INF8225 - Intelligence artificielle : techniques probabilistes et d'apprentissage

Hiver 2019

TP No. [2]

Groupe [2]

[1923715] – [BETTACHE Lyes Heythem]

Binome: [19844483] Flore tsafack tsobeng / groupe 3

[17-02-2019]

Partie I

a) Donnez le pseudocode incluant des calculs matriciels—vectoriels détaillés pour l'algorithme de rétropropagation pour calculer le gradient pour les paramètres de chaque couche étant donné un exemple d'entraînement.

Cours9 Réseaux de neurones page 16 inf8215

POLYTECHNIQUE MONTRÉAL

repeat
  for chaque exemplaire $\mathbf{x}^{(t)}$ **do**
    for chaque noeud i de la couche d'entrée **do**
      $a_i \leftarrow x_i^{(t)}$
    end for
    for $\ell = 2, \ldots, L$ **do**
      for chaque noeud j dans la couche ℓ **do**
        $in_j \leftarrow \sum_i w_{ij} a_j$
        $a_j \leftarrow \phi(in_j)$
      end for
    end for
    $\Delta[N] \leftarrow -\partial Loss / \partial in_N$ // N est le neurone de la dernière couche
    for $\ell = L-1, \ldots, 1$ **do**
      for chaque noeud i dans la couche ℓ **do**
        $\Delta[i] \leftarrow \phi(in_i)(1 - \phi(in_i)) \sum_j w_{ij} \Delta[j]$
      end for
    end for
    for chaque poids w_{ij} dans le réseau **do**
      $w_{ij} \leftarrow w_{ij} + \eta a_i \Delta[j]$
    end for
  end for
until critère d'arrêt pas satisfait

Daniel Aloise <daniel.aloise@polymtl.ca> (d'après les slides de H. Larochelle, Google) — Rés... novemb...
16/39

POLYTECHNIQUE MONTRÉAL LE GÉNIE EN PREMIÈRE CLASSE

Les caractéristiques de notre réseau sont :

- une couche d'entrée avec D = 100(unités)
- $L\ couches\ caches\ avec\ M = 100\ (unités)$
- y vecteur de sortie de dimension k (k classe)
- x_i ième vecteur de l'ensemble d'apprentissage
- f = [f…f_k] fonction d'activation de la couche finale (f_k étant une sigmoïde)
- $\mathrm{a}^{(l)} = \left[\mathrm{a}^{(1)} \ldots . \mathrm{a}^{(k)}\right]^T$ vecteur de pré-activation
- ${h_k}^{(l)}(\mathrm{a}^{(l)}\ (\mathrm{x_i}))$fonction d'activation de la couche cachée

<u>étant donné un exemple d'entraînement</u>

<u>Propagation avant :</u>

- Initialiser tous les poids W aléatoirement entre l'intervalle [0 1]

Wij = $\begin{bmatrix} W_{11} & \cdots & W_{1D} \\ \vdots & \ddots & \vdots \\ W_{1D} & \cdots & W_{DD} \end{bmatrix}$

Pour chaque couche l de L couches cachées, on doit :

L'entrée de la fonction d'activation des couches cachées

$$IN(l) = W(l) * x_i = \begin{bmatrix} W_{11} & \cdots & W_{1D} \\ \vdots & \ddots & \vdots \\ W_{1D} & \cdots & W_{DD} \end{bmatrix} * \begin{bmatrix} xi1 \\ xi2 \\ . \\ . \\ xiD \end{bmatrix}$$

L'activation des couches cachées

$$A(l) = h^{(l)}(IN(l)) = \begin{bmatrix} \dfrac{1}{1+\exp(-In1)} \\ . \\ . \\ \dfrac{1}{1+\exp(-InD)} \end{bmatrix}$$

<u>Avec:</u>

$IN(l)$ représente un vecteur des entrées de toutes les unités d'une couche cachée, même taille que le vecteur d'entrée x_i.

$W(l)$ représente les poids associés aux transitions entre 2 couches.

A[*l*] le vecteur des sorties de chaque couche l.

$h^{(l)}(x)$ la fonction sigmoïde d'activation qui on l'appliquer de façon distincte sur chaque élément $a_{1..D}$ du vecteur A[*l*].

Remarque : Si on tient compte le biais, pour calculer les entrées des unités en modifiant le vecteur d'entrée x_i afin de concaténer 1 à la suite du vecteur. Nous avons :

$$IN(l) = \Theta^{(l)} * x_i = \begin{bmatrix} W_{11} & \cdots & W_{1D} \\ \vdots & \ddots & \vdots \\ W_{D1} & \cdots & W_{DD} \end{bmatrix} \begin{bmatrix} W_{11} & \cdots & W_{1D} \\ \vdots & \ddots & \vdots \\ W_{D1} & \cdots & W_{DD} \\ b_1 & \dots & b_D \end{bmatrix} * \begin{bmatrix} xi1 \\ xi2 \\ . \\ . \\ xiD \end{bmatrix}$$

- Calcul d'activation des couches de sorties

$$\mathrm{IN_{sortie}} = \mathrm{W_{sortie}} * \mathrm{x_i} = \begin{bmatrix} W1 \\ . \\ . \\ WD \end{bmatrix}^T * \begin{bmatrix} xi1 \\ . \\ . \\ xiD \end{bmatrix}$$

Le vecteur A[*l*] devient ensuite le vecteur d'entrée x_i de la couche suivante pour l'exécution de la boucle.

<u>Propagation arrière</u>

-Initialiser le gradient en faisant une différence entre la cible en sortie d'un exemple i à la valeur attendue

en partant de la dernière couche vers la première couche.

$$\Delta_{\mathrm{i}} = -2h^{(l)}(\mathrm{IN_{sortie}})\vec{f}(\mathrm{IN_{sortie}})\,(1 - \vec{f}(\mathrm{IN_{sortie}}))(\vec{y} - \vec{f}(\mathrm{IN_{sortie}})\,)$$

Calcul de dérivée :

$$L_i = \left(y_i - f(x_i)\right)^T (y_i - f(x_i))$$

$$\Delta_\mathrm{i} = \frac{d\ L_i}{d\Theta^{(l)}}$$

L_{ik}=$\left(y_{k,i} - f_k(x_i)\right)^2$

En utilisant la règle de la dérivée en chaine

$$\frac{d\ L_{ik}}{d\Theta^{(l)}} = \frac{d\ \mathrm{a}^{(l)}}{d\Theta^{(l)}} \frac{d\ f(x_i)}{d\mathrm{a}^{(l)}} \frac{d\ L_{ik}}{df(x_i)}$$

1- $\frac{d\ L_{ik}}{df(x_i)} = -2(y_{k,i} - f_k(x_i)) = -2 \begin{bmatrix} y_{k,1} - f_k(x_1) \\ y_{k,2} - f_k(x_2) \\ . \\ . \\ y_{k,N} - f_k(x_N) \end{bmatrix} \Rightarrow \nabla_f L = -2\ (\vec{y} - \vec{f}\ )$

2- f_k étant une sigmoïde

$\frac{d\ f_k(x_i)}{d\mathrm{a}^{(l)}} = f_k(x_i)(1 - f_k(x_i))$ = $\begin{bmatrix} f_k(x_1)(1 - f_k(x_1) \\ f_k(x_2)(1 - f_k(x_2) \\ . \\ . \\ f_k(x_N)(1 - f_k(x_N) \end{bmatrix} \Rightarrow \nabla_a f = \vec{f}(x_i)\ (1 - \vec{f}(x_i))$

3-

$\frac{d\ a^{(l)}}{d\Theta^{(l)}} = \frac{d}{d\Theta^{(l)}} \sum_j {\Theta_j}^{(l)} * h_j = h_j \Rightarrow \nabla_{\Theta^{(l)}} a^{(l)} = h^{(l)}(IN(l)) = \begin{bmatrix} \frac{1}{1+\exp(-In1)} \\ . \\ . \\ \frac{1}{1+\exp(-InD)} \end{bmatrix}$

$$\Delta_\mathrm{i} = -2 \begin{bmatrix} \frac{1}{1+\exp(-In1)} \\ . \\ . \\ \frac{1}{1+\exp(-InD)} \end{bmatrix} \begin{bmatrix} f_k(x_1)(1 - f_k(x_1) \\ f_k(x_2)(1 - f_k(x_2) \\ . \\ . \\ f_k(x_N)(1 - f_k(x_N) \end{bmatrix} \begin{bmatrix} y_{k,1} - f_k(x_1) \\ y_{k,2} - f_k(x_2) \\ . \\ . \\ y_{k,N} - f_k(x_N) \end{bmatrix}$$

$$\Delta_\mathrm{i} = -2h^{(l)}(\mathrm{IN_{sortie}})\vec{f}(\mathrm{IN_{sortie}})\ (1 - \vec{f}(\mathrm{IN_{sortie}}))(\ \vec{y} - \vec{f}(\mathrm{IN_{sortie}})\ )$$

Rétropropager le Δ à la couche précédente (la dernière couche L)

On doit multiplier tous les éléments du vecteur de poids W_{sortie} par le Δ, puis par la dérivée de la fonction d'activation effectuée sur la couche précédente (Les multiplications faites élément par élément).

Ensuite, réactualiser les poids de W_{sortie}

en multipliant les sorties de la couche précédente contenues dans A[L – 1] par les quantités scalaires Δ et α et en additionnant le tout à W_{sortie}.

$$\Delta_{\mathrm{j}} = \vec{A}(\mathrm{l}-1)\ (1\ -\ \vec{A}(\mathrm{l}-1))\Delta_{\mathrm{i}}\mathrm{W}_{\mathrm{sortie}}$$

Rétropropager aux autres couches. (en commençant par L-2 jusqu'à 0)

$$\Delta_{\mathrm{k}} = \vec{A}(\mathrm{l})\ (1\ -\ \vec{A}(\mathrm{l}))\Delta_{\mathrm{j}}W(l+1)$$

-Rafraichir les poids

$$W(l+1) = W(l+1) + \boldsymbol{\alpha}\,\vec{A}(\mathrm{l})\ \Delta_{\mathrm{j}}^{T}$$

$$\Delta_{\mathrm{j}} = \Delta_{\mathrm{k}}$$

Avec $\boldsymbol{\alpha}$ = taux d'apprentissage.

<u>En résumé, le pseudocode suivant pour un exemple :</u>

$x_{\text{entrée}}$ =x_i

Pour chaque couche *l* de L couches cachées, faire :
    $In[l]$=$W[l]$× $x_{\text{entrée}}$
    $A[l]$=$h(l)$($In[l]$)
    $x_{\text{entrée}}$ =$A[l]$
$IN sortie$=$\mathrm{W}_{\mathrm{sortie}}$ × $x_{\text{entrée}}$

// Commencer la rétropropagation

$$\Delta_{\mathrm{i}} = -2h^{(l)}(\mathrm{IN}_{\mathrm{sortie}})\vec{f}(\mathrm{IN}_{\mathrm{sortie}})\ (1\ -\ \vec{f}(\mathrm{IN}_{\mathrm{sortie}}))(\ \vec{y} - \vec{f}(\mathrm{IN}_{\mathrm{sortie}})\ )$$

Δj=$A[L−1]$∗(1−$A[L−1]$)∗Δ$i\mathrm{W}_{\mathrm{sortie}}$
$\mathrm{W}_{\mathrm{sortie}}$=$\mathrm{W}_{\mathrm{sortie}}$+$\alpha$∗Δ$i$ $A[L−1]$

Pour chaque couche *l* de L en commençant par L-2 jusqu'à 0 :

$$\Delta_{\mathrm{k}} = \vec{A}(\mathrm{l})\ (1\ -\ \vec{A}(\mathrm{l}))\Delta_{\mathrm{j}}W(l+1)$$

$$W(l+1) = W(l+1) + \boldsymbol{\alpha}\,\vec{A}(\mathrm{l})\ \Delta_{\mathrm{j}}^{T}$$

$$\Delta_j = \Delta_k$$

Remarque : On devrait ainsi pouvoir reprendre le vecteur d'entrées **x**i et évaluer la sortie du prochain exemple.

b) (4 points) Imaginez que vous avez maintenant un jeu de données avec N=500,000 exemples. Expliquez comment vous utiliserez votre pseudocode pour optimiser ce réseau neuronal dans le contexte d'une expérience d'apprentissage machine correctement effectuée.

Une bonne façon d'optimiser l'expérience d'apprentissage machine avec ce réseau de neurones serait de faire la descente de gradient stochastique avec des mini-batches comme dans le premier TP du cours.

Au début on doit former quelques batches en séparant les N exemples.

Ensuite, on applique le pseudocode vu en a) et donc produire des sorties et rétropropager la différence sur la cible pour tous les exemples d'une batch (au lieu de le faire sur tout l'ensemble des données).

Partie 2

Note générale :Il est important de mentionner que nous nous sommes servies de différentes ressources dans cette partie afin d'expérimenter différentes architectures du réseau de neurone de la banque de données fashionMnist . Il s'agit entre autres de :

1- Article :' VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION' https://arxiv.org/pdf/1409.1556.pdf?fbclid=IwAR0i3WwWVjQQKL_294PyoVw7J3LWrPIAT685VHCpnIV_ff8JmEVcI-3P2BI
2- medium
https://medium.com/tensorflow/hello-deep-learning-fashion-mnist-with-keras-50fcff8cd74a?fbclid=IwAR2eCCEI2AUncHnBmvUPf0paUmQ7nXIrrFkrS_2-AnQES9aujBeqh6Bf3fQ
3- Xfan1025
https://github.com/Xfan1025/Fashion-MNIST/blob/master/fashion-mnist.ipynb
4- 'How to calculate the number of parameters in the CNN?' https://medium.com/@iamvarman/how-to-calculate-the-number-of-parameters-in-the-cnn-5bd55364d7ca

5- 'ensorFlow & Deep Learning – Episode 3 – Modifiez votre Réseau de Neurones en toute simplicité' https://blog.xebia.fr/2017/04/11/tensorflow-deep-learning-episode-3-modifiez-votre-reseau-de-neurones-en-toute-simplicite/

Dans notre TP nous avons essayé plusieurs d'architecture de type architecture de réseau de neurones convolutifs, qu'est formée par un empilement de couches de traitement :

Input Layer: qui représente la couche d'entrée et dont le role tout simplement est la lecture de l'image

Convolutional Layer : La **couche de convolution** est la composante clé des réseaux de neurones convolutifs, et constitue toujours au moins leur première couche.

Son but est de repérer la présence d'un ensemble de *features* dans les images reçues en entrée. Pour cela, on réalise un filtrage par convolution : le principe est de faire "glisser" une fenêtre représentant la *feature* sur l'image, et de calculer le produit de convolution entre la *feature* et chaque portion de l'image balayée. Une *feature* est alors vue comme un filtre : les deux termes sont équivalents dans ce contexte.

Pooling Layer : condensation de l'information. On ne cherche pas à connaître l'emplacement exact d'un pattern, sa localisation approximative suffit. Il est donc courant de faire suivre une couche de convolution par une phase de pooling qui va condenser l'information et réduire la dimension des phases intermédiaires (on va par exemple garder la valeur maximale de 4 neurones d'une même zone).

Fully-connected Layer: Ce type de couche reçoit un vecteur en entrée et produit un nouveau vecteur en sortie. Pour cela, elle applique une combinaison linéaire puis éventuellement une fonction d'activation aux valeurs reçues en entrée.

Dropout : entre chaque couche dense, il est commun d'utiliser du *dropout*. C'est une technique de régularisation (pour combattre l'overfitting) dont le principe est de désactiver aléatoirement à chaque itération un certain pourcentage des neurones d'une couche. Cela évite ainsi la sur-spécialisation d'un neurone (et donc l'apprentissage par coeur).

Output Layer: La dernière couche *fully-connected* permet de classifier l'image en entrée du réseau : elle renvoie un vecteur de taille K, où K est le nombre de classes dans notre problème de classification d'images. Chaque élément du vecteur indique la probabilité pour l'image en entrée d'appartenir à une classe.

la dernière couche est suivie d'une couche dense où tous les neurones sont liés entre eux, pour enfin appliquer une fonction *softmax*.

Une petite explication des valeurs choisies pour l'architecture :

L'objectif est d'être capable, d'une couche à l'autre, de réduire l'information pour aller à l'essentiel de ce que contient l'image. Mais même si l'information est réduite, il faut toujours être capable de capter les différentes formes qui composent une image et surtout être capable de les distinguer d'une classe à l'autre.

C'est pourquoi il n'est pas rare, d'une couche de convolution à l'autre, d'augmenter le nombre de *channels* (donc de filtres appliqués à l'image) et en même temps de réduire la taille de 'l'image' filtrée par la suite.

La taille de l'image de sortie reste la même, et c'est l'opération de *max pooling* qui nous permet de condenser l'information (en divisant la taille de l'image par deux dans notre cas). Le même principe est appliqué pour la seconde couche de convolution et les autres couches suivantes, puis on divise la taille de l'image par deux grâce à du *max pooling*. Ainsi, nous sommes capables de condenser l'information (en réduisant la taille des « images » intermédiaires) tout en ayant un mapping de plus en plus complet des différentes formes à distinguer dans l'image (en augmentant le nombre de *channels*).

Une fois les étapes de convolutions terminées, les neurones restants sont éclatés en un seul vecteur pour repasser à une couche dense, jusqu'à appliquer le *softmax*

Une fois cette architecture comprise, il est ensuite très simple d'encapsuler d'autres couches de convolution et d'autres couches denses pour améliorer les résultats du modèle.

Remarque :

Normaliser des données permet de chaque caractéristique ait une influence équivalente sur le résultat final, malgré les éventuelles différences d'échelles de mesure pour les caractéristiques des données entrantes, donc nous avons ajouté après la couche d'entrée une couche qui normalise nos données pour chaque expérience.

https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html

-pour l'expérience 1 a 6 nous avons pris 'epochs = 10' et 'batch_size = 256'

Et pour l'expérience 7 'epochs = 10' et 'batch_size = 20'

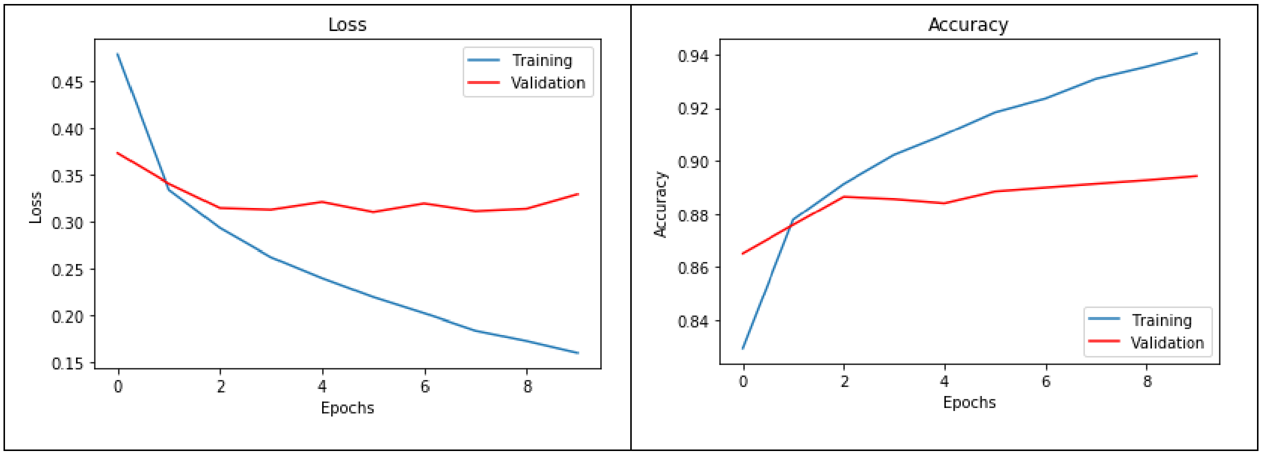
Dans l'expérience 1 et 2 nous avons juste utilisé un seul type de couche *fully connected*, pour voir si plus de couches du même type changeait de façon positive le résultat et si la couche **Dropout** avait un gros impact sur notre entraînement.

Expérience 1 :

Au début l'architecture caché qui a été essayée se compose de 4 couches complètement connectées (*fully connected*) qui sont activées par des fonctions ReLUs, ce qui fait que les noeuds les employant sont dits *Rectified Linear Units*. Les couches sont progressivement réduites en taille, ce qui permet de doucement converger vers le nombre de classes (10) du Fashion MNIST.

Voici l'architecture utilisée :

```
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_10 (Batc (None, 1, 28, 28)         112
_________________________________________________________________
flatten_5 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_18 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_19 (Dense)             (None, 256)               131328
_________________________________________________________________
dense_20 (Dense)             (None, 128)               32896
_________________________________________________________________
dense_21 (Dense)             (None, 64)                8256
_________________________________________________________________
dense_22 (Dense)             (None, 10)                650
=================================================================
Total params: 575,162
Trainable params: 575,106
Non-trainable params: 56
_________________________________________________________________
None
```



On remarque que l'ensemble de validation n'a pas bien suivie l'ensemble d'entraînement en ce qui concerne les pertes. Le réseau est rapidement surentraîné (l'overfitting) sur l'ensemble d'entraînement et n'est pas aussi efficace lorsqu'il a affaire à la validation ou aux tests.

```
test_labels = y_test
pred = model.predict(test_features)
# convert predicions from categorical back to 0...9 digits
pred_digits = np.argmax(pred, axis=1)
```

```
accuracy_score(test_labels, pred_digits)
```

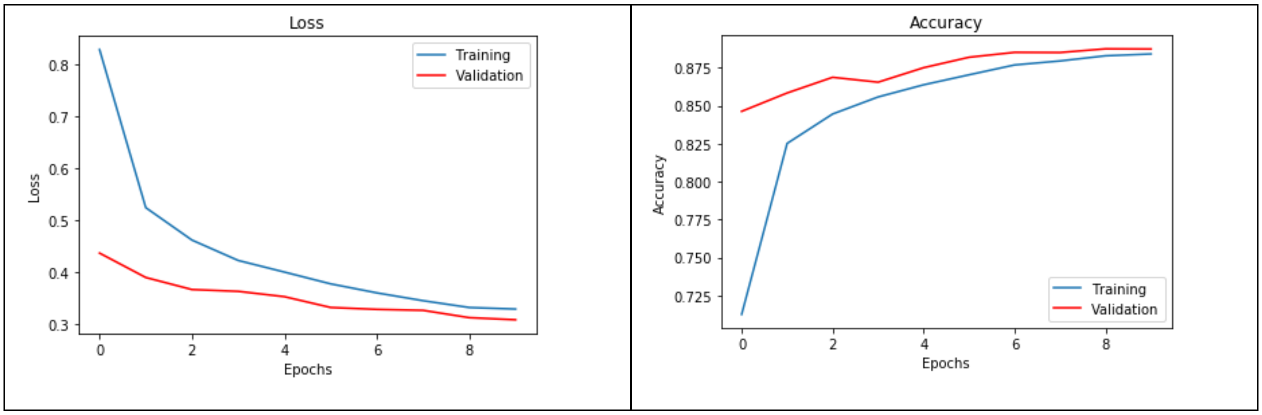
```
0.8862
```

Expérience 2

Dans cette expérience nous avons pris la même architecture de l'expérience1 sauf pour les 4 couches complètement connectées sont suivies de fonctions de *Dropout()* qui laissent tomber certains noeuds afin de prévenir le réseau de se surspécifier(l'overfitting) sur son ensemble d'entraînement.

Voici l'architecture utilisée :

```
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_12 (Batc (None, 1, 28, 28)         112
_________________________________________________________________
flatten_8 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_33 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_11 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_34 (Dense)             (None, 256)               131328
_________________________________________________________________
dropout_12 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_35 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_13 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_36 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_14 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                650
=================================================================
Total params: 575,162
Trainable params: 575,106
Non-trainable params: 56
_________________________________________________________________
None
```



On remarque que l'ensemble de validation a été moyenne bien suivre l'ensemble d'entraînement en ce qui concerne les pertes et la précision.

```
test_labels = y_test
pred = model.predict(test_features)
# convert predicions from categorical back to 0...9 digits
pred_digits = np.argmax(pred, axis=1)
```

```
accuracy_score(test_labels, pred_digits)
```

```
0.8767
```

Remarque : d'après l'expérience 1 et 2 on remarque que la couche Dropout a été bien éliminé l'overfitting

Afin d'augmenter le nombre de couches du même type, nous avons modifié l'expérience précédente en augmentant le nombre de couche et nous avons observé une baisse importante de la précision sur les ensembles de validation et de test. Ce problème vient du fait que chaque noeud d'une couche est connecté à tous les nœuds de la couche suivante. Ceci nous permet d'affirmer que l'augmentation des couches intermédiaires rend difficile la mise à jour des poids surtout au niveau de la retropropagation

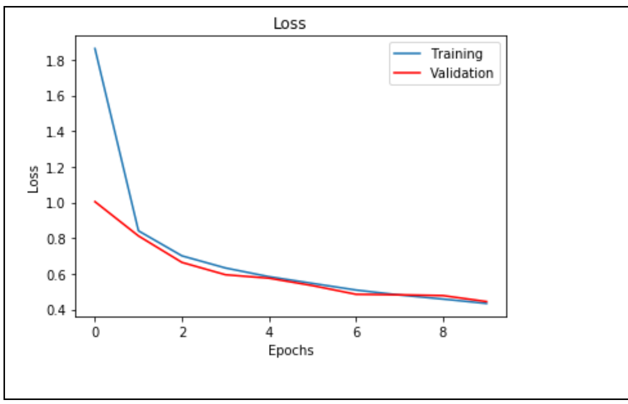
Expérience 3

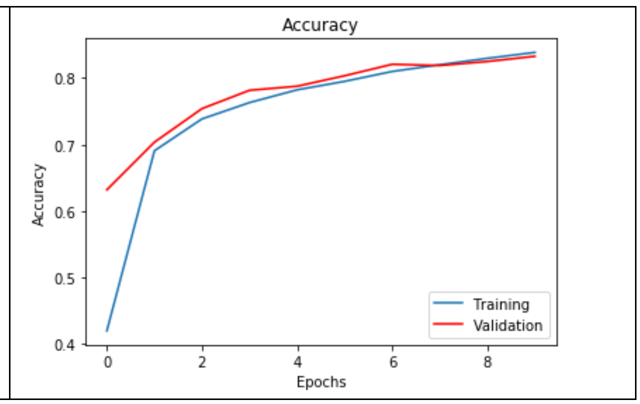
Dans cette expérience nous avons deux couches convolutives, toutes deux immédiatement suivies d'une couche de mise en commun (*pooling*). Les couches convolutives permettent d'associer les caractéristiques de l'image à une région de l'image en rendant les noeuds d'une couche sensible à une section spécifique des données provenant de la couche précédente. On suit souvent les couches convolutives par des couches de *pooling* afin de prévenir le surentraînement qui ferait qu'on aurait une excellente performance sur l'ensemble d'entraînement, mais un résultat nettement moins bon sur l'ensemble de validation ou de test. Faire la mise en commun (*pooling*) de certains pixels équivaut à brouiller l'image de la couche précédente.

A la fin on a regroupé progressivement les données à l'aide de 3 couches complètement connectées (*fully connected*) afin de faire une fonction de *softmax()* sur 10 groupes.

Voici l’architecture utilisée :

```
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_3 (Batch (None, 1, 28, 28)         112
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 40, 28, 28)        1040
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 40, 14, 14)        0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 70, 14, 14)        25270
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 200, 14, 14)       126200
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 200, 7, 7)         0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 250, 5, 5)         450250
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 250, 2, 2)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1000)              0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               100100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1010
=================================================================
Total params: 724,182
Trainable params: 724,126
Non-trainable params: 56
_________________________________________________________________
None
```





On remarque que l’ensemble de validation a été bien suivie l’ensemble d’entraînement en ce qui concerne les pertes et la précision.

```
test_labels = y_test
pred = model.predict(test_features)
# convert predicions from categorical back to 0...9 digits
pred_digits = np.argmax(pred, axis=1)
```

```
accuracy_score(test_labels, pred_digits)
```

```
0.8264
```

Expérience4

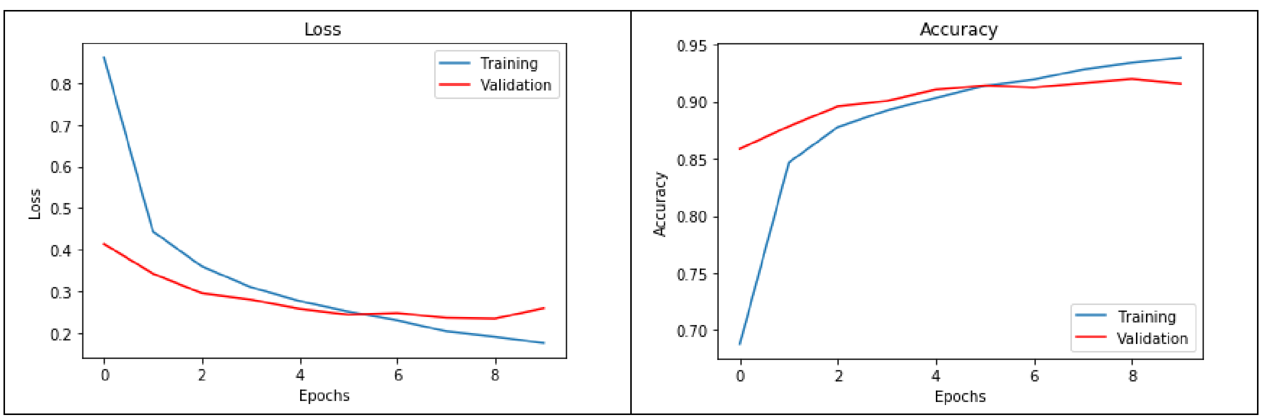
Dans cette expérience nous avons pris la même architecture de l'expérience 3 sauf pour les 3 couches complètement connectées sont suivies de fonctions de *Dropout()* qui laissent tomber certains noeuds afin de prévenir le réseau de se surspécifier sur son ensemble d'entraînement.

Voici l'architecture utilisée :

Layer (type)	Output Shape	Param #
batch_normalization_3 (Batch	(None, 1, 28, 28)	112
conv2d_15 (Conv2D)	(None, 40, 28, 28)	1040
max_pooling2d_12 (MaxPooling	(None, 40, 14, 14)	0
conv2d_16 (Conv2D)	(None, 70, 14, 14)	25270
conv2d_17 (Conv2D)	(None, 200, 14, 14)	126200
max_pooling2d_13 (MaxPooling	(None, 200, 7, 7)	0
conv2d_18 (Conv2D)	(None, 250, 5, 5)	450250
max_pooling2d_14 (MaxPooling	(None, 250, 2, 2)	0
flatten_6 (Flatten)	(None, 1000)	0
dense_18 (Dense)	(None, 180)	180180
dropout_1 (Dropout)	(None, 180)	0
dense_19 (Dense)	(None, 100)	18100
dropout_2 (Dropout)	(None, 100)	0
dense_20 (Dense)	(None, 100)	10100
dropout_3 (Dropout)	(None, 100)	0
dense_21 (Dense)	(None, 10)	1010

```
Total params: 812,262
Trainable params: 812,206
Non-trainable params: 56

None
```



On remarque que la validation avait tout d'abord moins de pertes que l'entraînement, ce qui s'est inversé durant les 3 dernières epochs. Une explication plausible à ce phénomène est que beaucoup de couches visant à rendre le réseau résistant au changement dans les données d'entrée, comme la normalisation ou encore les fonctions de ***Dropout*** ont été appliquées. Conséquemment, les poids ont mis plus de temps à s'ajuster correctement à l'ensemble d'entraînement.

```python
test_labels = y_test
pred = model.predict(test_features)
# convert predicions from categorical back to 0...9 digits
pred_digits = np.argmax(pred, axis=1)
```

```python
accuracy_score(test_labels, pred_digits)
```

```
0.9092
```

Remarque : d’après l’expérience 3 et 4 on remarque que la précision de l’ensemble test a été augmenté de 82,64 % à 90,92 % mais

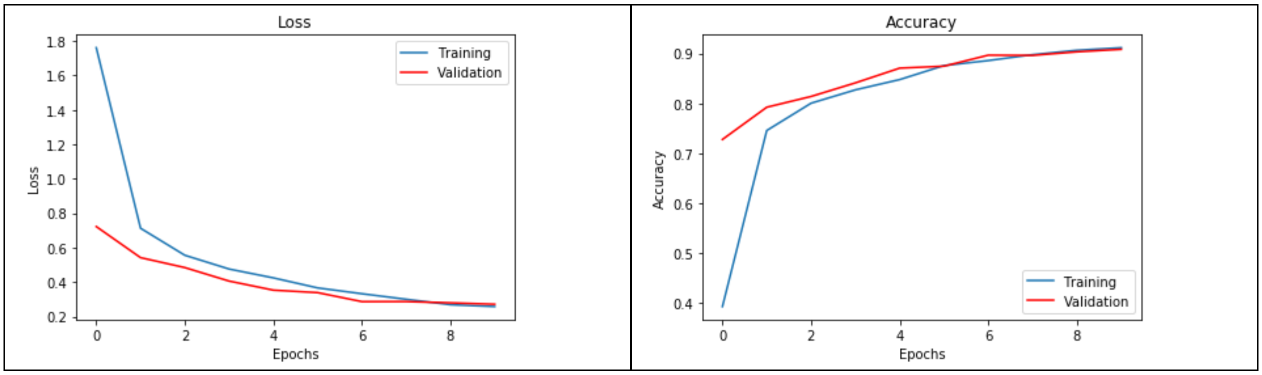
Expérience5

Dans cette expérience nous avons 6 couches convolutives, toutes 2 immédiatement suivies d’une couche de mise en commun (*pooling*) et toutes 2 paire 2 immédiatement suivies d’une couche de mise en commun (*pooling*). Les couches convolutives permettent d’associer les caractéristiques de l’image à une région de l’image en rendant les noeuds d’une couche sensible à une section spécifique des données provenant de la couche précédente. On suit souvent les couches convolutives par des couches de *pooling* afin de prévenir le surentraînement qui ferait qu’on aurait une excellente performance sur l’ensemble d’entraînement, mais un résultat nettement moins bon sur l’ensemble de validation ou de test. Faire la mise en commun (*pooling*) de certains pixels équivaut à brouiller l’image de la couche précédente.

A la fin on a regroupé progressivement les données à l’aide de 3 couches complètement connectées (*fully connected*) afin de faire une fonction de *softmax()* sur 10 groupes. Ces les 3 couches complètement connectées sont suivies de fonctions de *Dropout* qui laissent tomber certains noeuds afin de prévenir le réseau de se surspécifier sur son ensemble d’entraînement.

Voici l’architecture utilisée :

```
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_9 (Batch (None, 1, 28, 28)         112
_________________________________________________________________
conv2d_59 (Conv2D)           (None, 64, 28, 28)        640
_________________________________________________________________
max_pooling2d_38 (MaxPooling (None, 64, 14, 14)        0
_________________________________________________________________
conv2d_60 (Conv2D)           (None, 128, 12, 12)       73856
_________________________________________________________________
max_pooling2d_39 (MaxPooling (None, 128, 6, 6)         0
_________________________________________________________________
conv2d_61 (Conv2D)           (None, 256, 6, 6)         295168
_________________________________________________________________
conv2d_62 (Conv2D)           (None, 256, 6, 6)         590080
_________________________________________________________________
max_pooling2d_40 (MaxPooling (None, 256, 3, 3)         0
_________________________________________________________________
conv2d_63 (Conv2D)           (None, 512, 3, 3)         1180160
_________________________________________________________________
conv2d_64 (Conv2D)           (None, 512, 3, 3)         2359808
_________________________________________________________________
max_pooling2d_41 (MaxPooling (None, 512, 1, 1)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_9 (Dense)              (None, 256)               131328
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_10 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_11 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_6 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_12 (Dense)             (None, 10)                650
=================================================================
Total params: 4,672,954
Trainable params: 4,672,898
Non-trainable params: 56
_________________________________________________________________
None
```



On remarque que l'ensemble de validation a été bien suivre l'ensemble d'entraînement en ce qui concerne les pertes et la précision.

```
test_labels = y_test
pred = model.predict(test_features)
# convert predicions from categorical back to 0...9 digits
pred_digits = np.argmax(pred, axis=1)
```

```
accuracy_score(test_labels, pred_digits)
```

```
0.9062
```

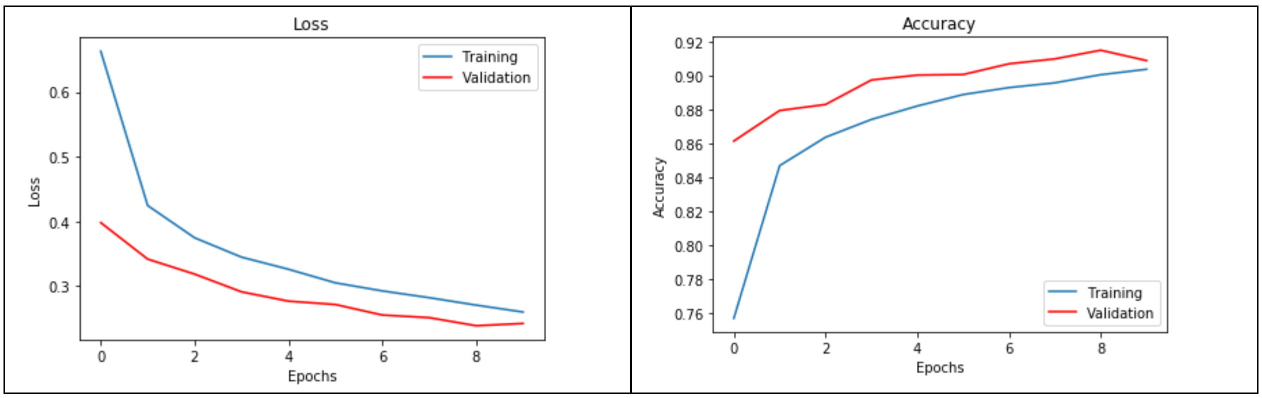
Expérience 6

Dans cette expérience nous avons 2 couches convolutives, toutes 2 immédiatement suivies d'une couche de mise en commun (*pooling*) et de fonctions de *Dropout()* qui laissent tomber certains noeuds afin de prévenir le réseau de se surspécifier sur son ensemble d'entraînement. Les couches convolutives permettent d'associer les caractéristiques de l'image à une région de l'image en rendant les noeuds d'une couche sensible à une section spécifique des données provenant de la couche précédente. On suit souvent les couches convolutives par des couches de *pooling* afin de prévenir le surentraînement qui ferait qu'on aurait une excellente performance sur l'ensemble d'entraînement, mais un résultat nettement moins bon sur l'ensemble de validation ou de test. Faire la mise en commun (*pooling*) de certains pixels équivaut à brouiller l'image de la couche précédente.

A la fin on a regroupé progressivement les données à l'aide de 1 couche complètement connectée (*fully connected*) afin de faire une fonction de *softmax()* sur 10 groupes. Cette couche complètement connectée est suivie de fonction de *Dropout* qui laissent tomber certains noeuds afin de prévenir le réseau de se surspécifier sur son ensemble d'entraînement.

Voici l'architecture utilisée :

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_13 (Batc (None, 1, 28, 28)         112
_________________________________________________________________
conv2d_65 (Conv2D)           (None, 64, 28, 28)        320
_________________________________________________________________
max_pooling2d_42 (MaxPooling (None, 64, 14, 14)        0
_________________________________________________________________
dropout_15 (Dropout)         (None, 64, 14, 14)        0
_________________________________________________________________
conv2d_66 (Conv2D)           (None, 32, 12, 12)        18464
_________________________________________________________________
max_pooling2d_43 (MaxPooling (None, 32, 6, 6)          0
_________________________________________________________________
dropout_16 (Dropout)         (None, 32, 6, 6)          0
_________________________________________________________________
flatten_9 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_38 (Dense)             (None, 256)               295168
_________________________________________________________________
dropout_17 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_39 (Dense)             (None, 10)                2570
=================================================================
Total params: 316,634
Trainable params: 316,578
Non-trainable params: 56
_________________________________________________________________
None
```




```
test_labels = y_test
pred = model.predict(test_features)
# convert predicions from categorical back to 0...9 digits
pred_digits = np.argmax(pred, axis=1)
```

```
accuracy_score(test_labels, pred_digits)
```

```
0.9071
```

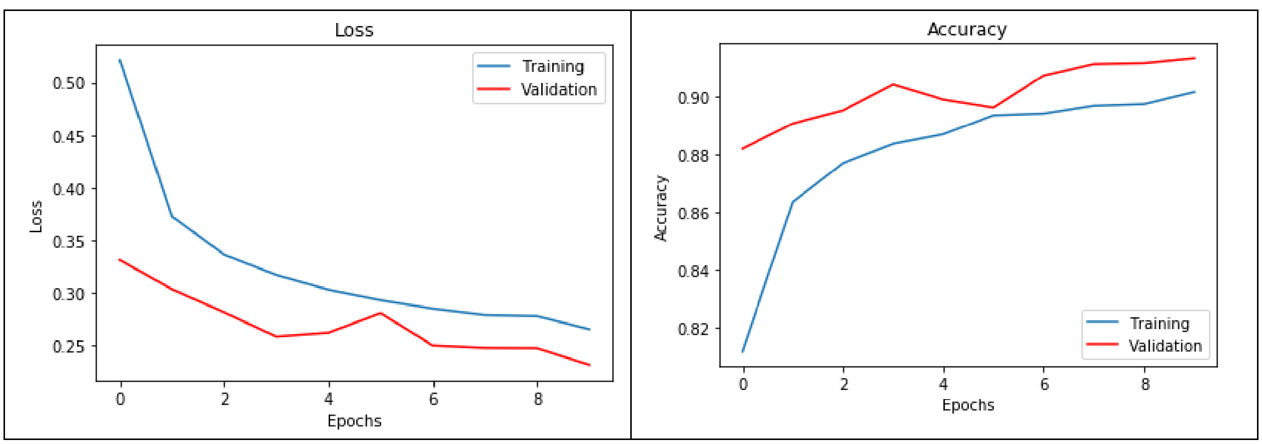
d'après l'expérience 5 et 6 on peut se rendre compte que l'ajout de couches intermédiaires supplémentaires ne permet pas toujours d'améliorer les résultats, et c'est parfois même le contraire. plus on ajoute de couche dense, plus il y a de paramètres à estimer (le nombre de paramètres explose très rapidement). De plus, plus il y a de couches intermédiaires, plus la mise à jour des poids reliant ces dernières par *back-propagation* devient difficile car les vitesses de mises à jour sont différentes entre les couches finales et les premières couches

Expérience 7

Dans cette expérience nous avons pris la même architecture de l'expérience 6 sauf qu'on a changé le mini-batches (20 exemple par rapport 256 pour l'expérience 6).

Voici l'architecture utilisée :

```
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_1 (Batch (None, 1, 28, 28)         112
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 28, 28)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 64, 14, 14)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 64, 14, 14)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 12, 12)        18464
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 32, 6, 6)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 32, 6, 6)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               295168
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 316,634
Trainable params: 316,578
Non-trainable params: 56
_________________________________________________________________
None
```




```python
test_labels = y_test
pred = model.predict(test_features)
# convert predicions from categorical back to 0...9 digits
pred_digits = np.argmax(pred, axis=1)
```

```python
accuracy_score(test_labels, pred_digits)
```

```
0.9099
```

Dans le premier TP1, les expériences avec différentes tailles de mini-batches montraient que les pertes sur l'ensemble de validation suivent moins les pertes sur l'ensemble d'entraînement lorsqu'on a de plus petites mini-batches, mais que la précision sur l’ensemble de validation augmente plus vite (car on met à jour les poids plus souvent). Cet effet a encore pu être observé, car les premières précisions sur l’ensemble de validation sont plus proches de la précision finale observée dans cet expérience (88% à 91%) que dans l’expérience 6 (86% à 90%).

Conclusion

En conclusion, nous pouvons retenir que on peut se rendre compte que l’ajout de couches intermédiaires supplémentaires garanti pas de bons résultats. Ceci n’est pourtant pas le cas en pratique car un ajout de couche permet généralement d’apprendre des fonctions beaucoup plus complexes. Pourtant, l’ajout de ces couches permet en théorie d’apprendre des fonctions plus complexes, et donc de réussir à mieux classifier les images. Cependant, nous avons observés le contraire en pratique car plus on ajoute des couches beaucoup plus denses plus les paramètres requièrent d’avantages une estimation. Ce qui ralenti la vitesse avec laquelle les poids sont mises à jour entre différentes couches. Néanmoins, ces différentes architectures nous ont également permis d’observes les paramètres à modifier dans un réseau de neurone convolutif pour avoir une meilleure précision.Pour améliorer nos résultats nous avons utilisé la fonction **Dropout** pour éliminer l’overfitting et pour aller plus loin on peut aussi changé la fonction d’activité, et aussi changer l’optimiseur pour l’apprentissage (Certains optimiseurs peuvent s’avérer être bien plus efficaces : *AdamOptimizer*, *AdagradOptimizer*, etc.). et on peut aussi Tester d’autres valeurs pour les tailles des couches cachées.