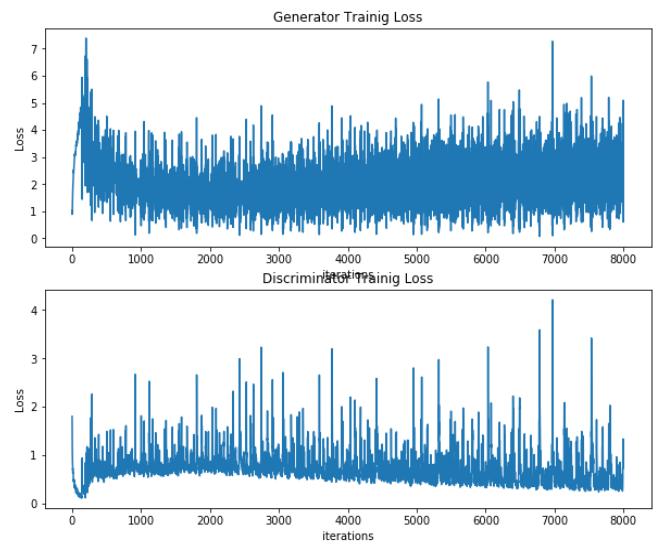
$$\min_G \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))]$$

## B - Expériences

4) Les **loss oscillent** et ne diminuent pas au fil des itérations car les réseaux ont des objectifs opposés, les loss sont **difficilement interprétables**. Les images obtenues **ressemblent bien à des visages** mais elles ne sont pas de bonne qualité, si on les regarde en détail on parvient aisément à distinguer les vraies images des images générées.



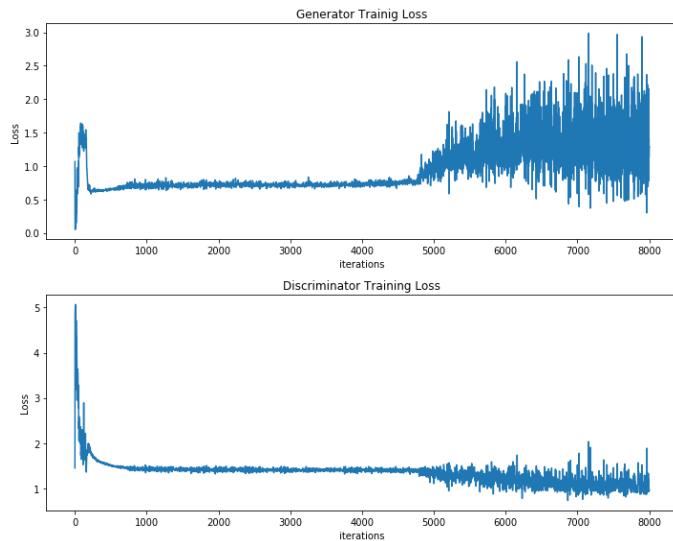
Visages générés



Les loss oscillent sans jamais converger

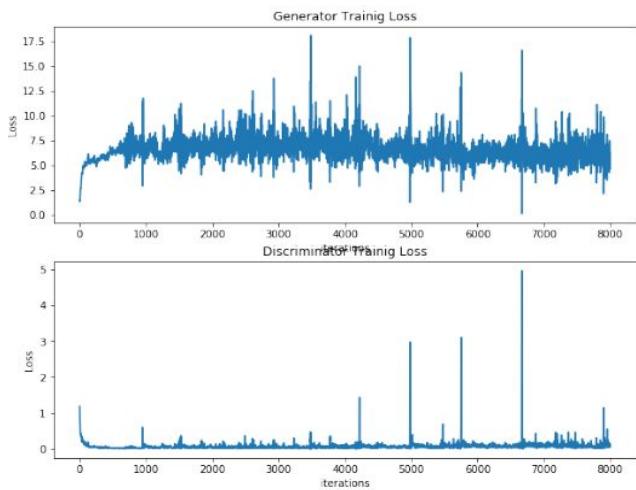
5) Expériences supplémentaires :

- Avec un **learning rate plus grand** pour les deux réseaux, les changements effectués aux réseaux sont plus importants, cela augmente la probabilité que les réseaux se mettent à osciller fortement et causent une chute de performance.



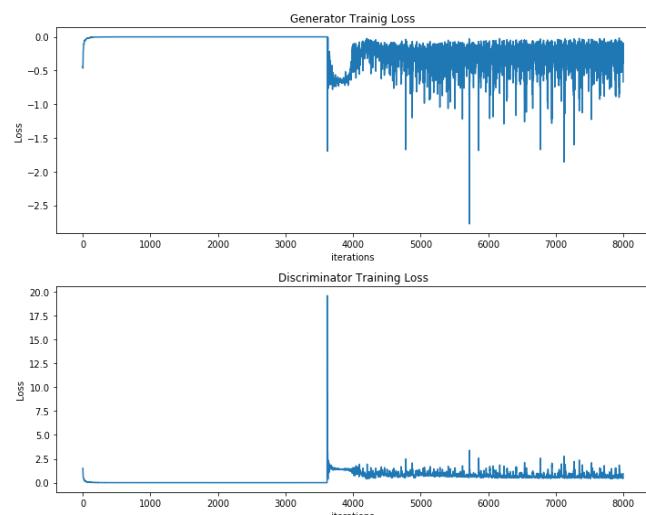
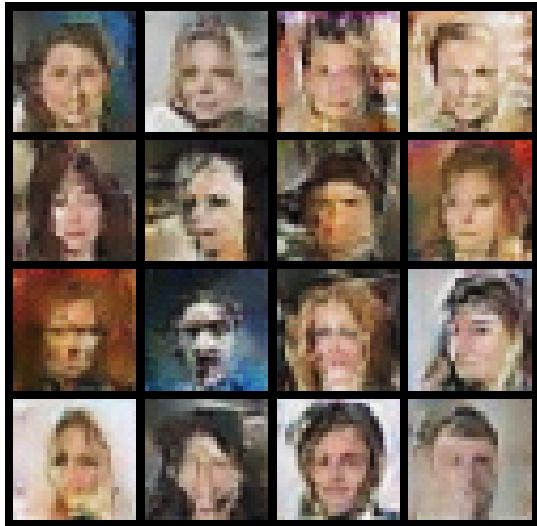
Les performances de notre GAN se sont dégradées à la 5000<sup>ème</sup> itération

- b) En **changeant le momentum** à 0.9, l'entraînement des réseaux est moins bon et les images obtenues sont **plus floues**.



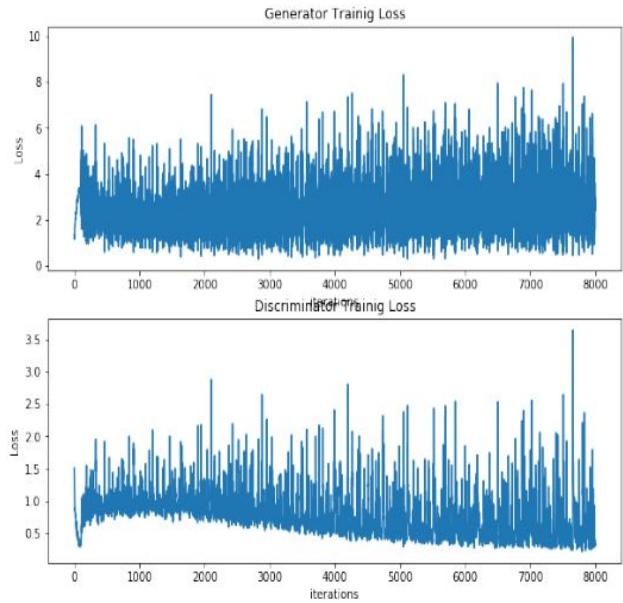
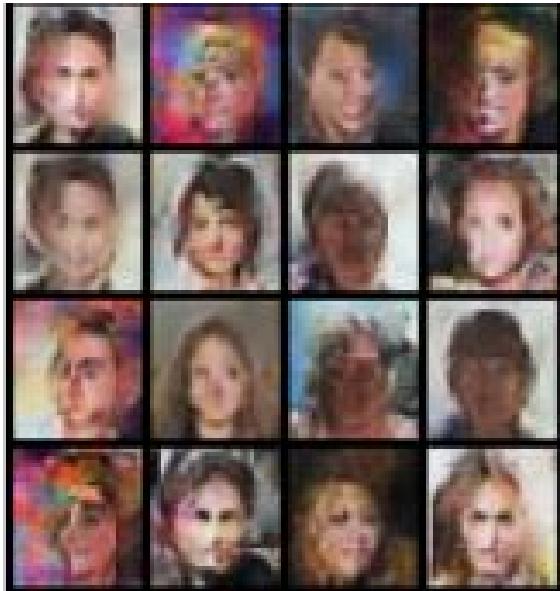
*Le discriminateur arrive à distinguer les images très facilement.*

- c) En **remplaçant la loss** d'apprentissage du générateur par la "vraie" loss dérivée de l'équation d'origine, l'apprentissage du générateur est plus lent, le discriminateur devient alors très fort, ce qui conduit  $D(G(z))$  à tendre vers 0. L'expression a alors un gradient presque nul ce qui fait que le générateur n'apprend plus.

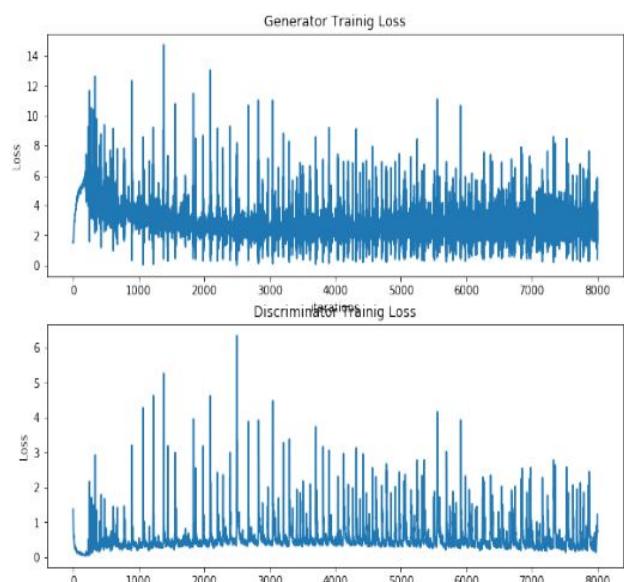
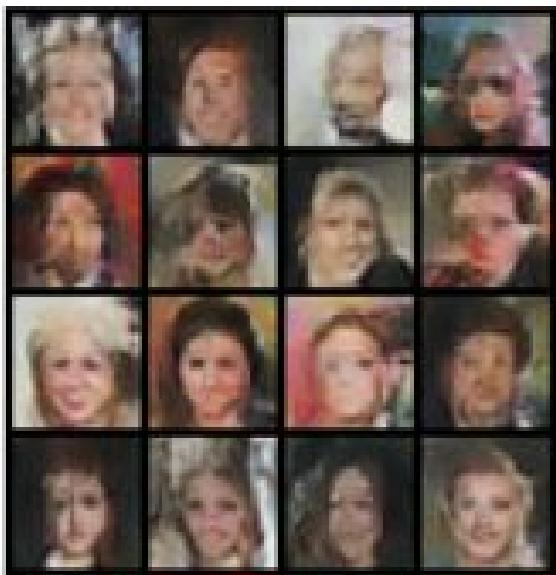


*Si les réseaux arrivent à s'entraîner, les résultats obtenus sont assez similaires, ce qui témoigne l'équivalence des formules*

- d) La taille du vecteur de bruit fourni en entrée n'est pas un hyper-paramètre important, la changer n'a **pas d'effet notable** sur la qualité des images générées. Les images et les loss obtenues se ressemblent beaucoup.

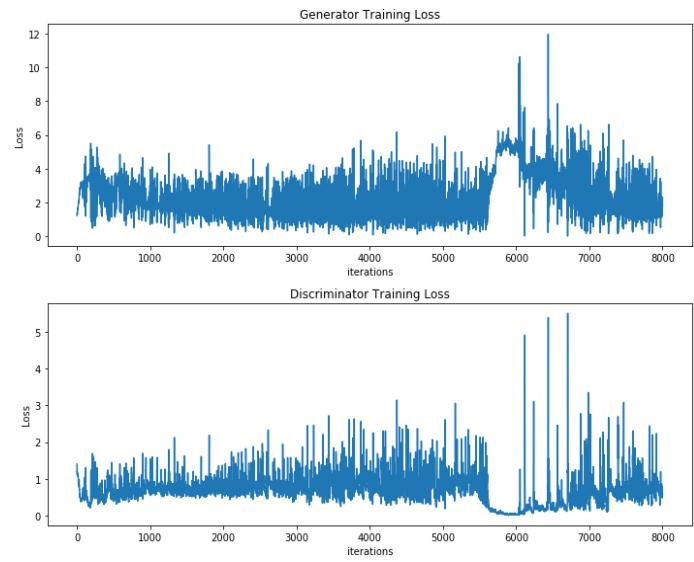


Résultats avec un vecteur de taille 10



Résultats avec un vecteur de taille 1000

- e) On entraîne un GAN sur un autre dataset ne représentant pas des visages : CIFAR-10.



*En 8000 itérations, beaucoup d'images générées n'ont pas de sens*

- f) Etant donné deux images, il est difficile de **passer de façon continue de l'une à l'autre** en générant des images réalistes. Les GAN permettent de passer aisément d'une image à une autre **à condition que ces images soient artificielles** et de connaître les vecteurs latents dont elles sont issues. Les images intermédiaires peuvent alors être obtenues en appliquant le générateur aux **interpolations linéaires entre les deux vecteurs latents**.



*Belle illustration obtenue. source : WGAN-GP*

## C - Conditional GAN

6) Dans le cas des GAN conditionnels, on rajoute une information 'y' en entrée à tous les réseaux, le problème devient :

$$G : \max_G E_{z \sim P(z)} [\log (D(G(z | y)))]$$

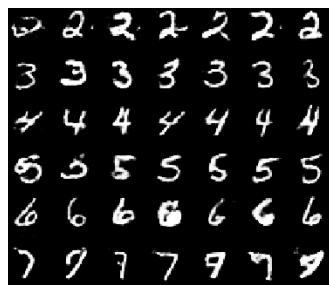
$$D : \max_D E_{x \sim P(\text{Data})} [\log (D(x | y))] + E_{z \sim P(z)} [\log (1 - D(G(z | y) | y))]$$

7) Ce modèle modifie une image en changeant l'âge ou le sexe de la personne. Afin de générer une image proche de l'entrée, le réseau prend cette image en entrée. De plus afin de pouvoir modifier l'âge ou le sexe de la personne, le réseau prend aussi ces informations en entrée.

8) Ce modèle génère une vidéo à partir d'une autre vidéo en changeant la saison. Une hypothèse plausible est que le réseau générateur prend en entrée une image de la vidéo d'entrée et génère une image correspondante. Il est de plus conditionné à la saison.

9) Ce modèle génère un monde urbain autour d'un personnage de jeu vidéo se déplaçant dans la ville. Le modèle ne reçoit pas de vidéo en entrée, afin conserver une cohérence d'une frame générée à la suivante, le modèle doit être conditionné à un petit nombre des dernières images générées. Il est de plus conditionné au style de ville et aux actions de l'utilisateur.

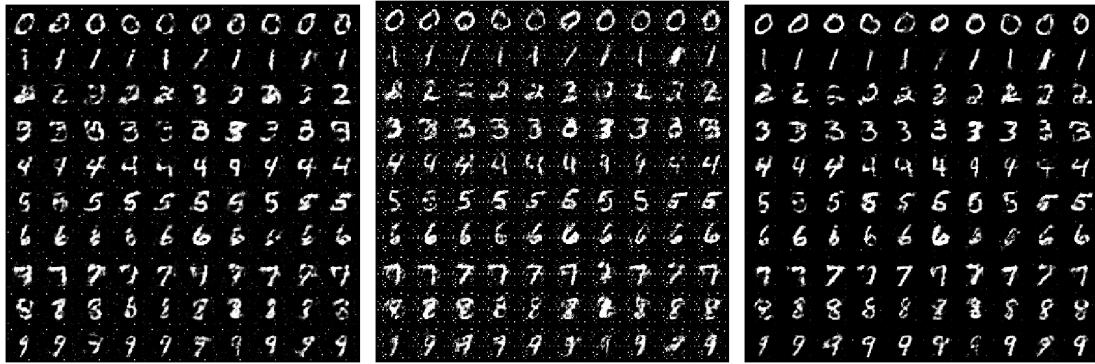
10) On entraîne un cDCGAN sur Mnist en utilisant la classe de l'image comme conditionnement (avec un 'one hot vector'). L'apprentissage d'un GAN conditionné est plus stable que celui d'un GAN classique. Les images générées ne sont pas de parfaits chiffres, cela n'est pas causé par une mauvaise performance des réseaux de neurones mais simplement par le fait que les images du dataset ne sont pas belles non plus.



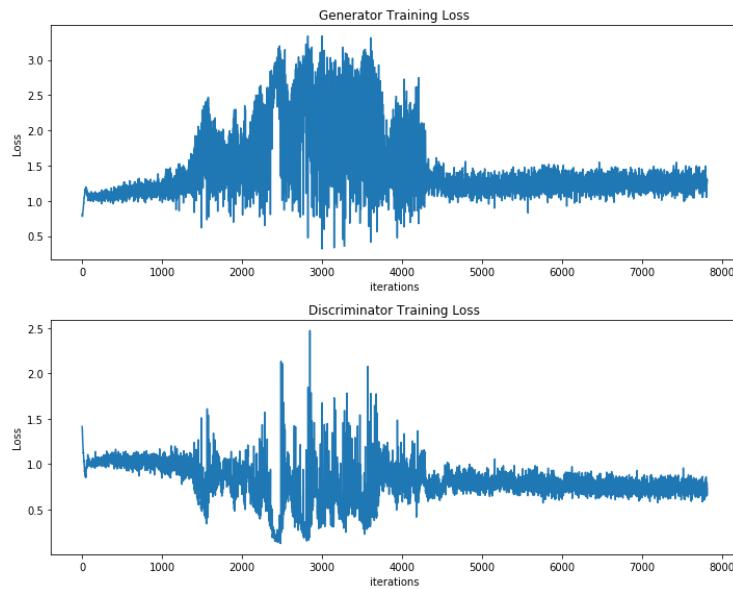
Chiffres générés par notre réseau

11) Donner l'information 'y' au discriminateur lui permet de détecter si le générateur génère un objet ne correspondant pas à ce qui a été demandé. Sans cette information, il y a plus de risque de mode collapse et plus aucune garantie que les images générées soient de la bonne classe.

12)



*images générées au début, milieu et fin de l'entraînement*



*loss obtenues pendant l'entraînement*

L'entraînement des réseaux est plus instable. Durant notre expérience, la qualité des images s'améliore pendant les 2000 premières itérations et se détériore ensuite jusqu'à l'itération 4000 environ. Les images générées au milieu de l'entraînement sont plus bruitées : elles présentent de nombreux points blancs quelle que soit le vecteur en entrée, ce phénomène s'accompagne de loss plus fortes. A la fin de notre expérience, les images générées sont de bonne qualité et les réseaux ont l'air de s'être stabilisés.

13) Le cGAN utilise des couches ‘fully-connected’ qui **ne capturent pas d'informations spatiales**, cela **complexifie la génération d'image globalement réaliste**.