

8 Apprentissage supervisé et classification

Dans ce chapitre, nous présentons les différentes notions utiles à la mise en place d'un processus d'apprentissage supervisé pour le problème de la classification. Le problème de classification d'images consiste à deviner la catégorie d'une image parmi une liste de catégories connues (chat ou chien par exemple). Pour cela, on dispose d'une base d'images (un set d'apprentissage) fournissant des milliers d'échantillons utilisés dans l'apprentissage. La base fournit aussi pour chaque échantillon la réponse exacte devant être prédite par le réseau, ainsi on parle d'apprentissage supervisé.

8.1 Base d'entraînement et base de validation

8.1.1 Le surapprentissage

Une des bêtes noires du machine learning est sans aucun doute le surapprentissage (en anglais overfitting). Voici le scénario associé : vous avez entraîné un réseau qui obtient un excellent taux de bonnes prédictions, par exemple 99%, c'est un miracle ! Mais en production, lorsque vous utilisez le réseau préentraîné sur de nouvelles données, les performances sont très médiocres. Cette situation peut être synonyme de surapprentissage. En effet, si trop de neurones sont présents dans le réseau, il peut apparaître un phénomène d'apprentissage par cœur des données d'entraînement : le réseau arrive à identifier chaque échantillon et à fournir la réponse associée. Il n'y a donc pas apprentissage au sens strict où le système aurait recherché des caractéristiques particulières dans les données d'entrée permettant de fournir une prédiction de qualité.

Pour prévenir le phénomène de surapprentissage, on utilise un deuxième ensemble d'échantillons utilisé uniquement durant une deuxième phase de test/validation. On n'utilisera jamais les échantillons du groupe de test/validation durant la phase d'apprentissage. Ainsi, une fois la phase d'apprentissage terminée, les performances du réseau sont évaluées à partir d'échantillons lui étant inconnus. Cette approche est très fiable et permet de détecter le phénomène de surapprentissage.

Suivant la base que vous utilisez, les deux sets peuvent directement être fournis : le set d'entraînement (train) et le set de validation (test). Les deux sets sont organisés exactement de la même manière, le set d'entraînement est généralement 5 fois plus grand que le set de validation. Vous pourrez trouver d'autres bases fournissant un seul set de données. Ce sera donc à vous de le diviser pour construire un set d'entraînement et un set de validation avec une répartition 80% 20% par exemple.

8.1.2 Chargement des sets d'exemples de PyTorch

Vous pouvez trouver la liste des bases d'apprentissage disponibles depuis la librairie PyTorch à la page :

<https://pytorch.org/vision/stable/datasets.html>

On peut trouver plus d'une trentaine de bases sur des sujets très variés. Les bases fournies par les librairies d'apprentissage sont très pratiques, car elles font gagner un temps précieux en nous évitant le décodage de données binaires stockées dans un obscur format. Voici, par exemple, la syntaxe utilisée pour charger les informations contenues dans le set d'entraînement MNIST :

```
TrainSet = datasets.MNIST('../data', train=True, download=True)
```

Examinons la documentation de datasets.MNIST :

MNIST

```
CLASS torchvision.datasets.MNIST(root: str, train: bool = True, transform:
Optional[Callable] = None, target_transform: Optional[Callable] = None, download:
bool = False) [SOURCE]
```

MNIST Dataset.

Parameters

- **root** (*string*) – Root directory of dataset where `MNIST/processed/training.pt` and `MNIST/processed/test.pt` exist.
- **train** (*bool, optional*) – If True, creates dataset from `training.pt`, otherwise from `test.pt`.
- **download** (*bool, optional*) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.

Voici la description des paramètres qui nous intéressent :

- Le premier paramètre (`root`) correspond au répertoire qui va accueillir les données téléchargées. Vous pouvez utiliser `./data/` pour indiquer la racine de votre espace de travail.
- Le booléen `train` permet d'indiquer si l'on désire télécharger le set de données d'entraînement ou de validation.
- Le booléen `download` sera généralement positionné à True pour télécharger les données depuis internet, ceci uniquement au premier lancement. Les données une fois téléchargées seront conservées pour les fois suivantes.

8.2 Préparation des données

8.2.1 Uniformisation des échantillons

De nombreux sets sont maintenant disponibles sur internet. Cependant, parfois les formats utilisés dans les bases sont très hétérogènes. Il faut donc examiner au cas par cas chaque set et éventuellement homogénéiser l'ensemble des données, car, rappelons-le, les inputs d'un réseau doivent toujours être de même taille. Voici une présentation des différents problèmes pouvant être rencontrés :

- Résolutions hétérogènes des données :
 - Images de résolutions différentes pouvant être en format portrait ou paysage.
 - Données texte représentant des phrases contenant un nombre variable de mot.
- Absence de données :
 - Lorsque l'on reçoit des infos depuis des capteurs distants (sismographes par exemple), parfois un capteur peut ne plus émettre d'informations pendant une heure. Comment traiter ces cas ? Faut-il faire des blocs d'1 heure et rejeter les blocs

contenant des pannes ? Faut-il remplir de zéro les zones sans informations et espérer que le réseau sache traiter ces imperfections ?

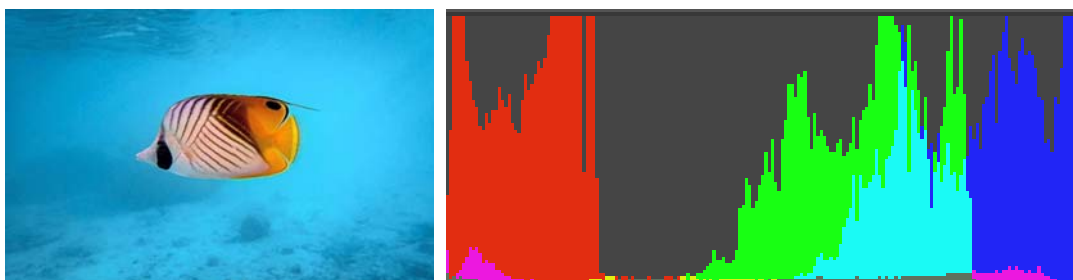
- Avec un flux vidéo qui transite sur internet, mais c'est aussi vrai pour une webcam, le framerate n'est pas fixe, il oscille entre 10 et 20 images par seconde par exemple. Par moment, il peut aussi chuter à 3 images par seconde. Le flux de données est donc irrégulier.
- Format des données :
 - Pixels d'une image stockés en x/y ou y/x
 - Codage des couleurs en RVB, BVR ou YCbCr
 - Images/Audio en 16 bits ou 8 bits
 - Enregistrement de sons en 44kHz ou 22kHz
- Echantillons triés par catégories (d'abord les tigres ensuite les avions...) ou échantillons classés aléatoirement ? Il serait préférable lors de l'apprentissage de présenter au réseau des images de différentes catégories.

L'intérêt des datasets fournis dans les bibliothèques d'apprentissage comme PyTorch est que les données sont déjà uniformisées. Cela permet d'économiser une semaine de travail ou plus. A titre d'exemple, dans la base d'images CIFAR10, quelle que soit l'orientation (portrait, paysage) et la résolution des images d'origine, les images ont toutes été redimensionnées vers une résolution de 32x32 pixels.

Remarque : une taille de 32x32 peut choquer, ne serait-ce que par la petitesse de la résolution. Mais plus la taille des images en entrée du réseau est grande, plus de couches de traitement seront nécessaires ce qui allongera forcément la durée d'apprentissage sans pour autant améliorer la performance.

8.2.2 Normalisation des entrées

Si nous prenons l'exemple des images RGB, trois plans de couleur sont disponibles. Si nous traitons des images de la faune marine par exemple, on obtient un histogramme comme ci-dessous :



Dans l'histogramme, les valeurs faibles sont comptabilisées sur la gauche et les niveaux hauts sur la droite. Ainsi, dans cette image les rouges sont très présents dans la zone des 0 à 20% et ils sont donc peu présents dans l'image. Les bleus sont évidemment présents dans les hautes valeurs entre 70% et 100%. La présence de vert dans les hautes valeurs peut surprendre, mais il faut remarquer que l'eau est plutôt d'un bleu céleste (38, 196, 236) ce qui correspond à l'histogramme que l'on retrouve ici.

Alors quel est le problème docteur ? Que les valeurs des rouges soient basses et que celles des bleues soient hautes peut être corrigé par le réseau, il suffit pour cela que les poids associés aux valeurs du rouge soient plus importants afin de rivaliser avec les valeurs des bleus déjà hautes. Effectivement, cela est possible, mais au démarrage de la phase d'apprentissage, les poids des

couches sont initialisés aléatoirement dans l'intervalle [0,1]. Ainsi, pour que les poids associés au rouge deviennent, en moyenne, 4 à 5 fois plus importants pour compenser le faible niveau du rouge, cela va nécessiter du temps et ce temps s'ajoute au temps global d'apprentissage.

Ainsi, il est d'usage de normaliser les plans R,V,B de toutes les images pour obtenir une distribution statistique ayant une moyenne de 0 et une variance de 1. Attention, cette normalisation s'effectue pour chaque canal en tenant compte de la totalité des images. Attention, la normalisation ne s'effectue pas pour chaque image. Ainsi, la normalisation du set d'images équivaut au code suivant :

```
# IMG : tenseur des images (60 000, 3, 32, 32) -> 60 000 images 32x32 en RVB
for c in range(3) :                # pour chaque canal R/V/B
    mean = IMG[:,c,:,:].mean()      # calcul de la moyenne
    std = IMG[:,c,:,:].std()         # calcul de la déviation standard
    IMG[:,c,:,:] = (IMG[:,c,:,:] - mean) / std    # normalisation
```

Cette normalisation doit être effectuée pour tous les datasets y compris les datasets fournies par PyTorch. Cependant, comme les bases sont de plus en plus grandes (ImageNet contient plusieurs millions d'images), l'évaluation de 3 moyennes et de 3 déviations standards peut prendre un certain temps. Ainsi, ces valeurs sont souvent écrites directement dans le code de chargement de la base à travers un exemple.

8.2.3 Construire une liste de transformations

Nous utilisons `transforms.Compose(...)` pour construire une liste de traitements à appliquer sur les données en entrée. Ensuite, le paramètre *transform* présent dans les constructeurs des datasets de PyTorch permet de passer cette série de transformations :

```
from torchvision import transforms
moy, dev = 0.1307, 0.3081
TRS = transforms.Compose([transforms.ToTensor(), transforms.Normalize(moy,dev)])
TrainSet = datasets.MNIST('./data', train=True, download=True, transform=TRS)
```

Les opérations de transformation sont appliquées dans l'ordre de la liste. La première opération consiste à convertir le tenseur des images utilisant des uint8 vers un tenseur en Float32. De plus, PyTorch effectue un scaling pour ramener les valeurs dans la plage [0,1]. Voici un extrait de la documentation en ligne ci-dessous :

CLASS `torchvision.transforms.ToTensor` [SOURCE]

Convert a `PIL Image` or `numpy.ndarray` to tensor. This transform does not support torchscript.

Converts a `PIL Image` or `numpy.ndarray` ($H \times W \times C$) in the range $[0, 255]$ to a `torch.FloatTensor` of shape $(C \times H \times W)$ in the range $[0.0, 1.0]$ if the `PIL Image` belongs to one of the modes (L, LA, P, I, F, RGB, YCbCr, RGBA, CMYK, 1) or if the `numpy.ndarray` has `dtype = np.uint8`

In the other cases, tensors are returned without scaling.

• NOTE

Because the input image is scaled to $[0.0, 1.0]$, this transformation should not be used when transforming target image masks. See the [references](#) for implementing the transforms for image masks.

La deuxième opération consiste à normaliser la distribution statistique des données. Pour les images en niveau de gris, nous donnons une seule valeur moyenne et une seule déviation standard. Voici un extrait de la documentation en ligne ci-dessous :

CLASS `torchvision.transforms.Normalize(mean, std, inplace=False)` [SOURCE]

Normalize a tensor image with mean and standard deviation. This transform does not support `PIL Image`. Given mean: `(mean[1], ..., mean[n])` and std: `(std[1], ..., std[n])` for `n` channels, this transform will normalize each channel of the input `torch.*Tensor` i.e., `output[channel] = (input[channel] - mean[channel]) / std[channel]`

• NOTE

This transform acts out of place, i.e., it does not mutate the input tensor.

Parameters

- **mean** (*sequence*) – Sequence of means for each channel.
- **std** (*sequence*) – Sequence of standard deviations for each channel.

Dans la documentation PyTorch, vous pouvez trouver l'ensemble des transformations disponibles :

<https://pytorch.org/vision/stable/transforms.html>

Remarque : cette série de transformation doit être effectuée en double pour construire le set d'images de validation.

8.3 La gestion du flux de traitement

8.3.1 Notions de batch, d'epoch et d'itérations

Le calcul de l'erreur totale sur l'ensemble des images peut être une opération très longue lorsque la base contient plusieurs centaines de milliers d'images. L'usage a montré qu'il était suffisant et plus efficace pour l'apprentissage de prendre un nombre limité d'images et de calculer un gradient

approché à chaque itération du pas du gradient. Ce lot d'images contenant un nombre constant d'images s'appelle en anglais un **batch**. Le set d'images est ainsi découpé en lots. Lorsque on a parcouru une fois la totalité des images, on dit que l'on a traité une **epoch**. Le nombre d'itérations utilisées durant une epoch est donné par la formule suivante :

$$\text{Nombre d'itérations dans une epoch} = \frac{\text{Nombres d'images dans la base}}{\text{Nombre d'images dans le batch}}$$

Une epoch est toujours constituée du même nombre d'itérations. Si nous avons un set de 50 000 images et une taille de lot de 100 images, il faudra donc 500 itérations pour compléter une epoch. La taille idéale du lot d'images n'est pas connue d'avance. Elle doit être déterminée empiriquement par essais successifs.

8.3.2 Traitement par lots

Une fois les données chargées, PyTorch dispose d'un outil permettant de construire un itérateur sur les lots d'images, il s'agit du DataLoader qui reçoit en paramètres la base à traiter ainsi que la taille des lots. L'itérateur met ainsi à disposition un lot d'images et les catégories associées à travers l'utilisation d'une boucle for :

```
loader = torch.utils.data.DataLoader(TrainSet, batch_size)
for (data, target) in loader:
    ...
```

8.3.3 Dans la pratique

Si nous examinons l'objet retourné par le dataset MNIST(...), on trouve dans ses attributs : la liste des transformations choisies et les données originales du dataset :

```
TrainSet = datasets.MNIST('../data', train=True, download=True, transform=TRS)
print(TrainSet)
>> Dataset MNIST
      Number of datapoints: 60000
      Root location: ../data
      Split: Train
      StandardTransform
Transform:
  Compose(
    ToTensor()
    Normalize(mean=0, std=0.5)
  )
Print(TrainSet.dataset.data[0])
>> tensor([[ 0,  0,  0,  0... , dtype=torch.uint8])
```

Les données des images sont stockées sous la forme d'uint8. Il semble que PyTorch n'est pas encore appliqué les transformations pour construire un tenseur contenant les images étalonnées en Float32. Ce choix technique peut se comprendre car les images stockées en 8 bits tiennent 4 fois moins de

place qu'en Float32. Ainsi, ce sera lors du traitement d'un nouveau lot, que les images retenues vont être converties. Cela permet d'économiser la mémoire des GPU (entre 10 et 20 Go), mémoire servant à la fois pour stocker les poids du réseau et les images de la base.

En examinant le tenseur associé au lot courant, nous obtenons les informations suivantes :

```
loader = torch.utils.data.DataLoader(TrainSet, batch_size = 100)
for (data, target) in loader:
    print(data.shape)
    print(data.dtype)
    print(data.mean())

>> torch.Size([100, 1, 28, 28])
>> torch.float32
>> tensor(-0.0242)
```

La taille du tenseur courant correspond à un lot de 100 images de résolution 28x28 pixels en niveaux de gris. En demandant la moyenne des images du lot courant avec la ligne `data.mean()`, nous constatons que cette valeur n'est pas nulle alors que la base d'images a été normalisée. Pourquoi ? Nous rappelons que la normalisation a lieu sur l'ensemble des images pour que la valeur moyenne des images soient nulles. Or, dans le lot courant, la valeur moyenne est calculée sur les 100 images présentes dans ce lot et non la totalité, ce qui explique que la moyenne affichée ne soit pas égale à 0.


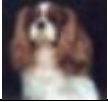

8.4 La classification d'images

8.4.1 Fonction d'erreur

Le problème de classification d'images consiste à associer une image à une catégorie parmi une liste de catégories connues. Parmi les bases les plus classiques, on trouve MNIST avec 10 catégories correspondant aux chiffres de 0 à 9, on trouve aussi la base CIFAR10 avec 10 catégories d'animaux : avion, voiture, oiseau, chat, daim, chien, grenouille, cheval, navire et camion. Un réseau de neurones devant classer ces images construit 10 valeurs notées S_0 à S_9 correspondant à un score associé à chaque catégorie, le score le plus haut indiquant la catégorie prédite. La base donne pour chaque image le numéro, noté k , de sa catégorie réelle. Ensuite, on peut utiliser la fonction d'erreur vue précédemment :

$$Erreur = \left[\sum_{i=0}^9 \max(0, (\Delta + S_i) - S_k) \right] - \Delta$$

La soustraction par un terme Δ permet de corriger la valeur obtenue avec $(\Delta + S_i) - S_k$ lorsque $i=k$. Ainsi, lorsque le score de la bonne catégorie dépasse de Δ les autres scores, la prédiction est considérée comme parfaite et l'erreur associée est nulle. Voici un exemple avec 3 images et 3 catégories. Le réseau en traitant ce lot de trois images génère un tenseur de scores 3x3. Dans le tableau ci-dessous, les scores associés aux bonnes catégories ont été soulignés. On peut ainsi calculer l'erreur introduite par chaque image en prenant $\Delta = 1$ et il nous suffit de sommer chaque erreur pour obtenir l'erreur totale.

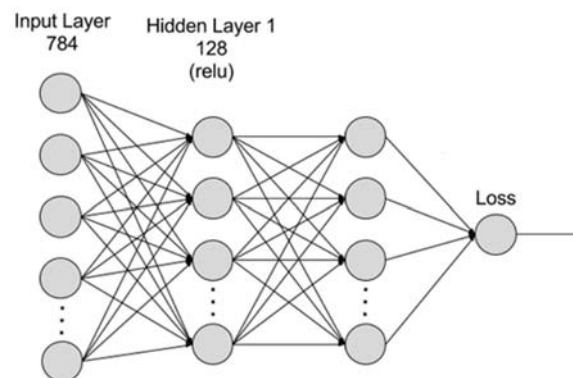
	Chien	Oiseau	Cheval		Erreur ($\Delta = 1$)	ErrTot
	5	4	<u>2</u>	→	$4+3+1-1 = 7$	15
	<u>4</u>	2	8	→	$1+0+5-1 = 5$	
	4	<u>5</u>	1	→	$0+1+0-1=0$	

8.5 Exercice 8 : classification sur MNIST

Ouvrez le fichier « Classification.py ». Les réponses aux questions se feront dans le code source, en début du document. Examinez la structure du code et répondez aux questions suivantes :

- Qu1 : quel est le % de bonnes prédictions obtenu au lancement du programme, pourquoi ?
- Qu2 : quel est le % de bonnes prédictions obtenu avec 1 couche Linear ?
- Qu3 : pourquoi le test_loader n'est pas découpé en batch ?
- Qu4 : pourquoi la couche Linear comporte-t-elle 784 entrées ?
- Qu5 : pourquoi la couche Linear comporte-t-elle 10 sorties ?

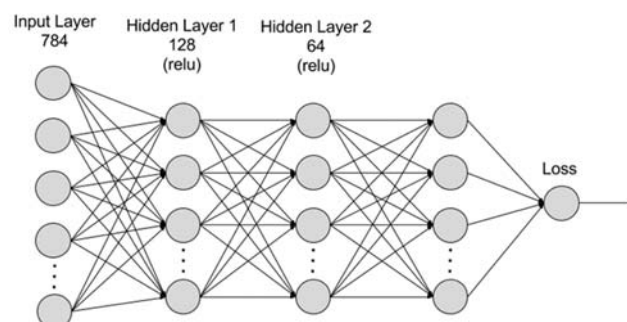
On se propose de faire évoluer le réseau actuel vers la topologie suivante :



Modifiez la classe Net pour que le réseau ait maintenant 2 couches Linear avec 128 neurones sur la première couche. La fonction d'activation sera ReLU.

- Question 6 : quelles sont les tailles des deux couches Linear ?
- Question 7 : quel est l'ordre de grandeur du nombre de poids utilisés dans ce réseau ?
- Question 8 : quel est le % de bonnes prédictions obtenu avec 2 couches Linear ?

Peut-on encore améliorer les performances en ajoutant une couche Linear supplémentaire ? La couche contenant le plus de poids est la couche gérant les entrées. Les couches proches de la sortie nécessitent beaucoup moins de poids, essayons d'ajouter une 3ème couche Linear.



Modifiez la classe Net pour que le réseau ait maintenant 3 couches Linear avec 128 neurones sur la première couche et 64 neurones sur la seconde. La fonction d'activation sera ReLU.

- Question 9 : obtient-on un réel gain sur la qualité des prédictions ?

8.6 Probabilités et entropie

8.6.1 La fonction Softmax

Nous allons maintenant utiliser une nouvelle fonction très célèbre dans le monde de l'apprentissage : la fonction Softmax. Elle permet à partir des valeurs de sortie d'un réseau fournissant 1 score par catégorie à prédire, de construire une probabilité d'appartenance pour chaque catégorie. Ainsi, la fonction Softmax transforme un tenseur de k valeurs réelles en un tenseur de k probabilités. La formule utilisée pour obtenir la i -ème valeur en sortie de la fonction Softmax pour un tenseur d'entrée $X=(x_j)_{0 \leq j < n}$ est donnée par :

$$\text{Softmax}(i, X) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Au niveau des calculs effectués, on peut découper cette formule en trois étapes comme dans le tableau ci-dessous :

	<i>Input</i>	<i>exp(x_i)</i>	<i>Total</i>	<i>exp(x_i)/Total</i>
x_0	-1	0,368	23,172	1,59%
x_1	1	2,718		11,73%
x_2	3	20,086		86,68%

- En premier lieu, chaque valeur présente dans le vecteur d'entrée est injectée dans une fonction exponentielle pour obtenir une série de valeurs intermédiaires.
- En second lieu, l'ensemble des valeurs intermédiaires sont sommées pour obtenir un total.
- Finalement, la série des valeurs intermédiaires est normalisée en divisant chaque valeur intermédiaire par le total. On obtient une série de probabilités.

On peut remarquer que les valeurs en sortie de la fonction Softmax sont positives et que leur somme est égale à 1. Ainsi la fonction Softmax à partir d'un tenseur d'entrée quelconque construit un tenseur de sortie correspondant à une distribution de probabilités.

La fonction Softmax présente certains avantages :

- Qu'elle que soit le signe des valeurs en entrée, l'exponentiation va produire des valeurs positives. On n'a donc pas de contraintes à imposer sur le signe des valeurs produites par les couches précédentes du réseau.
- Qu'elle que soit l'ordre de grandeur des valeurs d'entrée, la fonction Softmax construit une distribution de probabilités. Ainsi, que les valeurs d'entrée aient un ordre de grandeur autour de 1/10 ou de 1000 000, la fonction Softmax produit le résultat escompté.

Voici ce qu'indique la documentation PyTorch pour la fonction Softmax :

TORCH.NN.FUNCTIONAL.SOFTMAX

```
torch.nn.functional.softmax(input, dim=None, _stacklevel=3, dtype=None) [SOURCE]
```

Applies a softmax function.

Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

It is applied to all slices along dim, and will re-scale them so that the elements lie in the range $[0, 1]$ and sum to 1.

La fonction Softmax associe un ensemble de valeurs des probabilités. La fonction exponentielle étant croissante, cela sous-entend que plus une valeur d'entrée est grande plus sa probabilité augmente. Ainsi, lorsqu'une valeur d'entrée augmente par rapport aux autres, alors, sa probabilité augmente et par rééquilibrage les autres probabilités se voient diminuer. Prenons un exemple sur un cas simple :

Input	exp(...)	Sigma	prob
-1	0,368	23,172	1,59%
1	2,718	23,172	11,73%
3	20,086	23,172	86,68%

Si la deuxième valeur en entrée passe à 2, nous obtenons une nouvelle série de probabilités :

input	exp(...)	Sigma	prob
-1	0,368	27,842	1,32%
2	7,389	27,842	26,54%
3	20,086	27,842	72,14%

En passant de 1 à 2, la probabilité augmente de 11,75% à 26,54%. Les deux autres probabilités dont les valeurs d'entrée sont inchangées diminuent, il y a eu une forme de rééquilibrage.

Lorsque l'on utilise la fonction Softmax, on peut voir les valeurs d'entrée comme des intensités. Plus l'intensité est importante, plus la probabilité associée est importante. Ces valeurs n'ont pas d'unités, pas de plage de valeurs limitée, elles peuvent donc prendre une valeur quelconque mais dans tous les cas la fonction Softmax s'adapte et construit intelligemment une liste de probabilités cohérentes.

Propriété : les probabilités issues de la fonction Softmax sont invariantes par ajout d'une constante.

Cela propriété vient de la définition de la fonction Softmax :

Posons : $x'_i = x_i + c$ pour tout i , nous avons :

$$\text{Softmax}(i, X') = \frac{\exp(x_i + c)}{\sum_j \exp(x_j + c)} = \frac{\exp(x_i) \cdot \exp(c)}{\sum_j \exp(x_j) \cdot \exp(c)} = \text{Softmax}(i, X)$$

Pour tester cette propriété, décalons de 10 les valeurs de l'exemple précédent. Nous remarquons alors que les probabilités obtenues restent inchangées :

input	exp(...)	Sigma	prob
9	8103,084	510390,618	1,59%
11	59874,142	510390,618	11,73%
13	442413,392	510390,618	86,68%

Ou encore :

input	exp(...)	Sigma	prob
-11	1,6702E-05	1,0520E-03	1,59%
-9	1,2341E-04	1,0520E-03	11,73%
-7	9,1188E-04	1,0520E-03	86,68%

Peut-être pensez qu'une formule plus simple comme : $p(x_i) = x_i / \sum x_j$ auraient suffi, la somme des valeurs construites valant bien 1. Comparons cependant ces deux approches :

- Avec cette formule simplifiée, les valeurs d'entrées doivent être positives sinon on pourrait obtenir des probabilités négatives ce qui est impossible. Cette contrainte est difficilement satisfaisable à la sortie d'un réseau. En comparaison, la fonction Softmax s'adapte à tout type de valeurs, positives comme négatives et produit toujours des résultats positifs.
- Avec la formule simplifiée, il faut être sûr que les valeurs d'entrée soient non nulles sinon on obtient une division par zéro. Avec des entrées nulles, la fonction Softmax construit des probabilités équiprobables car les entrées sont équivalentes.
- Avec la formule simplifiée, les valeurs (1,1,2) donnent les probabilités en sortie de 25%, 25% et 50% alors qu'avec (10,10,11) les résultats sont proches de l'équiprobabilité. L'ordre de grandeur des valeurs d'entrée influence donc beaucoup le comportement de la fonction. Ainsi, la formule simplifiée produit des probabilités très différentes lorsque les valeurs se décalent. Il n'en ait rien pour la fonction Softmax qui corrige ce défaut. Ainsi pour la fonction Softmax seul compte l'écart entre la plus grande valeur d'entrée et les autres.

Si la formule simplifiée semble plus facile à mettre en œuvre au premier abord, son utilisation s'avère bien plus contraignante.

8.6.2 La notion d'entropie

Nous introduisons une notion importante pour la suite : l'entropie. L'entropie est une valeur caractérisant le niveau de désorganisation ou d'imprédictibilité d'un système. Pour une distribution de probabilité discrète $X(\Omega) = (x_i)_{0 \leq i < n}$ associée aux probabilités $(p_i)_{0 \leq i < n}$, l'entropie est définie ainsi :

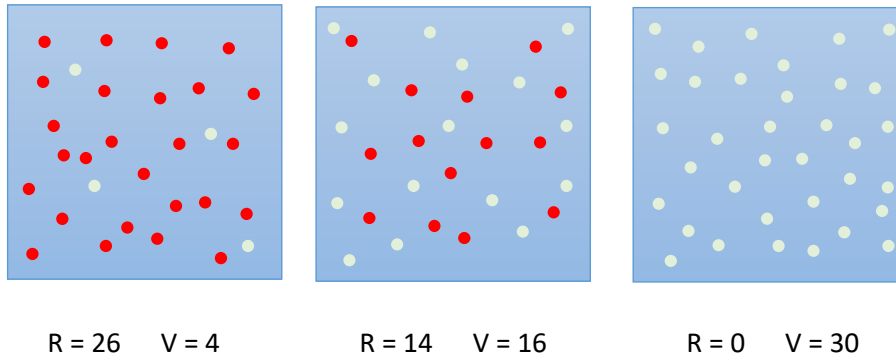
$$H(X) = - \sum_{p_i > 0} p_i \cdot \log_2(p_i)$$

Nous avons : $H(X) \geq 0$ car les probabilités étant des valeurs dans $[0,1]$, les termes en log sont tous négatifs, ainsi le signe – avant le grand sigma donne un résultat positif.

L'entropie $H(X)$ n'est pas une valeur bornée. En effet, si nous avons n valeurs équidistribuées, nous obtenons :

$$H(X) = - \sum_n \frac{1}{n} \cdot \log_2 \left(\frac{1}{n} \right) = \log_2(n)$$

Examinons un cas pratique de trois boîtes contenant des billes rouges et des billes vertes :



Dans la boîte de gauche, la probabilité de tirer une boule rouge est de 26/30 et celle de tirer une bille verte est de 4/30. Ainsi, la probabilité de tirer une boule rouge est plus certaine.

Dans la boîte au centre, la probabilité de tirer une boule rouge est de 14/30 et celle de tirer une bille verte de 16/30. Il y a presque une chance sur deux de tirer une boule rouge et une boule verte. Il devient difficile d'être sûr de la couleur tirée.

Dans la boîte de droite, nous sommes certains que la boule tirée sera de couleur verte. La probabilité de tirer une boule verte est de 1 contre 0 pour une boule rouge.

Calculons les valeurs de l'entropie pour chacune des situations :

- Boîte de gauche : $-26/30 \cdot \log_2(26/30) - 4/30 \cdot \log_2(4/30) = 0.3926$
- Boîte au centre : $-14/30 \cdot \log_2(14/30) - 16/30 \cdot \log_2(16/30) = 0.6909 < \log(2)=0.6931$
- Boîte de droite : $-30/30 \cdot \log_2(30/30) = 0$

L'entropie de la boîte de gauche est inférieure à la boîte du centre car en tirant une boule dans cette boîte nous sommes presque sûr qu'elle sera rouge. Dans la boîte au centre, nous ne sommes sûr de rien, elle peut être rouge comme verte et l'entropie calculée est proche de l'entropie maximale possible : $\log(2)$. La boîte de droite correspond à une situation idéale où nous sommes absolument certains que toute boule tirée sera verte, en conséquence l'entropie est nulle.

8.6.3 Exprimer l'erreur grâce à l'entropie

Nous allons utiliser la définition de l'entropie croisée pour construire une fonction d'erreur. Dans la littérature anglo-saxonne on trouve plusieurs terminologies : **Cross-Entropy Loss, Logarithmic Loss ou Logistic Loss**. L'entropie croisée pour deux distributions discrètes p et q est définie de la façon suivante :

$$CrossEntropyLoss(p, q) = - \sum_{p_i > 0} p_i \cdot \log_2(q_i)$$

Quand on compare une distribution de probabilités q avec une distribution de probabilités p , l'entropie croisée atteint son minimum lorsque les deux distributions sont égales i.e. $p_i = q_i$. Ce minimum correspond à la valeur $H(p) = H(q)$. Supposons qu'une distribution soit connue et que nous cherchions à faire en sorte qu'une deuxième distribution apprenne cette distribution de « référence » afin d'être la plus proche possible. Alors nous pouvons formuler ce problème comme un problème de minimisation de l'entropie croisée.

Dans le problème de la classification, si nous avons trois catégories, lorsque nous savons que la réponse doit correspondre à la première catégorie, cela équivaut à dire que nous aimerions que les probabilités obtenues par la fonction Softmax correspondent à la distribution (1,0,0). Ainsi, si l'on nomme $icat$ le numéro de la catégorie de l'échantillon courant, l'expression de l'entropie croisée précédente se réécrit de manière équivalente en :

$$CrossEntropyLoss(p, icat) = -\log_2(p_{icat})$$

De l'expression précédente, il ne reste qu'un seul terme correspondant à la probabilité non nulle. Ainsi en minimisant la fonction CrossEntropyLoss, on force les poids du réseau à s'optimiser pour que les probabilités en sortie de la fonction Softmax correspondent à une probabilité de 1 pour les catégories désirées.

Voici les informations données dans la documentation PyTorch :

CROSSENTROPYLOSS

```
CLASS torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=- 100,
    reduce=None, reduction='mean', label_smoothing=0.0) [SOURCE]
```

This criterion computes the cross entropy loss between input and target.

8.6.4 Instabilité numérique

Cette section est réservée aux curieux et elle peut être sautée pour ceux qui sont en retard.

A la question : pourquoi avons-nous choisi telle ou telle fonction / algo / paramètre ? Nous répondons habituellement par une de ces trois réponses :

- On utilise moins d'itérations : couche dropout, normalisation des inputs...
- On effectue autant d'itérations, mais les calculs sont plus économiques et consomment moins de ressources CPU/GPU : ReLU (plutôt que tanh)...
- Les prédictions sur l'ensemble de validation sont meilleures : couche Conv

Nous allons introduire un nouveau couple de fonctions : LogSoftmax et NegativeLogLikelihood Loss qui vont remplacer le couple précédent : Softmax et Logarithmic Loss. Cette modification ne s'est pas imposée cette fois par un gain de performance mais pour corriger des instabilités numériques. Mais où se situent les risques lorsque l'on évalue la fonction Softmax et Logarithmic Loss pourtant assez robustes en apparence ? Pour rappel, voici la définition de la fonction Softmax :

$$p_i = \text{Softmax}(i, X) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Le premier problème provient des exponentielles présentes dans la fonction. Car avec une exponentielle, les grandeurs associées peuvent grimper vite, très vite. Or, les calculs dans le réseau sont faits en Float32 pouvant au maximum représenter la valeur 3.402823E+38. Ainsi, il suffit qu'en entrée de la fonction Softmax une valeur soit supérieure à $\ln(3.402823E+38) \sim 88.72$ pour que le résultat de l'exponentielle soit égal à $+\infty$ et c'est l'accident. La suite des calculs sera forcément erronée. Voici un exemple :

x_i	$\exp(x_i)$	Float 32
80	5,540622E+34	5,540622E+34
89	4,489613E+38	$+\infty$
90	1,220403E+39	$+\infty$

Pour éviter le problème de saturation à l'infini, nous allons soustraire à l'ensemble des valeurs x_i le maximum des x_i noté c pour obtenir une série de nouvelles valeurs notées x'_i . Nous avons précédemment vu dans le chapitre sur la fonction Softmax que cela ne changerait pas sa valeur finale. Ainsi, nous évitons avec cette astuce toute forme de saturation :

x_i	c	$x'_i = x_i - c$	$\exp(x_i)$	$\sum \exp(x_i)$	Prob
80	90	-10	4,54E-05	1,367925	0,003%
89		-1	0,367879	1,367925	26,893%
90		0	1	1,367925	73,103%

Une fois cette première astuce utilisée, nous allons rencontrer un autre problème pour les exposants avec une valeur négative. En effet, avec des nombres en Float32, la valeur minimale que l'on puisse représenter est -3.402823466 E+38. Ainsi, lorsqu'il existe une valeur x_i telle que $x_i < c-82$, nous allons obtenir un résultat nul :

x_i	c	$x'_i = x_i - c$	$\exp(x_i)$	$\sum \exp(x_i)$	Prob
2	90	-88	0	1,367879	0,000%
89		-1	0,367879	1,367879	26,894%
90		0	1	1,367879	73,106%

En soit, ce résultat n'est pas incohérent car la probabilité est quasi nulle. Cependant, nous injectons ce résultat dans la formule de la CrossEntropyLoss avec une fonction logarithmique et l'évaluation de $\log(0)$ produira une erreur.

Nous allons donc utiliser une deuxième astuce pour gérer ce cas. Pour cela, nous allons construire une formule effectuant le calcul du Softmax puis du Log dans la même expression :

$$\text{LogSoftmax}(i, X') = \log \left(\frac{\exp(x'_i)}{\sum_j \exp(x'_j)} \right)$$

En développant cette expression nous obtenons :

$$\text{LogSoftmax}(i, X') = x'_i - \log \left(\sum_j \exp(x'_j) \right)$$

Par définition des x'_j , une de ces valeurs est nulle et comme l'exponentielle est toujours positive et que $\exp(0)=1$, le résultat du grand sigma est supérieur ou égal à 1. Ainsi le logarithmique ne reçoit jamais une valeur nulle en paramètre et son évaluation n'est plus problématique.

On peut maintenant définir la NegativeLogLikelihood Loss qui correspond à la fonction Logarithmic Loss privée de son logarithme :

$$\text{Negative Log Likelihood Loss}(X, \text{icat}) = -x_{\text{icat}}$$

8.7 Suite de l'exercice 8

8.7.1 Les modules PyTorch

Sous PyTorch, les fonctions sont parfois disponibles à deux endroits différents :

- Dans `torch.nn.functional` : dans ce cas, il s'agit d'une fonction.
- Dans `torch.nn` : dans ce cas, il s'agit d'un objet foncteur, qu'il faut d'abord instancier et ensuite utiliser comme une fonction. L'objet foncteur permet de stocker des informations supplémentaires et de préserver des données dans le temps : états, poids ; ce qui est impossible avec une fonction.

Dans le cas où vous avez le choix, préférez la version présente dans `torch.nn.functional` pour éviter tout comportement inattendu. D'autres fois, les deux versions disponibles seront légèrement différentes : une version aura plus d'options ou sera plus souple... le choix se portera alors sur l'option la plus pratique... En résumé, rien d'évident sur ce point.

8.7.2 Implémentation

Nous allons reprendre le code de l'exercice de classification et implanter les deux nouvelles fonctions : `Softmax` et `CrossEntropy`. Pour cela, vous utiliserez :

- La fonction **Softmax** disponible dans `torch.nn.functional`
- L'objet **CrossEntropy** de `torch.nn`, (examinez l'exemple dans la documentation). Cette fonction est aussi disponible dans `torch.nn.functional` mais son usage est plus contraignant.

Si vous avez lu la section sur les instabilités numériques, vous pouvez utiliser :

- La fonction **log_softmax** disponible dans `torch.nn.functional`
- La fonction **nll_loss** disponible dans `torch.nn.functional`

Répondez à la question suivante :

- Question 10 : pourquoi ne change-t-on pas le code de la fonction `TestOK()` servant à déterminer le % de prédictions correctes sur le `TestSet` suite à cette modification ?

9 Les réseaux convolutionnels

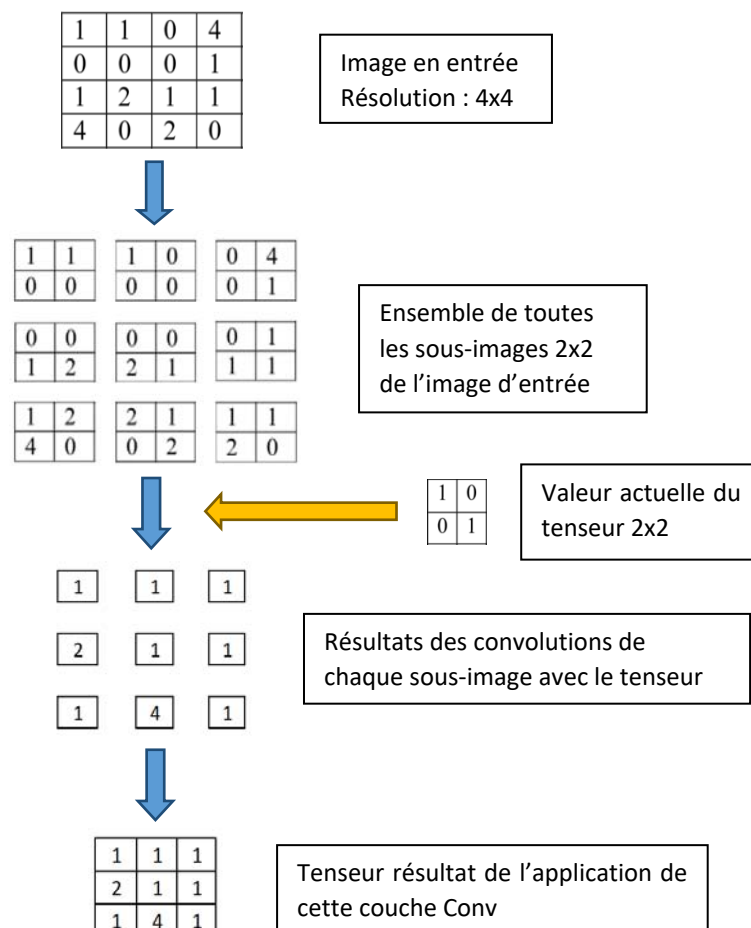
9.1 Les couches Conv et Dropout

9.1.1 La convolution 2D

Pour calculer la convolution entre un tenseur 2D de taille (u,v) et une image représentée par un tenseur de taille (n,m) , on parcourt chaque pixel de l'image et lorsqu'on peut extraire une sous-image de taille (u,v) depuis le pixel courant, on effectue la multiplication membre à membre entre cette portion d'image et le tenseur de convolution. L'ensemble des résultats obtenus forme un tenseur de taille $(n-u+1, m-v+1)$. Voici le code de principe correspondant :

```
def Conv2D(Tconv, ImageTensor) :
    sizeT      = Tconv.shape
    sizeImage  = ImageTensor.shape
    # tenseur résultat :
    R = torch.FloatTensor(size = (sizeImage[0]-sizeT[0]+1, sizeImage[1]-sizeT[1]+1))
    for x in range(0, R.shape[0]):
        for y in range(0, R.shape[1]):
            exImage = ExtractImage(ImageTensor,x,y,sizeT)
            R[x,y] = torch.mul(Tconv, exImage)    # multiplication membre à membre
```

Voici un cas pratique
utilisant une image 4x4
et un tenseur 2x2 :



Dans une image, les zones de transition sont souvent « douces ». Il n'est pas toujours nécessaire de calculer la convolution en chaque point de l'image, mais seulement tous les k pixels. Cela permet de plus de réduire la taille du tenseur de sortie d'un facteur k^2 . Le **pas** ou **stride** désigne ainsi le « saut » entre chaque convolution.

Les bords peuvent aussi poser problème. En effet, la taille du tenseur de sortie diminue légèrement en fonction de la taille du tenseur de convolution. Pour éviter cela, on peut élargir virtuellement l'image d'entrée avec des 0. Ce paramètre de **marge** ou **zero padding** permet ainsi de régler finement la taille du tenseur de sortie.

9.1.2 La convolution 2D multicanaux

Nous avons présenté la convolution dans le cas d'une image en niveaux de gris. Pour une image RVB, trois canaux sont présents. Notons $*$ l'opération de convolution 2D. Si on utilisait le même tenseur de convolution pour les trois canaux couleur, on obtiendrait :

$$T_{out} = \sum_{k=0}^2 Tconv * Image[k] = Tconv * \sum_{k=0}^2 Image[k]$$

Cette opération reviendrait à d'abord sommer les canaux couleurs $\sum_{k=0}^2 Image[k]$, ce qui équivaut à convertir l'image RVB en niveaux de gris, pour ensuite appliquer une convolution. Ainsi, l'information couleur est perdue, il faut donc choisir une autre approche.

Ainsi, lorsque l'on traite une image RVB avec trois canaux, on va utiliser 3 tenseurs pour la convolution ; chacun étant dédié à une couche couleur. Les trois tenseurs doivent être exactement de même dimension. La mise en place de cette technique se traduit par la formule :

$$T_{out} = \sum_{k=0}^2 Tconv[k] * Image[k]$$

La notation $Image[k]$ désigne la k -ième couche du tenseur image de taille (k, L, H) . Les k tenseurs de taille (u, v) utilisés pour la convolution par canal sont simplement regroupés dans un même tenseur de taille (k, u, v) ; l'écriture $Tconv[k]$ désigne ainsi le k -ième tenseur utilisé pour la k -ième couche.

9.1.3 La couche Conv2D

Nous présentons maintenant les couches convolutionnelles, en abrégé Conv, ayant permis aux réseaux de neurones d'atteindre un tout autre niveau de performance. Une couche Conv gère un groupe de **caractéristiques (feature en anglais, ou encore kernel)** correspondant à des tenseurs 2D de même taille. Le nombre de features s'appelle **la profondeur de la couche Conv**. La sortie d'une couche Conv correspond à un tenseur contenant les différentes convolution 2D entre chaque feature et l'image d'entrée. A noter que l'on parle de **CNN, Convolutional Neural Network**, lorsqu'au moins une couche Conv est présente dans le réseau. Un réseau à base de couches Linear ne permet pas d'obtenir un CNN !

La couche Conv participe aux calculs Forward du réseau et son résultat contribue à l'erreur finale. Ainsi les valeurs contenues dans ses features se comportent comme des poids qui vont ainsi être optimisés par la méthode du gradient. Le pas, la marge et la profondeur de la couche CONV sont des hyperparamètres d'apprentissage du réseau : ils sont choisis par l'utilisateur en début d'apprentissage et n'évoluent pas durant l'apprentissage.

Il reste à gérer le nombre de channels en entrée et en sortie. Examinons les informations données par la documentation en ligne à l'adresse :

<https://pytorch.org/docs/stable/nn.html#convolution-layers>

CONV2D

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
    dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None) [SOURCE]
```

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{out}, H_{out}, W_{out})$ can be precisely described as:

$$\text{out}(N_i, C_{out_j}) = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) \star \text{input}(N_i, k)$$

La formule n'est pas évidente à saisir à la première lecture. Si l'on note la convolution multicanale par le symbole \otimes , on peut la réécrire en utilisant le pseudo-code suivant :

```
Images : tenseur [N,Cin,L,H]          # n images LxH avec Cin canaux
Features : tenseur [Cout,Cin,u,v]      # Cout features de taille (Cin,u,v)
Out : tenseur [N,Cout,L-u+1,H-v+1]    # résultats
for i in range(N) :                   # boucle sur les images
    for j in range(Cout) :             # boucle sur les features
        Out[i,j,:,:] = b[j] + Features[j]  $\otimes$  images[i]
```

A retenir :

- L'utilisation de n features produit un résultat ayant n canaux
- Si k canaux sont présents en entrée, les features (u,v) sont en fait de taille (k,u,v)

Voici un exemple :

```
RVB = torch.FloatTensor( size = ( 1000,3,28,28) )
C1 = torch.nn.Conv2d(in_channels=3,out_channels=32,(5,5))
R = C1(RVB)
print(R.shape)
print(C1.weight.shape)

>> torch.Size([1000, 32, 24, 24])
>> torch.Size([32, 3, 5, 5])
```

Dans cet exemple, la couche Conv2D gère 32 features de taille 5x5, cette couche prend en entrée un tenseur de 1000 images RVB 28x28. En sortie de cette couche, le tenseur résultat a une taille de [1000,32,24,24] correspondant aux 1000 images et 32 features. La taille 24x24 vient de la taille des images en entrée 28x28 réduite de 4 = 5-1, avec 5 correspond à la taille des features. Le tenseur des poids est de taille [32,3,5,5] correspondant aux 32 features de taille 5x5 traitant des images à 3 canaux RVB. Cette couche utilise $32 \times 3 \times 5 \times 5 + 32 = 2\,432$ poids. On peut remarquer que cela est beaucoup moins qu'une couche Linear placée en début de réseau.

9.1.4 Interprétation des features

Mais à quoi peuvent correspondre les features une fois appris ? Il s'agit ici d'une question difficile. Cependant, on a réussi à déterminer que pour la couche traitant l'image d'entrée, ces features correspondent à des détecteurs de contours. Voici un exemple de résultat pour la classification des chiffres manuscrits :



Ainsi les features cherchent à détecter, dans les images traitées, des « caractéristiques » permettant de décrire l'information présente dans l'image. Si une couche Conv a seulement 5 features, elle va devoir en choisir 5 qui représentent au mieux l'information recherchée dans l'image.

9.1.5 La Couche Pool

L'objectif d'une couche Pool est de réduire la quantité d'informations. Par exemple, pour un tenseur 2D, on peut choisir de le diviser en blocs de taille 2x2 et d'effectuer pour chaque bloc une opération du type : moyennage, min ou max... Le tenseur résultat aura donc une taille 4 fois moindre. Les couches Pool permettent de réduire le flux de données d'une couche à l'autre. Vous pouvez trouver plus d'informations dans la documentation PyTorch ici :

<https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html>

MAXPOOL2D

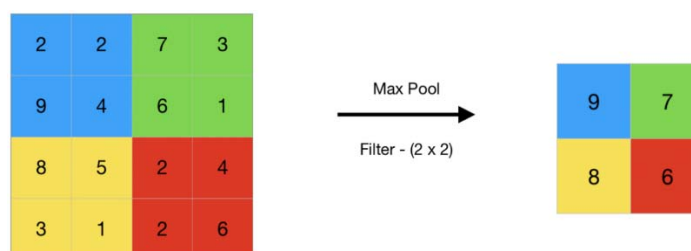
```
CLASS torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1,
    return_indices=False, ceil_mode=False) [SOURCE]
```

Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W) , output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0, \dots, kH-1} \max_{n=0, \dots, kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

Nous présentons un exemple de MaxPool-2D avec une réduction 2x2 :



9.1.6 La couche Dropout

Cette couche permet de mettre à zéro aléatoirement certaines valeurs d'un tenseur ceci avec une probabilité donnée. Il n'y a pas réduction de la taille des données, mais simplement une simulation d'une perte d'information. L'intérêt est d'améliorer l'indépendance entre les différentes features. En effet, supposons que le réseau doive trouver deux features A et B. Sans la technique de Dropout, il pourrait aussi bien sélectionner les features A et A+B, car les traitements sur les couches suivantes pourraient compenser ce choix pour reconstruire les features A et B idéaux. En désactivant de temps à autre la feature A ou la feature B, cela force le feature restant à converger sans tenir compte des valeurs du feature voisin.

Les couches Dropout ne sont utiles que durant la phase d'apprentissage et non durant la phase de validation. Ainsi, il faudra écrire :

- Au début du traitement du lot : `MonReseau.train()` pour activer les couches Dropout
- Au début de la phase de validation : `MonReseau.eval()` pour désactiver les couches Dropout

Les choses ne se résument pas seulement à une activation/désactivation. En effet, si l'on choisit une probabilité p d'effacer une donnée, cela veut dire que la somme des valeurs en sortie est abaissée (en moyenne) d'un facteur $1-p$ par rapport à l'entrée. Pour compenser cette perte, un facteur $1/(1-p)$ est appliquée aux valeurs conservées pour préserver le niveau moyen des valeurs d'entrée.

Voici un extrait des informations que l'on peut trouver dans la documentation de PyTorch à l'adresse :

<https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html>

DROPOUT

CLASS `torch.nn.Dropout(p=0.5, inplace=False)` [SOURCE]

During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper [Improving neural networks by preventing co-adaptation of feature detectors](#).

Furthermore, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training. This means that during evaluation the module simply computes an identity function.

Parameters

- **p** – probability of an element to be zeroed. Default: 0.5
- **inplace** – If set to `True`, will do this operation in-place. Default: `False`

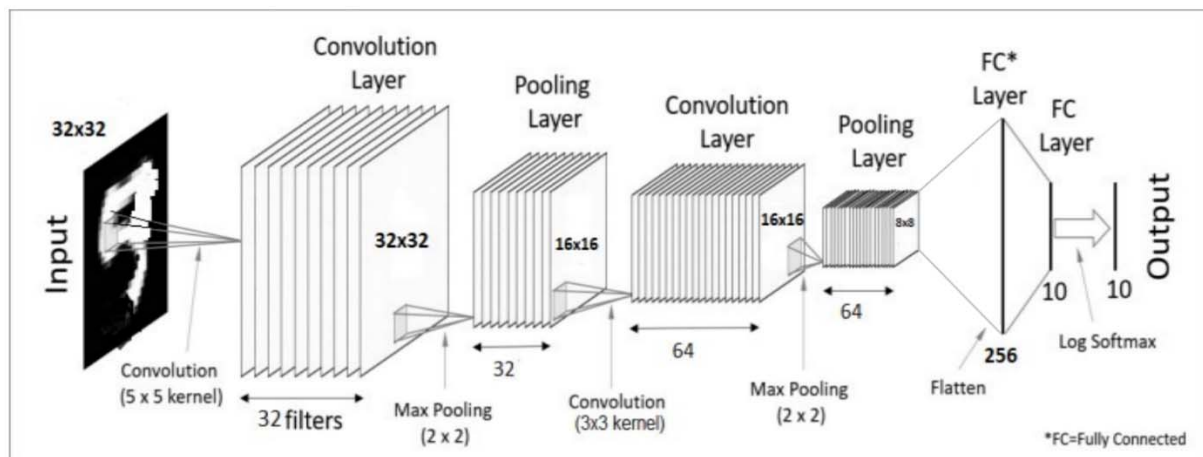
Voici un exemple de l'application d'une couche Dropout avec une probabilité de 0.5 sur un tenseur rempli de valeurs 1. Vous remarquerez que sur les 10 valeurs présentes 6 ont été mises aléatoirement à zéro. Les valeurs de sortie sont multipliées par $2 = 1/p$. Si vous lancez plusieurs fois ce test, vous obtiendrez des résultats différents.

```
import torch
m = torch.nn.Dropout(p=0.5)
input = torch.ones(10)
output = m(input)
print(input)
print(output)
```

```
>> tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
>> tensor([0., 0., 0., 0., 2., 2., 0., 0., 2., 2.])
```

9.1.7 Lecture des illustrations

Les articles sur les réseaux de neurones ont fait apparaître un art graphique dédié à la représentation des différents réseaux ! Les styles peuvent varier, quoique les empilements de rectangles ou de cube soit assez présents mais la tendance générale reste au style épuré. Ainsi, pour décrypter ces diagrammes, il faut connaître les spécificités de chaque couche pour déduire les informations implicites. Voici un exemple :



Analysons ce réseau de gauche à droite :

- En entrée, on trouve une image 2D de résolution 32x32.
- La sortie de la première couche est le résultat d'une convolution 2D donnant un tenseur de taille (32,32,32). Il y a donc 32 features dans cette couche Conv. Les features étant de taille 5x5, la résolution de sortie aurait dû être de 28x28 et non 32x32, un paramètre de zéro padding a dû être utilisé pour compenser la perte de résolution.
- La couche de Max-Pooling réduit la résolution par 2, ce qui donne un tenseur (32,16,16). Il y a toujours 32 canaux car l'opération de MaxPooling2D ne diminue pas le nombre de canal.
- Une nouvelle couche Conv de 64 features est appliquée. L'entrée étant de (32,16,16), la sortie 64x16x16 et le kernel de 3x3, la couche Conv2D a été construite avec les paramètres (32,64,(3,3)). Cette couche traite donc le tenseur d'entrée comme une image 16x16 avec 32 canaux.
- La couche de Max-Pooling suivante réduit la résolution par 2, ce qui donne un tenseur (64,8,8) en sortie. Le nombre de canaux reste inchangé.
- Le tenseur [64,8,8] subit un reshape pour être mis sous la forme d'un tenseur 1D de taille $64 \times 8 \times 8 = 4096$, cette opération est indiquée par la flèche flatten (aplatissage).
- On trouve ensuite une couche Linear (aussi appelée Fully Connected). La grandeur 256 indique sûrement le nombre de neurones présents dans cette couche. La fonction d'activation n'est pas précisée, on peut supposer qu'il s'agit d'une fonction ReLU.
- La dernière couche correspond à une deuxième couche Linear. Elle contient 10 neurones qui calculent 10 scores correspondant aux 10 catégories étudiées.
- La fonction d'erreur indiquée par Log Softmax correspond au critère Softmax et entropie.

9.2 Un peu d'histoire

9.2.1 La quête de performance

Dans le domaine de la classification d'images, une course à la performance s'est instaurée pour atteindre les taux d'erreur les plus faibles sur les bases existantes.

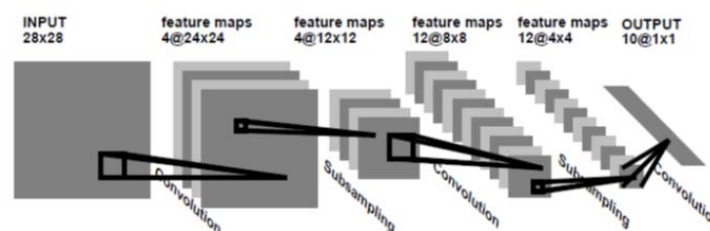
En ce qui concerne le problème de la classification des chiffres manuscrits, plusieurs algorithmes ont été mis en place pour passer de 98% de précision à 99.85% aujourd'hui, gagnant chaque fois quelques dixièmes de pourcent. Cela peut sembler superflu mais il n'en est rien. En effet, l'application historique de la reconnaissance de chiffres manuscrits était la lecture de codes zip sur les enveloppes postales. Si l'on part du principe qu'une machine de tri traite 100 000 enveloppes par jour et qu'il y a 5 chiffres à analyser, un faible taux d'erreur de 2% génère à grande échelle beaucoup d'erreurs.

On peut aussi citer la base d'images annotées ImageNet et son concours ImageNet Large Scale Visual Recognition Challenge (ILSVRC) organisé de 2010 à 2017. Chaque année sont produits des centaines de réseaux permettant de gagner quelques dixièmes de pourcent de précision. Cela peut aussi paraître futile mais à l'échelle de la conduite automatique, où l'algorithme peut fonctionner 8h par jour sur plusieurs jours et dans plusieurs millions de voitures, la recherche d'un taux d'erreur le plus faible possible peut se comprendre.

9.2.2 LeNet

La reconnaissance des chiffres manuscrits est un problème historique de la classification d'images et le réseau LeNet, spécialement créé pour cette problématique, fait partie des classiques aujourd'hui. LeNet est un réseau neuronal convolutif proposée par Yann LeCun et al. en 1989 a été construit en utilisant des couches Conv, Pool et Linear qui représentent aujourd'hui des couches incontournables pour la création de CNN. Nous présentons quatre étapes de l'évolution de ce réseau dans la suite.

9.2.3 LeNet-1



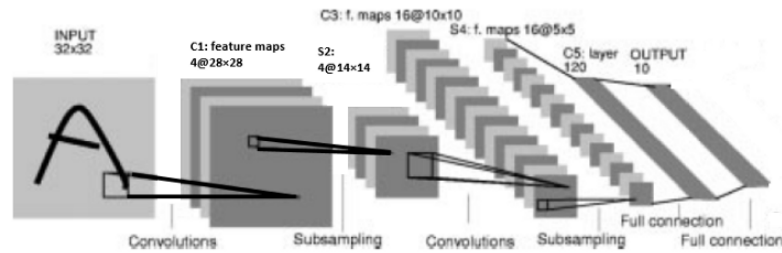
Topologie :

- Input : 28×28 input image
- Conv : 4 features 5×5 → (4,24,24)
- Average Pooling 2×2 → (4,12,12)
- Conv : 12 features 5×5 → (12,8,8)
- Average Pooling 2×2 → (12,4,4)
- Linear – 10 neurones

LeNet-1 atteint un niveau d'erreur de 1,7% sur les données de test. Lorsque LeNet-1 a été mis en place, les auteurs ont utilisé des couches average-Pool 2x2. Aujourd'hui, on trouve plus souvent des couches max-Pool 2x2 car à l'usage, il semble qu'elles accélèrent l'apprentissage. Le fait de choisir les valeurs les plus fortes permet au réseau de se focaliser sur les features les plus importants.

Les fonctions d'activation utilisées à l'époque étaient tanh et sigmoïde. La fonction ReLU n'existait pas encore. Aujourd'hui, il semble que la fonction ReLU soit largement préférée car elle accélère grandement la convergence durant l'entraînement.

9.2.4 LeNet-4

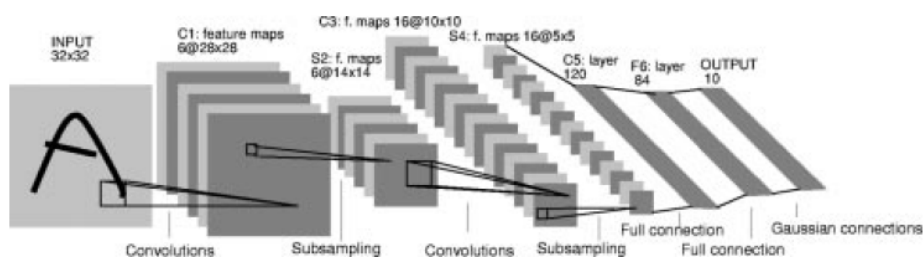


Topologie :

- Input : 32x32 input image
- Conv : 4 features 5x5 → (4,28,28)
- Average Pooling 2x2 → (4,14,14)
- Conv : 16 features 5x5 → (16,10,10)
- Average Pooling 2x2 → (16,5,5)
- Linear 120 neurones
- Linear 120 neurones

Avec plus de features et une couche Linear supplémentaires, le taux d'erreur descend à 1,1%.

9.2.5 LeNet-5



LeNet-5, plus communément appelé "LeNet" amène finalement peu de modifications par rapport à LeNet-4.

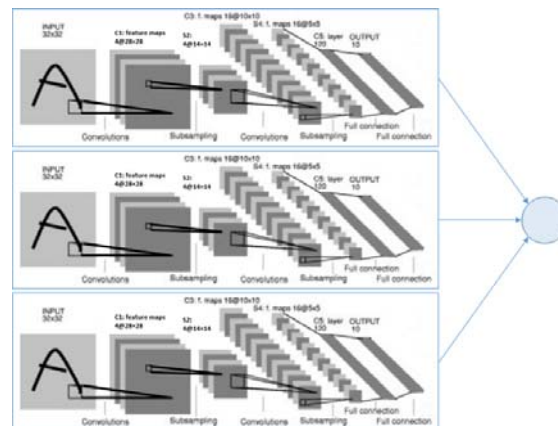
Topologie :

- Input : 32x32 input image
- Conv : 6 features 5x5 → (6,28,28)
- Average Pooling 2x2 → (6,14,14)
- Conv : 16 features 5x5 → (16,10,10)
- Average Pooling layers 2x2 → (16,5,5)

- Linear - 120 neurones
- Linear – 84 neurones
- Linear – 10 neurones

En augmentant le nombre de features et en ajoutant une couche Linear, le taux d'erreur atteint cette fois 0.95% lors des tests.

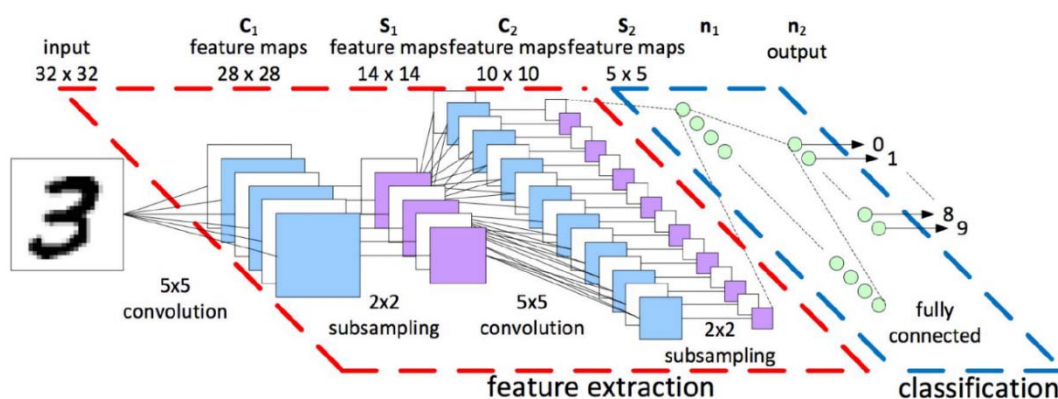
9.2.6 Boosted – LeNet 4



Le boosting est une technique consistant à combiner les résultats de plusieurs réseaux pour obtenir une réponse plus précise. Vous utilisez aussi cette technique lorsque vous demandez un conseil à différents experts pour avoir plusieurs avis. Dans cette version, la fusion des résultats consistait simplement à cumuler les scores obtenus par chaque réseau.

En utilisant la méthode du boosting sur 3 réseaux de type LeNet-4, le taux d'erreur est tombé à 0.7% soit une meilleure précision que LeNet-5. Par la suite, la technique du boosting a été largement utilisée pour d'autres réseaux afin d'améliorer les meilleurs résultats connus.

9.2.7 Extraction de features et classification



La partie gauche du réseau, proche de l'entrée, cherche à évaluer des critères (features en anglais) permettant à la partie droite du réseau de faciliter sa décision. Ainsi, les réseaux utilisés en classification d'images ont tendance à se séparer en deux parties :

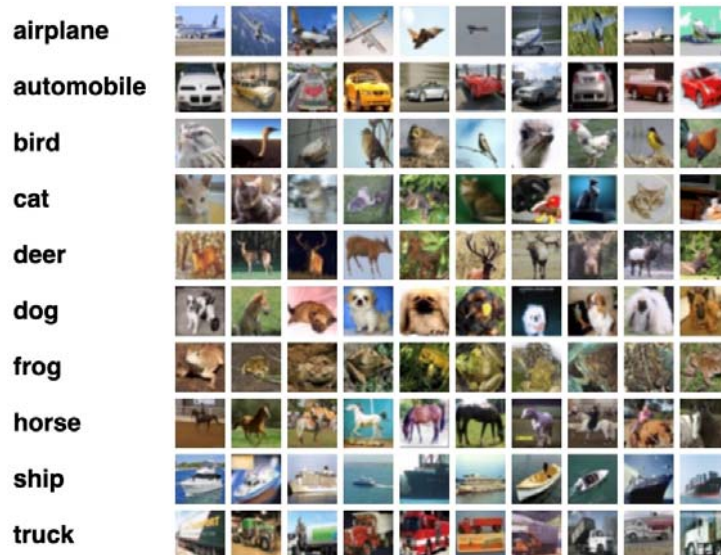
- La partie gauche, proche des données d'entrée, a pour objectif d'estimer et de trouver un ensemble de critères « feature extraction » ceci en empilant plusieurs couches Conv.

- La partie droite doit construire sa réponse et apprendre à classifier l'image à partir des critères évalués par la partie gauche du réseau. Pour mettre en place cette partie, on utilise plusieurs couches Linear empilées.

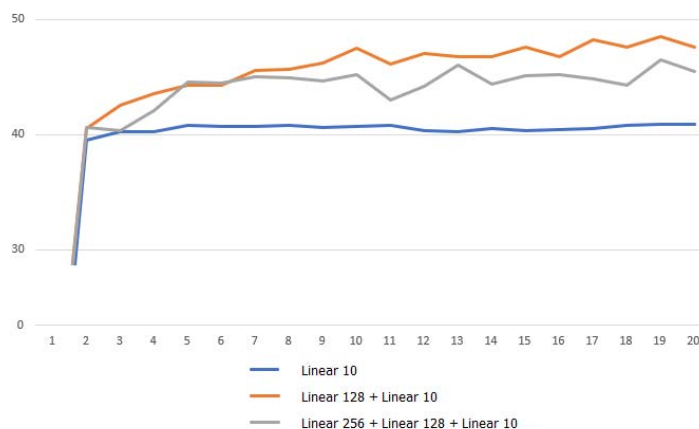
9.3 Exercice 9 : CNN avec CIFAR10

9.3.1 Mise en place

Ouvrez le fichier nommé « Ex Convolution.py ». Nous allons maintenant travailler sur la base d'images de CIFAR10 définissant 10 classes d'images :



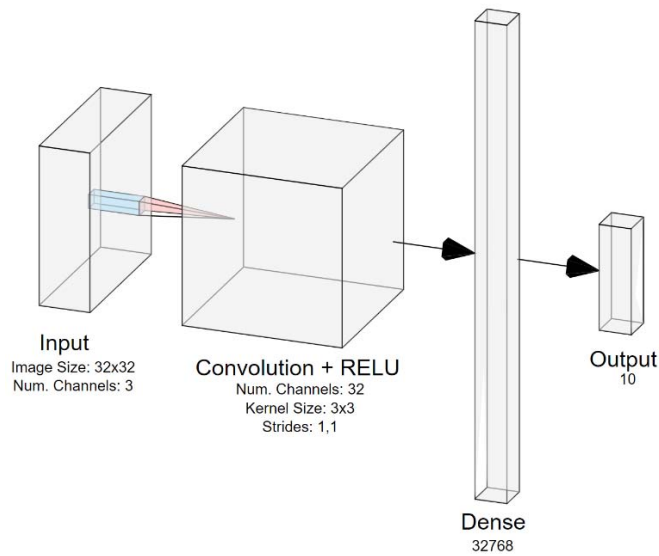
Le réseau présent dans l'exemple utilise une première couche Linear suivie d'une deuxième couche Linear de 10 neurones. En lançant le programme, normalement, le taux maximal de prédiction devrait être autour des 48%. En comparaison, un réseau avec une seule couche Linear de 10 neurones n'atteint que 41%, il y a donc eu un gain en rajoutant une couche. Par contre, l'utilisation de 3 couches Linear 256/128/10 n'augmente pas les performances. Nous présentons dans le graphique ci-dessous les résultats obtenus sur 20 epochs :



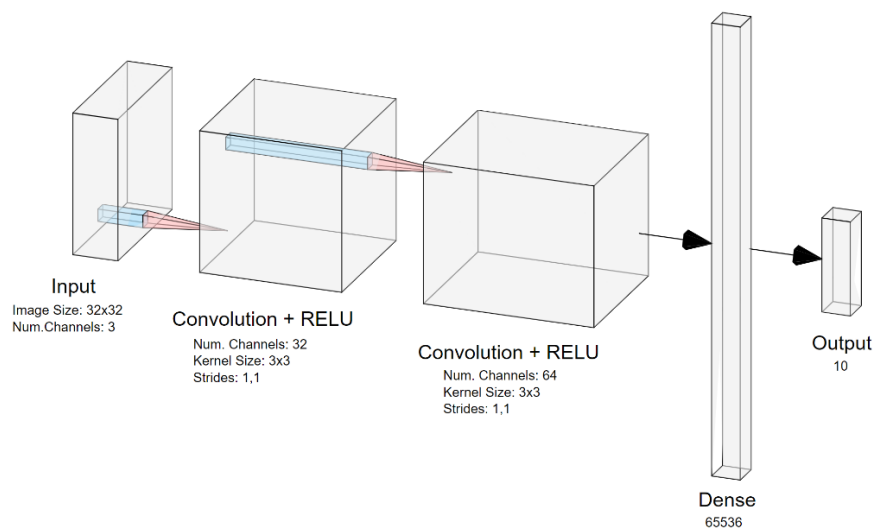
Vous remarquez que le taux de prédiction est largement en dessous des 98% obtenus sur la base MNIST. Pour améliorer les performances, vous allez mettre en place des couches convolutionnelles qui vont permettre d'améliorer les prédictions.

9.3.2 A votre tour

On vous propose d'implémenter deux réseaux à partir de leurs descriptions graphiques ci-dessous.



Réseau 1 :



Réseau 2 :

Il vous sera demandé d'envoyer le code des deux réseaux (dans un seul fichier) par mail avec le graphique des performances obtenues pour comparaison. Il est conseillé d'aller jusqu'à 80-120 epochs.