Sorbonne Université Sciences

Rapport RDFIA des TMEs 4, 5, 6 et 7

Introduction aux Réseaux de Neurones

Master 2 Données, Apprentissage et Connaissances

OUARDA FENEK

2019 - 2020

1 Réponses aux questions du TME 4_5

1. L'ensemble d'apprentissage sert à apprendre les paramètres de notre modèle. L'ensemble de validation sert à fixer les hyper paramètres. Et enfin, l'ensemble de test sert à évaluer les performances du modèle.

2. L'effet du nombre d'exemples :
   Trop peu de données en apprentissage aboutisseront à une mauvaise approximation. Un modèle surchargé sous exploitera le petit jeu de données, tandis qu'un modèle sous contraint (trop simple) entrainera un sur-apprentissage du modèles qui mènera vers des scores médiocres. Quant au test, trop peu de données entrainera une estimation optimiste et à forte variance de la performance du modèle.
   Par contre un très grand nombre d'exemples sera très long à entrainer. Il faut donc s'arranger à faire l'équilibre entre. les deux cas.

3. Rôles des fonctions d'activation :
   Tout d'abord, elle permettent de rendre la combinaison des transformantions (linéaire + activiation) non-linéaire, permettant ainsi d'appliquer plusieurs transformations de suite. Ensuite, elle permettent de choisir un intervalle d'arrivée différent de R^{n_y}.

4. Sur la figure : $n_x = 2,\ n_h = 4,\ n_y = 2$
   n_x est choisi par rapport au nombre de features de x (la dimension de x)
   n_h est le nombre de neuronnes dans la couche cachée, il existe certaines règles empiriques pour le poser. Celle qui est le plus souvent invoquée est «la taille optimale de la couche masquée est généralement comprise entre la taille de l'entrée et la taille des couches de sortie». Ou bien la moyenne du nombre de neuronnes d'entrée et de sortie. Mais celà reste dans les cas généraux. Selon le problème, on pourrait être amenés à changer cet hyper paramètres (en validation) jusqu'à en trouver le meilleur.
   n_y représente le nombre de clusters distincts qu'on a dans notre dataset.

5. y^{hat} représente la sortie prédite pour l'entrée du réseau de neurones.
   y représente la sortie attendue pour l'entrée en question.
   La différence entre ces deux quantités est l'erreur qu'on cherchera à minimiser par la suite.

6. SoftMax permet de se ramener dans l'intervalle [0, 1] et tel que $\sum_i SoftMax(x)_i = 1$ permettant ainsi d'avoir un vecteur assimilable à une distribution de probabilités $SoftMax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j}$.

7. Les équations mathématiques permettant de faire le $forward$:
   $h^{tilde} = W_h x + b_h$
   $h = Tanh(h^{tilde})$
   $y^{tilde} = W_y h + b_y$
   $y^{hat} = SoftMax(y^{tilde})$

8. Dans l'erreur des moindres carrés on veut que le y^{hat} s'approche du y. Dans la cross-entropie, on veut que le y^{hat}_i s'approche de 1 et tout les autres de 0.

9. L'erreur des moidres carrés est adaptée aux problèmes de regression, vu qu'on charche à approximer la valeur réelle avec la prédiction. Quant à la cross-entropie, elle est adaptée aux

problèmes de classification, car dans sa formule, il n'y a que le terme qui est associé à la bonne classe qui n'est pas nul.

10. Dans la descente de gradient stochastique, comme on utilise qu'un exemple à la fois, le chemin vers les minima est plus bruyant (plus aléatoire) que quand on utilise un mini batch. Mais c'est correct car nous sommes indifférents au chemin, tant que cela nous donne le minimum et un temps d'entraînement plus court.
    Et donc donc pour le mini batch, le chemin vers le minimum est plus stable, par contre l'entrainement est plus long.
    De celà découle que la version classique est encore plus lente, mais elle mène de façon stable vers le minimum.

11. Le pas d'apprentissage joue sur la vitesse de convergence, si il est trop petit, la convergence est très lente et on risque de ne pas y arriver au bout d'un très grand nombre d'itéartions. Par contre si il est très grand, on risque de diverger carrément. Donc il faut faire attention à choisir de façon arbitraire sa valeur.

12. Avec l'algorithme de back-propagation, la complexité du forward et de la back-propagation est de $o(dim \text{ x } nbNeurons)$ par couche, ça fait donc en total $o(nbLayers \text{ x } dim \text{ x } nbNeurons)$ et c'est beaucoup plus couteux en termes de calculs que l'approche naïve.

13. La back-propagation consiste à propager la dérivée de l'erreur dans les couches successives de neurones, donc pour la permettre la fonction de coût doit être dérivable, et il faut faire attention à sauvegarder les gradients calculés à chaque étape si l'implémentation est manuelle, sinon, il faut activer le paramètre $requires_grad$ des tenseurs.

14. Simplification de la fonction de coût :
    On a : $L(y, \hat{y}) = -\sum_i y_i log\ \hat{y}_i$
    et $\hat{y}_i = Softmax(\tilde{y}) = \frac{e^{\tilde{y}_i}}{\sum_i e^{\tilde{y}_j}}$
    En faisant la distribution :

$$\begin{aligned} L(y, \hat{y}) &= -\sum_i y_i log\ e^{\tilde{y}_i} + \sum_i y_i log \sum_j e^{\hat{y}_j} \\ &= -\sum_i y_i \tilde{y}_i + \sum_i y_i log \sum_j e^{\hat{y}_j} \end{aligned}$$

    Sur tout les i on a un seul qui vérifie $y_i = 1$, les autres sont tous à 0 donc on pourrait remplacer la somme sur les y_i par 1 directement, et dans ce cas on aurra :

$$L(y, \hat{y}) = -\sum_i y_i \tilde{y}_i + log \sum_i e^{\hat{y}_i}$$

15. Le gradient de la Loss par rapport à la sortie $\tilde{y}_i$:
    Formule terme à terme : $\delta_{y,i} \frac{\partial L}{\partial \tilde{y}_i} = \hat{y}_i - y_i$
    Formule vectorielle : $\nabla \tilde{y} = \hat{y} - y$

16. Le gradient de la Loss par rapport aux poids de la couche de sortie :
    Formules terme à terme :
    $\frac{\partial L}{\partial W_{y,ij}} = \delta_{y,i} h_j$
    $\frac{\partial L}{\partial b_{y,i}} = \delta_{y,i}$

Formules vectorielles :
$\nabla W_y = \nabla \tilde{y} h^T$
$\nabla b_y = \nabla \tilde{y}$

17. Le gradient de la Loss par rapport aux poinds de la couche cachée :
Formules terme à terme :
$\delta_{h,i} = \frac{\partial L}{\partial \tilde{h}_i} = (1 - h_i^2) \sum_j \delta_{y,j} W_{y,ij}$
$\frac{\partial L}{\partial W_{h,ij}} = \delta_{h,i} x_j$
$\frac{\partial L}{\partial h_{y,i}} = \delta_{h,i}$

Formules vectorielles :
$\nabla \tilde{h} = W_y^T \nabla \tilde{y} (1 - h^2)$
$\nabla W_h = \nabla \tilde{h} x^T$
$\nabla b_h = \nabla \tilde{h}$

2 Réponses aux questions du TME 6_7

1. Ayant une entrée $x * y * z$, si on lui applique un filtre de convolution de padding p, de stride s et de kernel k, on aura une sortie dont les dimensions sont comme suit :
$x^{'} = \frac{x-k+2p}{s} + 1$
$y^{'} = \frac{y-k+2p}{s} + 1$
et $z^{'} = 1$ car on a utlisé un seul filtre (ref : C4W1L06 Convolutions Over Volumes), si on avait n, on aurait que $z^{'} = n$.

   Le nombre de paramètres à apprendre serait correspondant à la taille du filtre + le biais, ça serait donc $k * k * z + 1$ (z correspond à $in\_channels$. Si on avait appliqué n filtres, le nombre de paramètres serait $k * k * z * n + n$ (un biais pour chaque filtre).

   Si on devait avoir une sortie de la même taille avec une couche Fully-Connected, le nombre de paramètres à apprendre serait : taille de l'entrée * taille de la sortie + taille de la sortie ($W + b$), i.e : $x * y * z * x^{'} * y^{'} * z^{'} + x^{'} * y^{'} * z^{'}$.

2. L'avantage qu'apporte les couches de convolution est qu'elles réduisent beaucoup le nombre de paramètres à apprendre par rapport à des couches FC.
La limite principale du produit de convolution c'est qu'il prend du temps à appliquer, l'apprentissage du réseau serait long. De plus, contrairement à l'algorithme SIFT par exemple, on ne garantit pas l'invariance à la translation et l'orientation en utilisat des CNN sur des images.

3. Le pooling permet de réduire les dimensions des couches, celà donc réduit le nombre de paramètres à apprendre à la couche suivante sans une grande perte d'informations, vu que toutes les informations du voisinage seront regroupées dans le cas d'Average Pooling, où l'information la plus importante est gardée dans le cas du Max Pooling. Cette opération rend l'apprentissage du réseau faisable et plus rapide en termes de gain d'espace mémoire et de temps de calcul.

4. Si la taille de l'image en entrée est supérieure à la taille d'entrainement du CNN, on échantillonne l'image en sous fenêtres et on les passe au CNN, c'est comme si le filtre parcourait

l'image. Mais en arrivant aux couches fully connected, il y aura un problème de dimensions, car le fully connected est sensible à la taille de l'entrée.
En gros on ne pourra pas utiliser un tel réseau pour apprendre de plus grandes images.

5. Il est possible de convertir une couche FC en une couche de convolution si nous définissons la taille du noyau (kernel) pour correspondre à la taille de l'entrée. Définir le nombre de filtres revient alors à définir le nombre de neurones de sortie dans une couche FC.

6. Maintenant que les couches FC sont remplacées, on aura pas de problèmes de dimensions, vu qu'avec couche CNN on peut avoir une entrée supérieure à la dimension d'entrainement, il suffit juste de la sous échantilloner en fenêtres de même taille qu'on alimentera aux couches CNN une par une.
Au final, la taille de la sortie sera variable en fonction de la taille de l'entrée.

7. Dans un réseau CNN, le champ de réception peut être augmenté en utilisant différentes méthodes telles que : empiler plus de couches (profondeur), sous-échantillonnage (pooling, foulée), dilatation du filtre (convolutions dilatées), etc.
Si on empile plus de couches, on pourra augmenter le champ de réception de manière linéaire. Cependant, dans la pratique, les choses ne sont pas simples, comme nous le pensions, le concept de «champ récepteur efficace» s'introduit, L'intuition qui sous-tend ce concept est que tous les pixels du champ de réception ne contribuent pas de manière égale à la réponse de l'unité de sortie. Lors du passage en avant, nous pouvons voir que les pixels du champ de réception central peuvent propager leurs informations vers la sortie en utilisant de nombreux chemins différents, car ils font partie des calculs de plusieurs unités de sortie. Tandis que les pixels sur les bordures se propagent le long d'un seul chemin.
Pour calculer la taille du champs récepteur on applique la formule $kernel * profondeur + bias$.

8. Si on veut garder les mêmes dimensions de l'entrée, on prend $s = 1$ et $p = \frac{k+1}{2}$.

9. Pour réduire les dimensions spatiales d'un facteur de 2 avec le MaxPooling, on choisit un $p = 2$ et un $s = 2$.

10. La taille de la sortie et le nombre de poids à apprendre pour chaque couche :
On a l'entrée qui est de dimensions : 32x32x3
$conv1$: 32 convolutions 5×5, stride=1, padding=2 : La sortie sera de dimensions : 32x32x32, le nombre de paramètres à apprendre : 32x(3x5x5+1)=2432
$pool1$: stride=2, padding=0 : La sortie sera de dimensions : 16x16x32, aucun paramètre à apprendre.
$conv2$: 64 convolutions 5×5, stride=1, padding=0 : La sortie sera de dimensions : 12x12x64, le nombre de paramètres à apprendre : 64x(32x5x5+1)=56064
$pool2$: stride=2, padding=0 : La sortie sera de dimensions : 6x6x64, aucun paramètre à apprendre.
$conv2$: 64 convolutions 5×5, stride=1, padding=0 : La sortie sera de dimensions : 2x2x64, le nombre de paramètres à apprendre : 64x(64x5x5+1)=102464
$pool3$: stride=2, padding=0 : La sortie sera de dimensions : 1x1x64, aucun paramètre à apprendre.
$fc4$: Linear(64, 1000) : La sortie sera de dimension 1x1000, le nombre de paramètres à apprendre 64x1000+1000=65000.
$fc5$: Linear(1000,10) : La sortie sera de dimensions 1x10, le nombre de paramètres à apprendre

1000x10+10=10010.
On voit que les couches de convolutions ont beaucoup moins de paramètres à apprendre par rapport aux couches fully-connected, et l'application du max-pooling réduit encore les dimensions des sorties, ce qui réduit le nombre de paramètres à apprendre à la couche suivante.
On applique quand même des fc pour adapter la sortie au problème de classification exprimé.

11. Celà fait en total 235970
    Le nombre d'exemples de la base de données CIFAR-10 est relativement petit par rapport à ce nombre de paramètres.

12. Le nombre de paramètres dans un CNN est de l'ordre de milliers, il augmente en ajoutant des couches, concernant l'approche BoW suivi d'un SVM le nombre de paramètres pour la construction du BoW il n'y a pas de paramètres à apprendre, et pour le SVM, le nombre de paramètres à apprendre correspond au vecteur de poids qui est la taille de l'entrée $X(N, d)$ x $W(d, 1) + bias = Y(N, 1)$.

13. Exécution

14. La différence qu'il y a entre la façon de calculer la loss en train et en test, c'est qu'en test on s'assure d'avoir désactivé la sauvegarde des gradients, on précise qu'on est en mode d'évaluation pour que le modèle ne se sert pas des gradients calculés à cette étape lors des back-propagations à venir.
    $model.eval()\ if\ optimizer\ is\ None\ else\ model.train()$
    Mais la formule de calcul de l'accuracy reste la même.

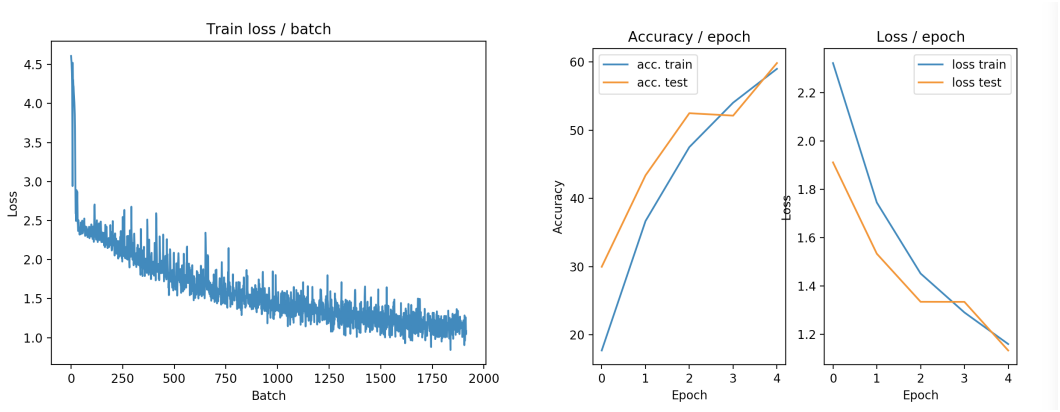
15. Modification du réseau :

```python
class ConvNet(nn.Module):
    """
    Cette classe contient la structure du réseau de neurones
    """

    def __init__(self):
        super(ConvNet, self).__init__()
        # On défini d'abord les couches de convolution et de pooling comme un
        # groupe de couches `self.features`
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, (5, 5), stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d((2, 2), stride=2, padding=0),
            nn.Conv2d(32, 64, (5, 5), stride=1, padding=0),
            nn.ReLU(),
            nn.MaxPool2d((2, 2), stride=2, padding=0),
            nn.Conv2d(64, 64, (5, 5), stride=1, padding=0),
            nn.ReLU(),
            nn.MaxPool2d((2, 2), stride=2, padding=0),
        )
        # On défini les couches fully connected comme un groupe de couches
        # `self.classifier`
        self.classifier = nn.Sequential(
            nn.Linear(64, 1000),
            nn.ReLU(),
            nn.Linear(1000, 100),

            # Rappel : Le softmax est inclus dans la loss, ne pas le mettre ici
        )

    # méthode appelée quand on applique le réseau à un batch d'input
    def forward(self, input):
        bsize = input.size(0) # taille du batch
        output = self.features(input) # on calcule la sortie des conv
        output = output.view(bsize, -1) # on applati les feature map 2D en un
                                        # vecteur 1D pour chaque input

        output = self.classifier(output) # on calcule la sortie des fc
        return output
```

16. Effet du $learning\_rate$ et du $batch\_size$:
    Le fait d'avoir un learning rate très petit ralentit la vitesse de convergence, mais le fait de les avoir très grand aussi peut faire diverger le modèle.
    Concernant la taille du batch, en la prenant petite, le modèle devient bruyant. Mais si elle est très grande, l'apprentissage devient compliqué en terme d'emplacement mémoire.
    Il est donc important de bien fixer ces hyper-paramètres.

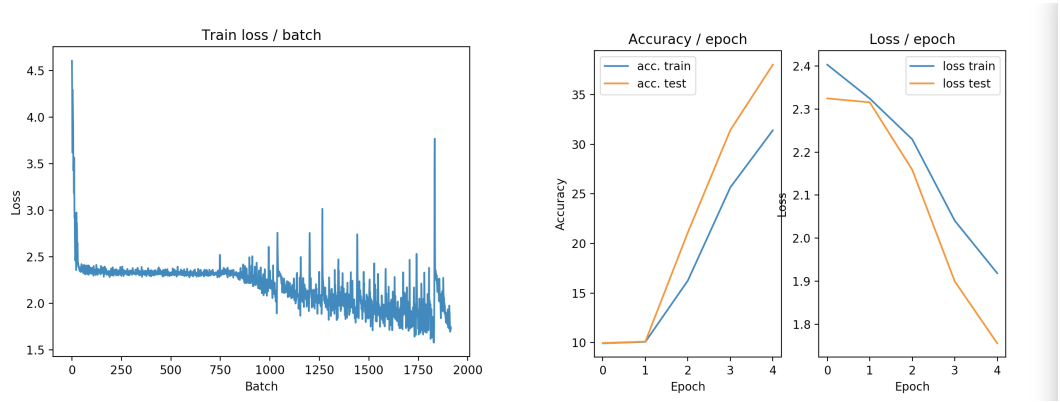
17. L'erreur à la première époque correspond à l'erreur causée par les paramètres initialisé aléatoirement, il n'y a eu aucun apprentissage fait, aucune back-popagation ou mise à jour. C'est l'erreur initiale avant le début de l'appretissage du modèle. Et en pricipe elle est toujours assez élevée vu qu'on a encore rien appris sur les données. En construisant notre modèle, on devrait toujours être capable de faire mieux que cette erreur initiale (de faire mieux que l'aléatoire) sinon notre modèle est juste médiocre.

18. Exploration et interprétation des résultats :



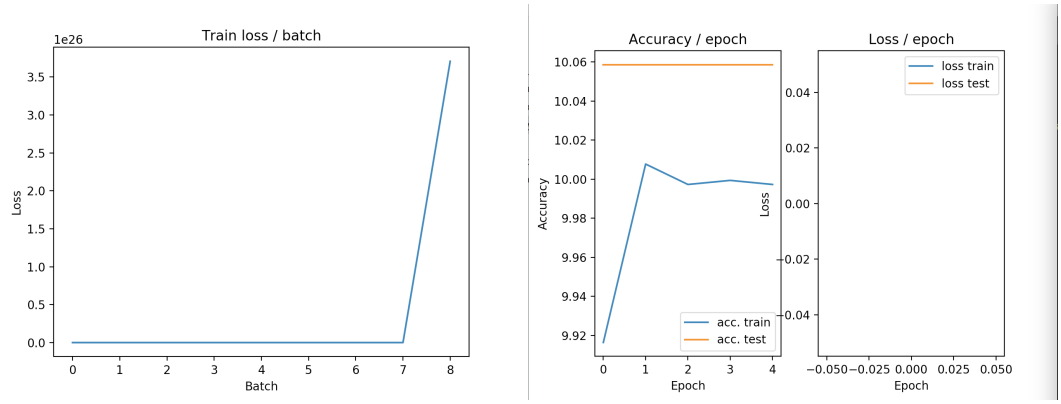
$learning\_rate = 0.1$ *and* $batch\_size = 128$

Le modèle converge à atteindre 60% d'accuracy, l'erreur diminue de façon assez lente (pas d'apprentissage petit) et de façon bruitée (batch petit aussi).



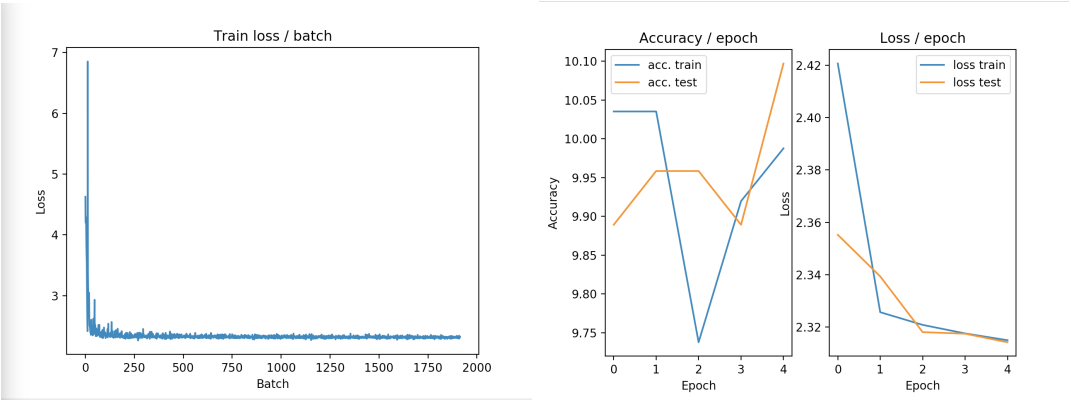
$learning\_rate = 0.5$ *and* $batch\_size = 128$

Dans ce cas, on voit que le modèle converge lentement de façon assez bruitée vu que la taille du batch et le pas d'apprentissage sont petit. Mais il se comporte bien, on a pas de sous ou sur apprentissage, le modèle se comprte aussi bien en train qu'en test. ET on atteind les 40% d'accuracy (on voit qu'on a perdu 20% d'accuracy par rapport au cas précédent).



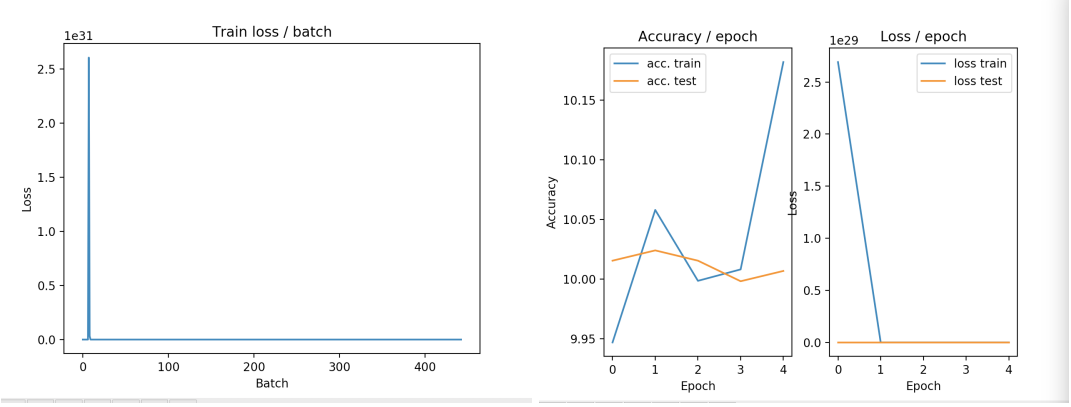
$learning\_rate = 0.5$ *and* $batch\_size = 512$

Dans ce cas, le modèle est mal entrainé, on se retrouve avec un problème de divergence ou la loss explose après un certain nombre d'époche.



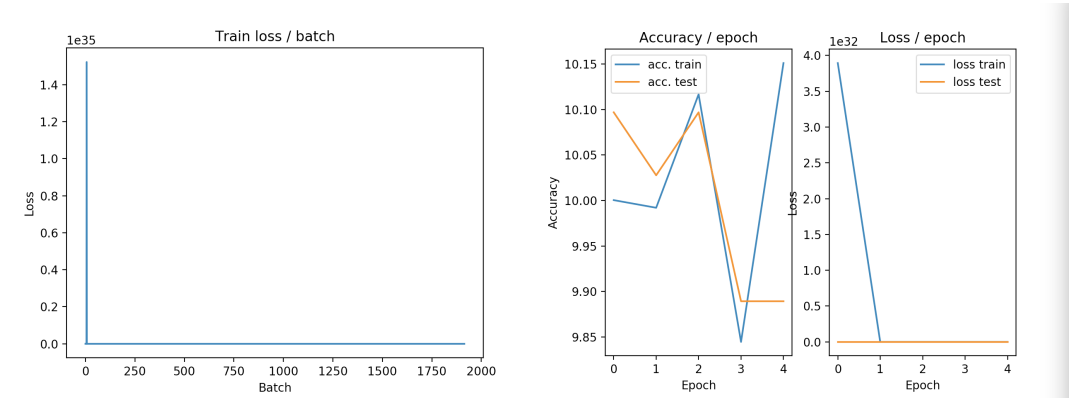
$$learning\_rate = 1 \; and \; batch\_size = 128$$

Le pas d'apprentissage est adéquat dans ce cas, le modèle converge assez vite mais il est un bruyant vu que la taille du batch est petite. Par contre on a perdu en terme d'accuracy aussi (50% de perte par rapport au premier cas de figure).



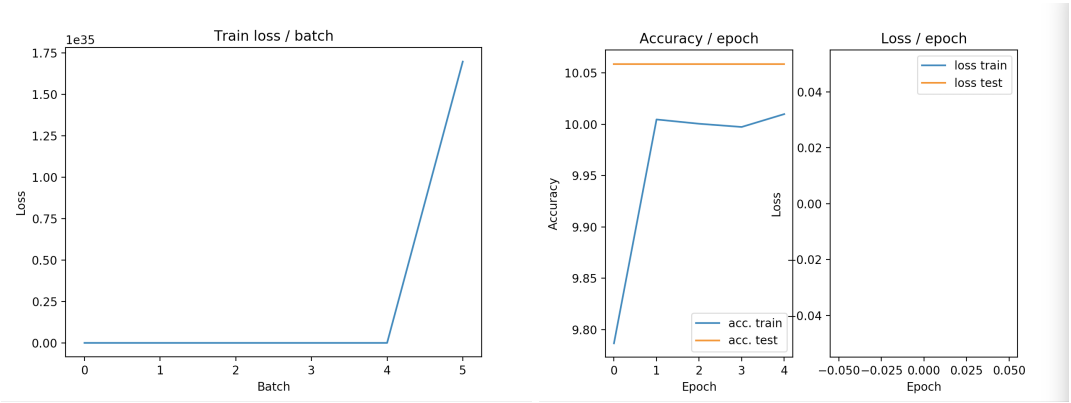
$$learning\_rate = 1 \; and \; batch\_size = 512$$

On a un pas d'apprentissage qui permet au modèle de converger très vite et de façon stable vu que le batch est grand, mais on a pas une bonne accuracy.



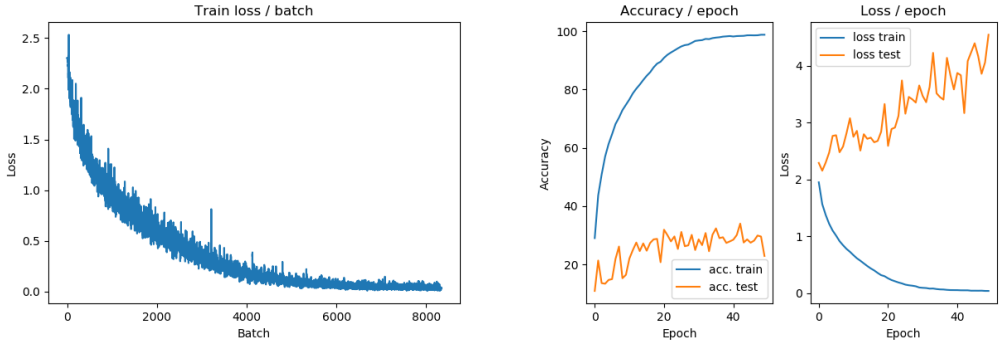
$$learning\_rate = 2 \; and \; batch\_size = 128$$

Le modèle converge mais on a une perte en accuracy par rapport aux cas de convergence précédents.



$$learning\_rate = 2 \; and \; batch\_size = 512$$

Encore une fois, on est sur un cas ou le modèle diverge ou bout de la 4 ième époque.

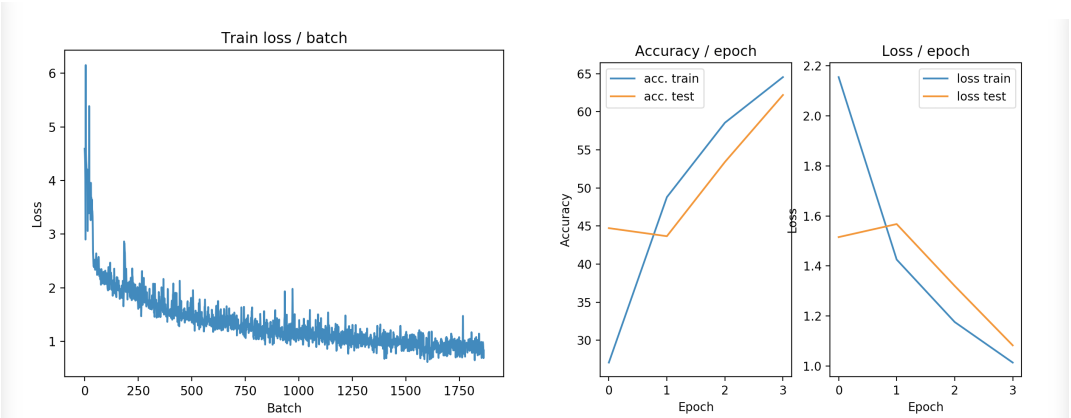


$$learning\_rate = 0.1 \; and \; batch\_size = 300 \; and epochs = 150$$

Pour ces hyper-paramètres on voit clairement le problème de sur-apprentissage qui apparait.

En fixant ainsi les hyper-paramètres, on pourrait donc avoir plusieurs problèmes, dont la divergence du modèle, la lenteur de convergence, le sur apprentissage ou même le sous apprentissage.

19. En normalisant nos données d'apprentissage, on a des meilleurs résultats, une convergence plus rapide et plus stable car on a fait en sorte que nos données soient de la même grandeur. Le fait d'avoir des données qui se ressemblent améliore la convergence des algorithmes et la précision des calculs.
En faisant cette manipulation, on a gagné en accuracy, tout à l'heure on était à 60% pour le cas de figure ou $learning\_rate = 0.1 \; and \; batch\_size = 128$ qui était le meilleur, là on atteind les 65% et on diminue la loss jusqu'à arriver en dessous de 1.



$$learning\_rate = 0.1 \; and \; batch\_size = 128$$

20. Généralement on a pas accès au données de test, donc on a interet à utiliser les mêmes paramètres pour tous. De plus l'ensemble d'apprentissage est assez grand, la moyenne calaculée sur ces données serait proche de la moyenne des données de test. Un autre interêt serait de ne pas perdre de généralité, ne pas recalculer la moyenne à chaque fois que l'ensemble de test est changé.

21. La ZCA est une méthode de normalisation qui pondère les pixels, par exemple, dans une image, les pixels des alentours ne sont pas en général ce qui contiennent l'information pertinente, ils auront donc des poids faibles, contrairement à ceux du milieu.
    Cependant son utilisation n'est pas très pratique, vu qu'elle est trop lente.

22. En appliquant la data augmentation, on a augmenté le nombre d'exemples d'apprentissage avec modification de certains facteurs tels que l'orientation celà améliorera la robustesse du modèle et diminuera sa sensibilité à l'orientation par exemple. Dans ce cas, ça été en notre faveur, on a eu une amélioration des résultats, ou on a atteind 70% d'accuracy.

23. Cette approche peut conduire à une confusion pour certaines images telles que les images de chiffres, si on applique une symétrie horizontale pour un "6" par exemple, on créera une confusion avec "9". C'est la même confusion qui peut être créée en appliquant une symétrie verticale sur le chiffre "2", on aura un "5".

24. L'augmentation des données est utile dans une certaine mesure, mais il est toujours préférable d'avoir plus de données. Le réseau s'entrainera, mais lorsqu'il est testé sur un jeu de données inédit, il ne fonctionnera pas bien, on a certes eu des données mais elles ne sont pas de qualité. De plus des fois celà crée une confusion (tel que le cas décris précédement), ou dan sun autre cas ou la couleur serait deterministe, par exemple, les femmes noires ont 42% plus de risques de mourir du cancer de sein en raison d'un large éventail de facteurs, l'application des changements de couleurs pour augmenter les données ne permetterait pas de détecter celà.

25. Autres types de data augmentation : Rotation, changement de couleurs, changement d'échelle (zoom), symétrie verticale ...

26. Avec cette manipulation, notre accuracy devient beaucoup plus stable, en effet elle augmente dès les premières époques et garde une valeur élevée tout au long. En plus elle ateind sa valeur maximale jusuqe là (par rapport aux étapes précédentes) qui est aux alentours de 68%.
    Le momentum est en effet une méthode permettant d'accélérer les vecteurs de gradients dans les bonnes directions, permettant ainsi une convergence plus rapide. Il s'agit de l'un des algorithmes d'optimisation les plus répandus, et de nombreux modèles ultramodernes sont formés à son utilisation

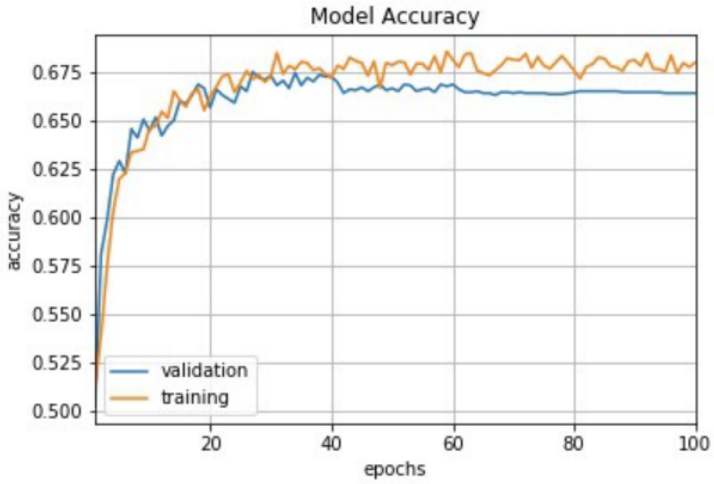


Fig 4a : Exponential Decay Schedule

27. Le fait de prendre en considération la direction du gradient fournie au batch précédent permet d'accélérer les vecteurs de gradients dans les bonnes directions, conduisant ainsi à une convergence plus rapide.
Le fait de faire diminuer le pas d'apprentissage après un certain nombre d'époques permet de diminuer la sévérité de la modification des paramètres en avançant dans l'apprentissage, au début, les paramètres sont aléatoires donc on aimerait apporter de grands changements, mais au fur et à mesure de l'apprentissage ils deviennet plus stables, et si un exemple différent intervient on voudrait pas qu'il déstabilise nos paramètres.

28. Variantes :
    (a) **planification du Learning Rate :**
        — **Learning Rate Constant :**
        C'est la planification par défaut dans l'optimiseur SGD. L'élan et le taux de décroissance sont tous deux mis à zéro par défaut. Il est difficile de choisir le bon taux d'apprentissage. En expérimentant avec la gamme de taux d'apprentissage dans notre exemple, lr = 0,1 montre une performance relativement bonne pour commencer. Cela peut nous servir de base pour expérimenter différentes stratégies de taux d'apprentissage.

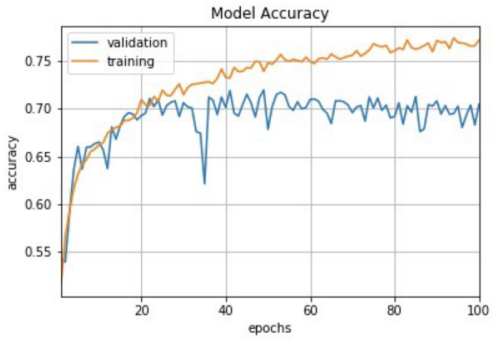


Fig 1 : Constant Learning Rate

        — **Time-Based Decay :**
        La forme mathématique de la décroissance temporelle est $lr = \frac{lr0}{1+kt}$ où lr, k sont des hyperparamètres et t est le nombre d'itérations.

```
learning_rate = 0.1
decay_rate = learning_rate / epochs
momentum = 0.8
sgd = SGD(lr=learning_rate, momentum=momentum, decay=decay_rate, nesterov=False)
```

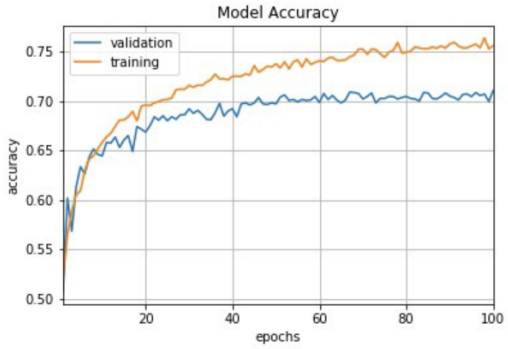


Fig 2 : Time-based Decay Schedule

— **Step Decay :**
Ce dernier diminue le taux d'apprentissage d'un facteur toutes les quelques époques. La forme mathématique de step decay est : $lr = lr0 * drop^{floor\frac{epoch}{epochs-drop}}$

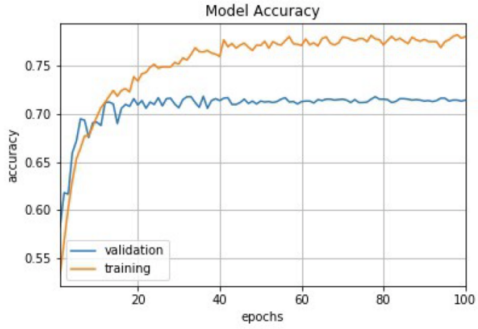


Fig 3a : Step Decay Schedule

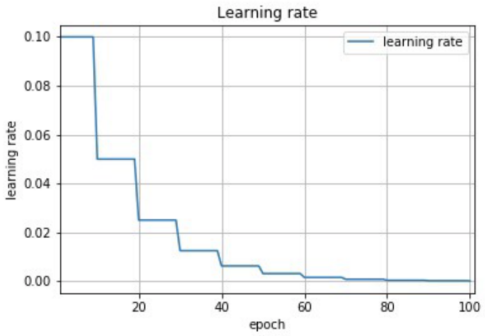


Fig 3b : Step Decay Schedule

— **Exponential Decay :**
Une autre planification courante est la décroissance exponentielle. Elle a la forme mathématique $lr = lr0 * e^{-kt}$, où lr, k sont des hyperparamètres et t est le nombre d'itérations.

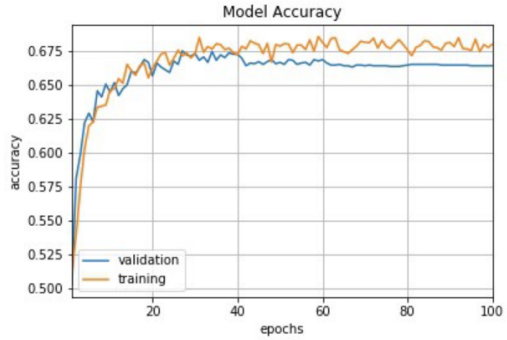


Fig 4a : Exponential Decay Schedule

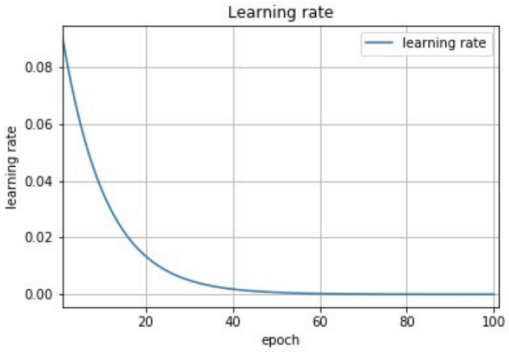


Fig 4b : Exponential Decay Schedule

(b) **Adaptive Learning Rate Methods (variantes de SDG) :**
Le problème que pose l'utilisation des plannings de taux d'apprentissage est que leurs hyperparamètres doivent être définis à l'avance et dépendent fortement du type de modèle et du problème. Un autre problème est que le même taux d'apprentissage est appliqué à toutes les mises à jour de paramètres. Si nous avons des données éparses, nous voudrons peut-être mettre à jour les paramètres dans des proportions différentes.
Des algorithmes de descente de gradient adaptatifs tels que **Adagrad, Adadelta, RMSprop, Adam** offrent une alternative au SGD classique. Ces méthodes de taux d'apprentissage par paramètre fournissent une approche heuristique sans nécessiter un travail coûteux pour le réglage manuel des hyperparamètres pour la planification du taux d'apprentissage.

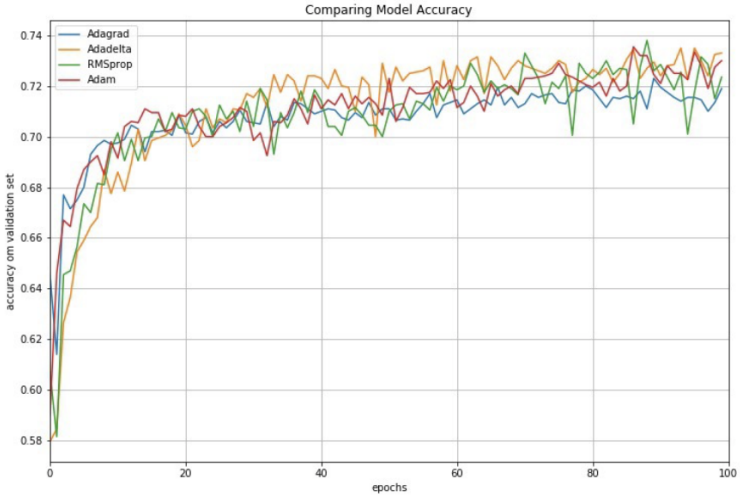


Fig 6 : Comparing Performances of Different Adaptive Learning Algorithms

29. En ajoutant le *dropout*, on gagne déjà en temps d'éxécution, et on a que pour différentes va-

leurs du pas d'apprentissage et de la taille du batch, on a plus de sur apprentissage ou les résultats en test sont médiocres, au contraire, on a obtenu une stabilité des résultats.

30. Régularisation : Les réseaux neuronaux profonds avec un grand nombre de paramètres sont des systèmes d'apprentissage automatique très puissants. Cependant, la suralimentation est un problème sérieux dans de tels réseaux. On trouvera ci-dessous quelques techniques proposées récemment et qui sont devenues une norme générale dans les réseaux de neurones à convolution.
    - Dropout : est une technique permettant de résoudre ce problème. L'idée clé est de supprimer au hasard des unités (ainsi que leurs connexions) du réseau de neurones pendant l'entraînement. La réduction du nombre de paramètres à chaque étape de l'entraînement a un effet de régularisation. Le décrochage a montré des améliorations dans les performances des réseaux de neurones en ce qui concerne les tâches d'apprentissage supervisé dans les domaines de la vision, de la reconnaissance vocale, de la classification des documents et de la biologie computationnelle, obtenant des résultats à la pointe de la technologie sur de nombreux ensembles de données de référence.
    - Kernel Regularizer : permet d'appliquer des pénalités sur les paramètres de la couche lors de l'optimisation. Ces pénalités sont intégrées à la fonction de perte que le réseau optimise. Cet argument en couche convolutive n'est autre que la régularisation L2 des poids. Cela pénalise les poids en pointe et garantit que toutes les entrées sont prises en compte. Lors de la mise à jour des paramètres de descente de gradient, la régularisation ci-dessus de L2 signifie en fin de compte que chaque poids est décomposé de manière linéaire, c'est pourquoi nous appelons décroissance du poids.
    - Batch Normalization : normalise l'activation de la couche précédente à chaque batch, c'est-à-dire qu'elle applique une transformation qui maintient l'activation moyenne proche de 0 et l'écart type d'activation proche de 1. Elle résout le problème du décalage de covariable interne. Elle agit également comme un régularisateur, éliminant dans certains cas la nécessité du dropout. La normalisation par batchs permet d'obtenir la même précision avec moins d'étapes d'entraînement, accélérant ainsi le processus d'entraînement.
31. Dans l'apprentissage automatique, la régularisation est un moyen d'empêcher le sur-apprentissage. La régularisation réduit le sur-apprentissage en ajoutant une pénalité à la fonction de perte. En ajoutant cette pénalité, le modèle est formé de manière à ne pas apprendre un ensemble interdépendant de poids de caractéristiques. Le dropout est une approche de la régularisation dans les réseaux de neurones qui aide à réduire l'apprentissage interdépendant entre les neurones.
    - Le dropout force un réseau de neurones à apprendre des fonctionnalités plus robustes, utiles avec de nombreux sous-ensembles aléatoires différents des autres neurones.
    - Le dropout double approximativement le nombre d'itérations nécessaires pour converger. Cependant, le temps de formation pour chaque époque est inférieur.
    - Avec H unités cachées, chacune pouvant être larguée, nous avons 2^H modèles possibles. En phase de test, l'ensemble du réseau est pris en compte et chaque activation est réduite d'un facteur p.
32. Influence des hyper-paramètres (meilleurs constats) :
    - En règle générale, l'utilisation d'une petite valeur de dropout de 20% à 50% des neurones, 20% constituant un bon point de départ. Une probabilité trop faible a un effet minimal et une valeur trop élevée entraîne un sous-apprentissage du réseau.
    - Utilisation d'un plus grand réseau. On obtiendrait probablement de meilleures performances lorsque le dropout est utilisé sur un réseau plus vaste, ce qui donne au modèle plus l'occasion d'apprendre des représentations indépendantes.

— Utilisation des dropout sur les couches entrantes (visibles) et cachées. L'application du décrochage à chaque couche du réseau a donné de bons résultats.

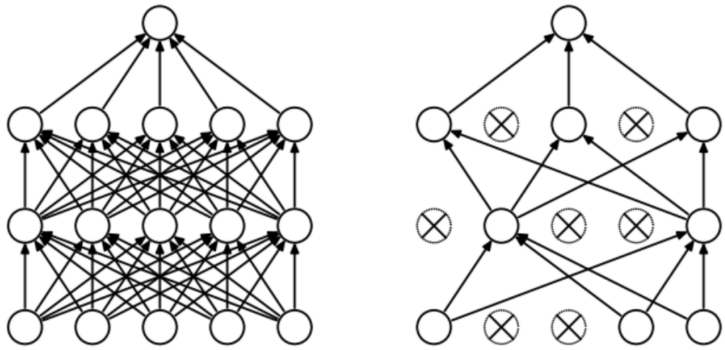
— Utilisation d'un taux d'apprentissage élevé avec décadence et grand élan. Augmentation du taux d'apprentissage par un facteur de 10 à 100 et utilisation d'une valeur momentum élevée de 0,9 ou 0,99.

— Limiter la taille des poids du réseau. Un taux d'apprentissage élevé peut engendrer des poids de réseau très importants. L'imposition d'une contrainte sur la taille des poids du réseau, telle que la régularisation de la norme maximale avec une taille de 4 ou 5, s'est avérée améliorer les résultats.

33. Le dropout en apprentissage et en test :

— Phase d'apprentissage : pour chaque couche masquée, pour chaque échantillon d'apprentissage et pour chaque itération, ignorer (mettre à zéro) une fraction aléatoire, p, de nœuds (et les activations correspondantes).

— Phase de test : Utiliser toutes les activations, mais les réduire d'un facteur p (pour tenir compte des activations manquantes pendant l'entraînement).



(a) Standard Neural Net (b) After applying dropout.

34. En ajoutant encore cette régularisation, les résultats s'améliorent, l'accuracy en test a atteind les 70% et 80% en train.