Compte rendu TP 6-7
Réseaux convolutionnels pour l'image

BEROUKHIM Keyvan

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 (k2-1) * s1 + k1.
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**.

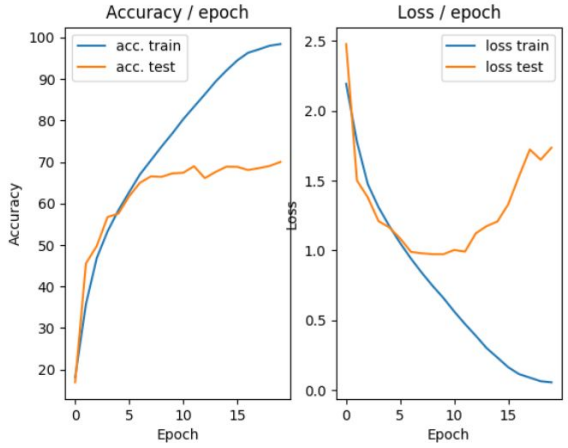
12) Pour l'approche **BoW+SVM** utilisée précédemment avec un dictionnaire de 1000 SIFT de taille 128, on apprend de l'ordre de 128 000 paramètres. Il y a plusieurs hyperparamètres à fixer comme la taille du dictionnaire appris, la constante 'C' du SVM et le type de SIFT utilisé. Avec le **réseau convolutionnel** 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).

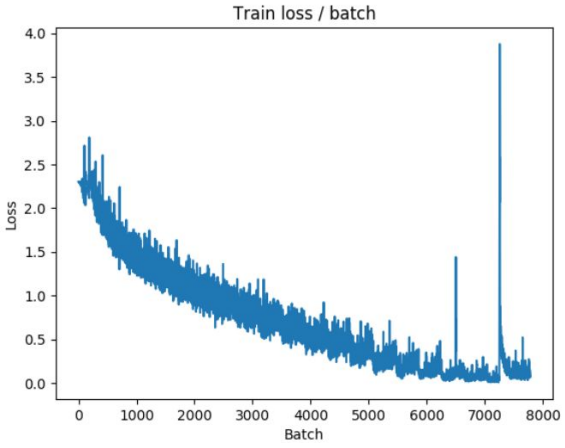
16) La convergence dépend du choix de η : si il est trop petit, le modèle va être **long à entraîner**, si il est trop grand, le modèle **peut ne pas arriver à converger**. En pratique, on fait souvent **décroitre η au fil des itérations**.
Selon la quantité des données utilisées pour la calculer le 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)



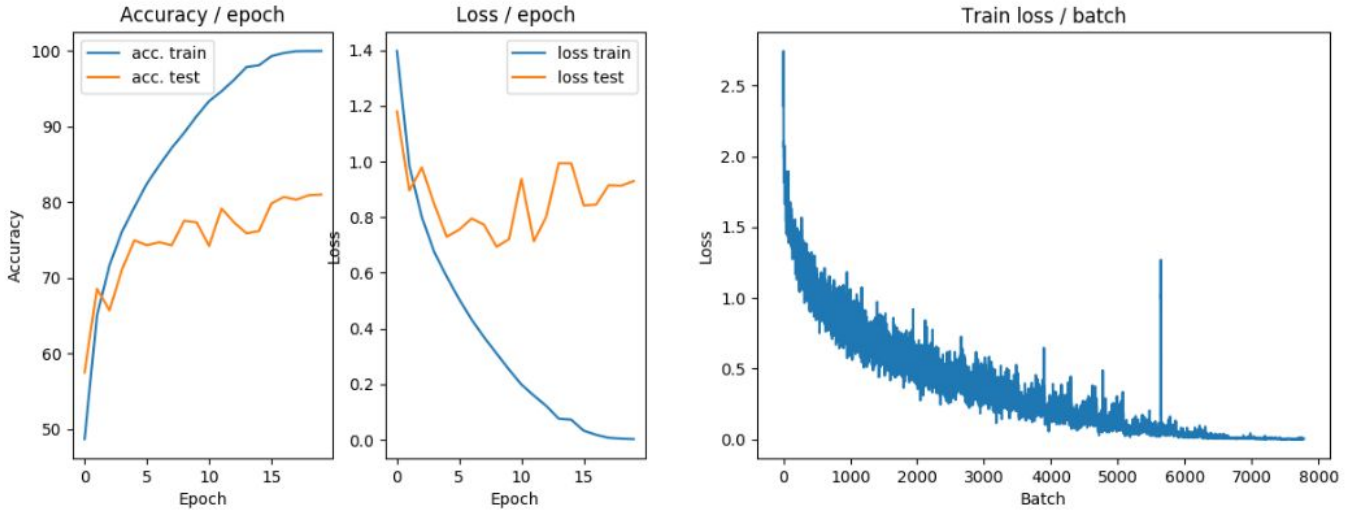
a) accuracy et loss en train et en test pour chaque epoch



b) loss obtenue pour chaque itération en train

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)



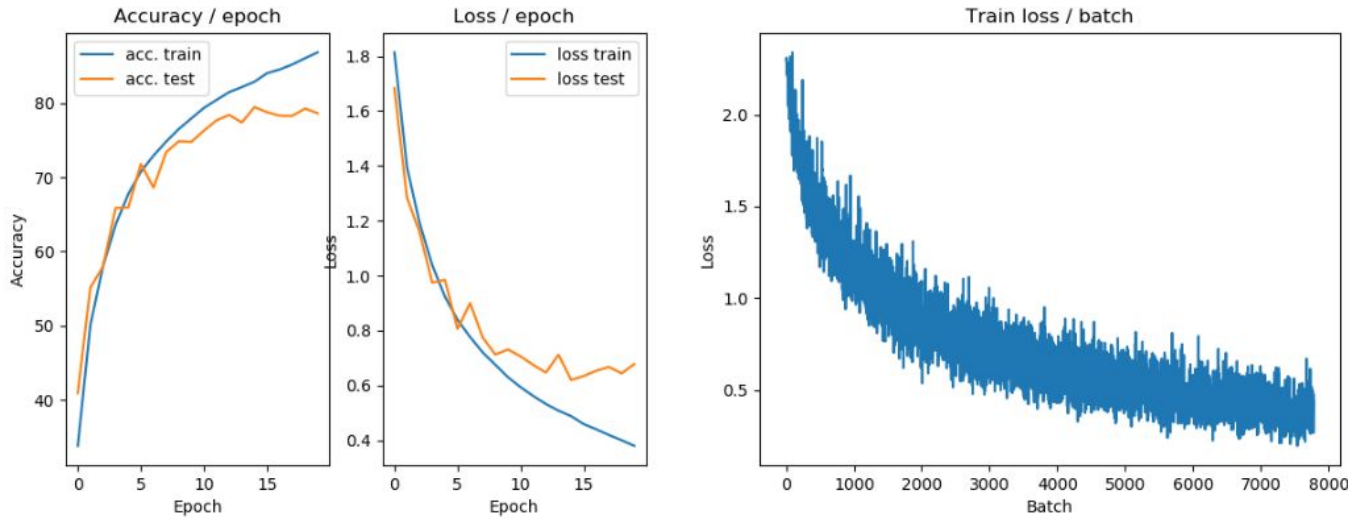
rajout d'une normalisation

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)



rajout de data augmentation

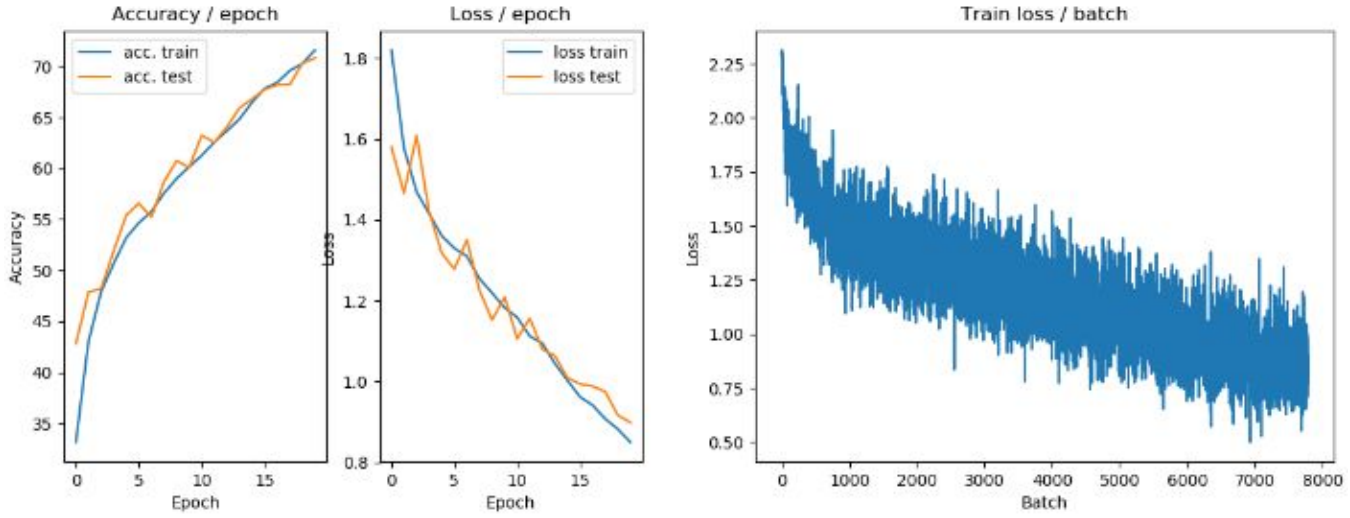
En rajoutant de la data augmentation, le sur-apprentissage arrive plus tardivement et est moins prononcé: les performances diffèrent moins entre le jeu de train et le jeu de test.

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 on rajoutant un **momentum** de 0.9 et on 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 surement 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 η par la variance et à la place de multiplier par le gradient on multiplie par la moyenne des gradients.

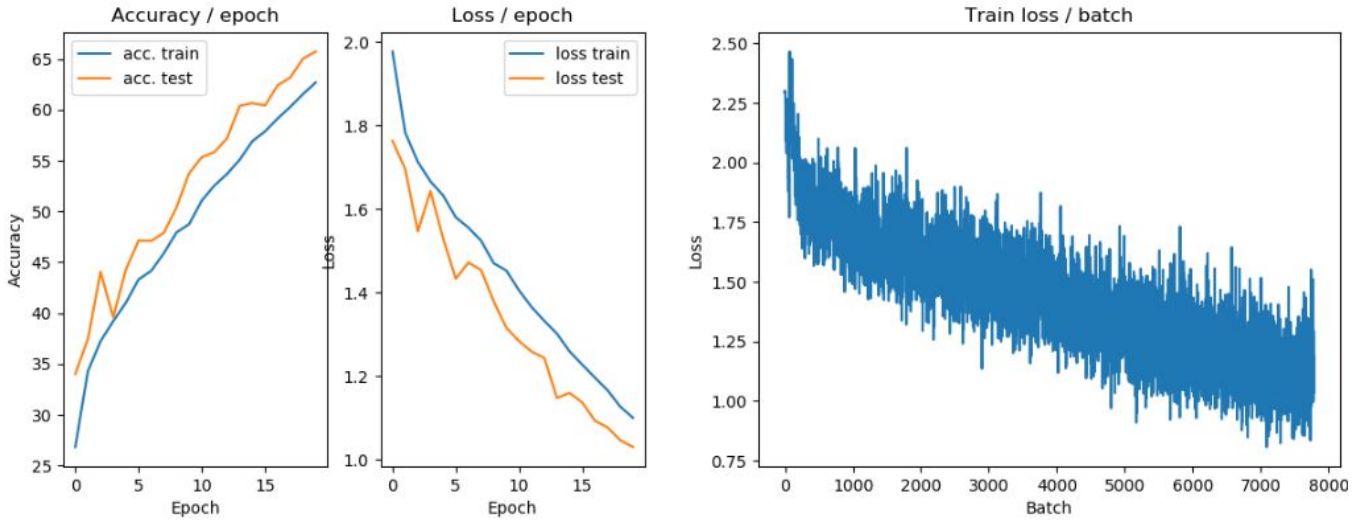
->AdaMax : Une variante d'Adam basée sur la norme de l'infini; à la place d'utiliser la norme l2 de la variance on calcule le max entre la variance estimée et la norme des gradients ce qui donne l'avantage d'être moins sensible au choix des hyper-paramètres .

->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)



rajout de dropout entre les deux couches linéaires

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.

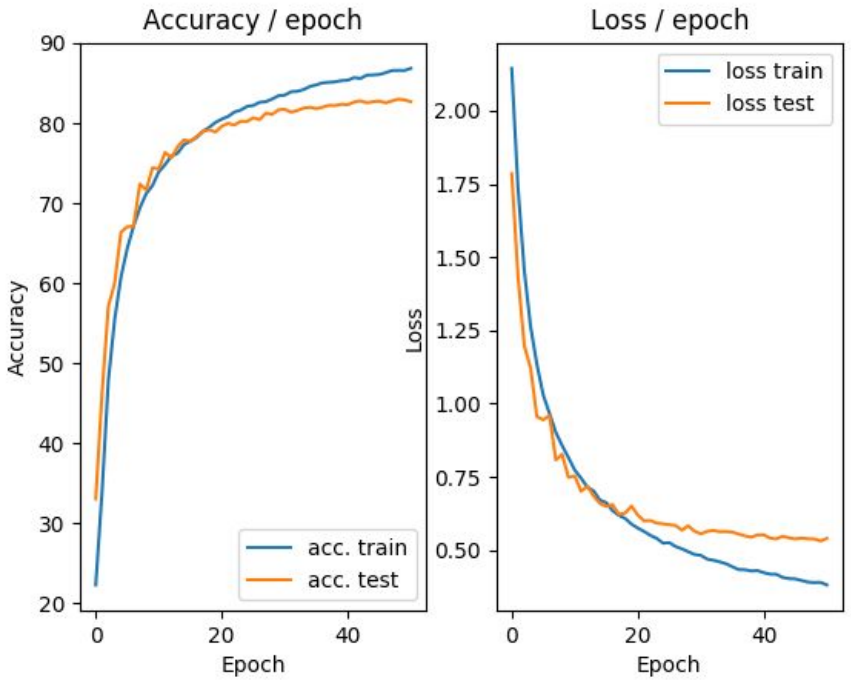
30) La régularisation est un **processus de pénalisation envers la complexité du modèle**. Cela permet de réduire le risque de sur-apprentissage.

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.