

[IAR] Apprentissage supervisé

Apprentissage profond pour la classification d'images

Nicolas Perrin-Gilbert

perrin@isir.upmc.fr



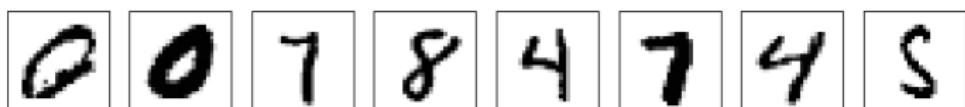
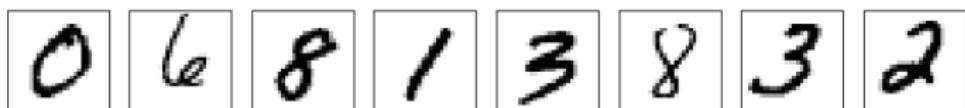
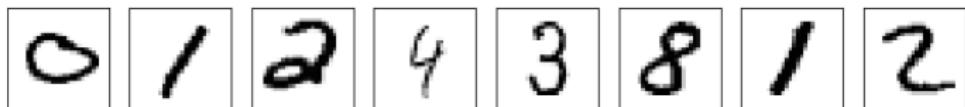
Exemple avec MNIST

Architectures plus complexes

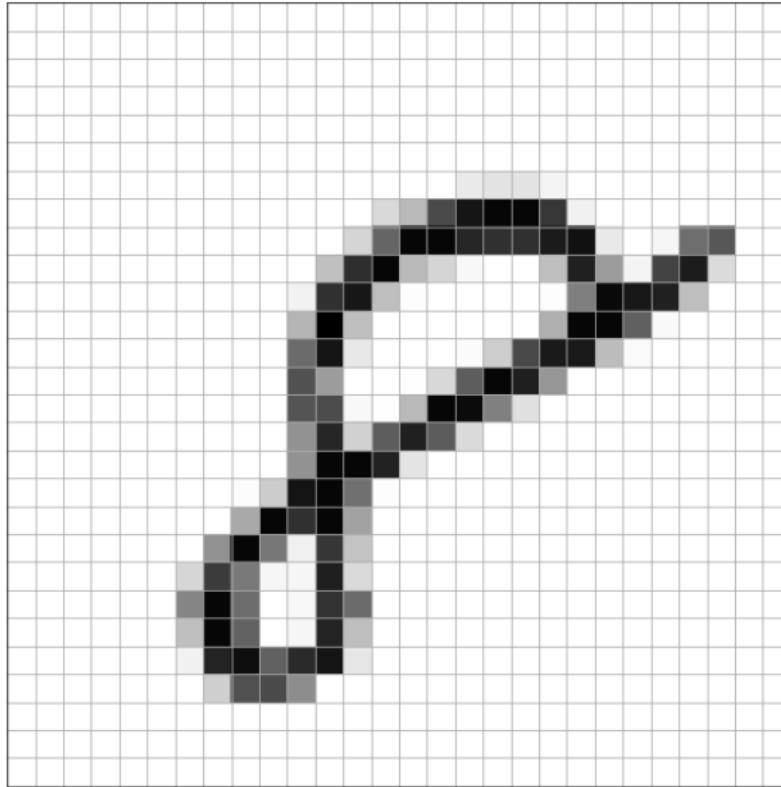
TD

GANs

MNIST : une base d'images de référence



Entrées: 28 x 28 pixels (=784) en niveaux de gris



MNIST

Un ensemble d'entraînement de taille 60000 :

```
mnist_train_dataset = datasets.MNIST(root='./data',
                                      train=True,
                                      download=True,
                                      transform=torchvision.transforms.ToTensor())
```

```
>>> len(mnist_train_dataset.data)
60000
```

MNIST

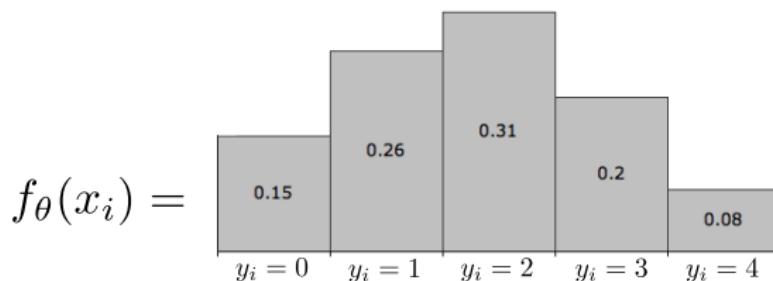
Et un ensemble de test de taille 10000 :

```
mnist_test_dataset = datasets.MNIST(root='./data',  
                                    train=False,  
                                    download=True,  
                                    transform=torchvision.transforms.ToTensor())
```

```
>>> len(mnist_test_dataset.data)  
10000
```

La classification

- ▶ Pour des données x_1, x_2, \dots, x_n appartenant à un ensemble X (typiquement un espace vectoriel de dimension finie), on dispose d'étiquettes y_1, y_2, \dots, y_n prenant leurs valeurs dans un ensemble fini $Y = \{0, \dots, k\}$.
- ▶ On dispose également d'une fonction paramétrée f_θ qui prend en entrée un élément de X et renvoie une distribution de probabilité discrète p de support Y .



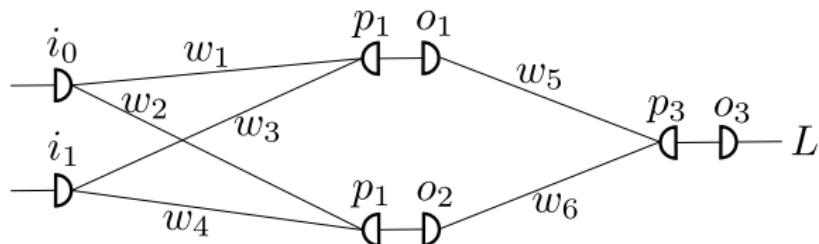
La classification

L'objectif de la classification est d'entraîner f_θ , c'est-à-dire ajuster θ de manière à approximer de mieux en mieux la vraie distribution des étiquettes y_i sur les données observées.

Plus simplement, on veut que les prédictions faites par f_θ s'améliorent : si par exemple la distribution $p = f_\theta(x_i)$ donne une probabilité élevée pour l'événement $y_i = 3$, alors on souhaite qu'il y ait effectivement de bonnes chances pour que l'étiquette correspondant à x_i soit 3.

Les réseaux de neurones

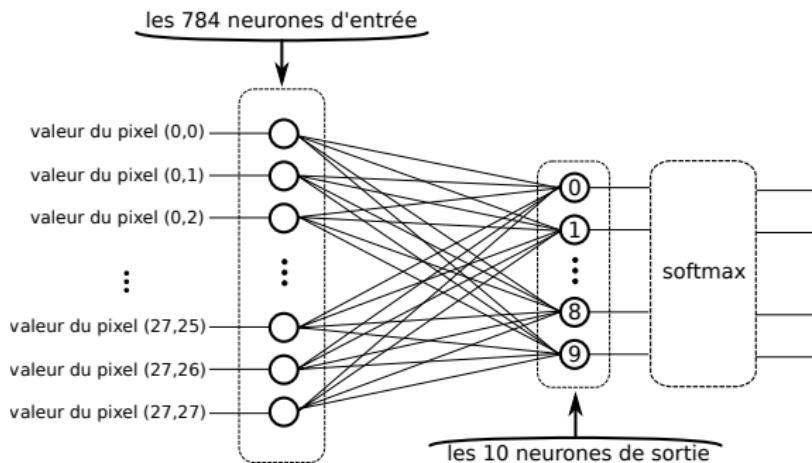
On choisit de représenter f_θ par un réseau de neurone. Voici un exemple de réseau très simple :



Les paramètres θ de f_θ sont les poids du réseau (les w_i).

Les réseaux de neurones

Pour MNIST, on commencera par utiliser un réseau dense sans aucune couche cachée :



Softmax

La fonction softmax permet de transformer les valeurs de la couche de sortie en valeurs positives dont la somme est 1, donc en une distribution de probabilité discrète. Il s'agit d'une exponentielle renormalisée :

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

Softmax

Voici le code pytorch utilisé pour définir le réseau de neurone précédent :

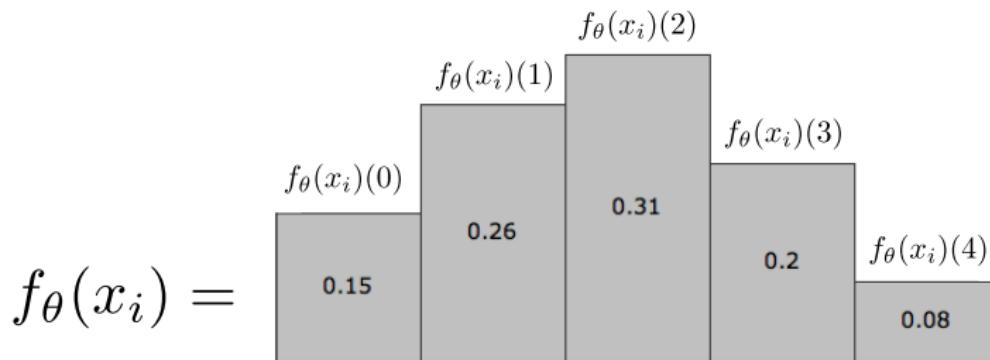
```
class Reseau1(nn.Module):
    def __init__(self):
        super(Reseau1, self).__init__()
        self.layer = nn.Linear(784, 10)

    def forward(self, x):
        x = F.softmax(self.layer(x.view(-1,784)), dim=1)
        return x

reseau1 = Reseau1().to(device)
```

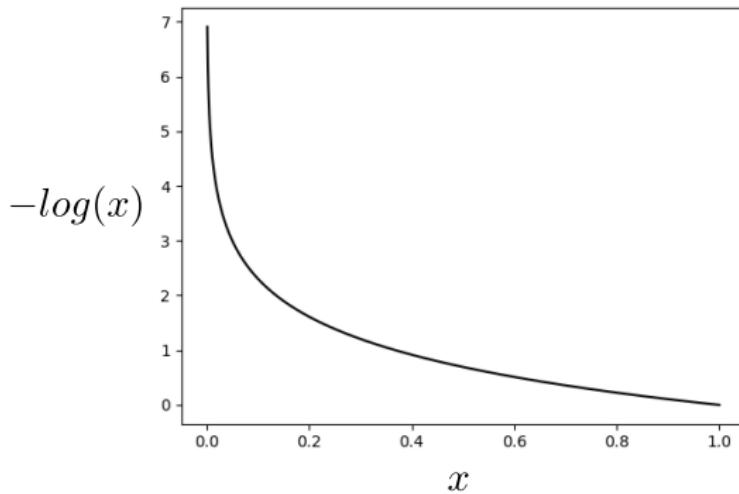
Entraînement sur des "batchs"

Nous allons entraîner notre réseau itérativement, en tirant à chaque fois des "batchs" (des sous-ensembles de taille fixe) tirés aléatoirement parmi les données d'entraînement. Considérons un batch $(B = (x_0, y_0), (x_1, y_1), \dots, (x_k, y_k))$. On note $f_\theta(x_i)(y_0)$ la probabilité estimée par le réseau pour l'étiquette y_i lorsque l'observation est x_i :



Entraînement sur des "batchs"

Sur le batch ($B = (x_0, y_0), (x_1, y_1), \dots, (x_k, y_k)$), intuitivement l'objectif est d'augmenter les probabilités $f_\theta(x_i)(y_i)$. On utilise pour cela la "negative log-likelihood function" (il y a d'autres choix possibles) : à la place de directement tenter d'augmenter $f_\theta(x_i)(y_i)$, on tente de minimiser $-\log(f_\theta(x_i)(y_i))$.



Entraînement sur des "batchs"

Pour combiner les différents éléments du batch, on fait simplement une somme des objectifs (une moyenne est possible également), ce qui nous donne un objectif global qui correspond à la minimisation de la fonction de coût suivante :

$$L^B(\theta) = \sum_i -\log(f_\theta(x_i)(y_i))$$

Remarque : la minimisation de l'espérance de $L^B(\theta)$ correspond à une minimisation de la "cross-entropy" entre les vraies distribution de probabilité $p(y_i|x_i)$ et les distributions de probabilité $f_\theta(x_i)(y_i)$ de notre modèle.

L'algorithme d'apprentissage fonctionne de la manière suivante :

- ▶ Définir aléatoirement un batch B .
- ▶ Effectuer une petite modification de θ pour diminuer légèrement $L^B(\theta)$ si possible.
- ▶ Recommencer.

Entraînement sur des "batchs"

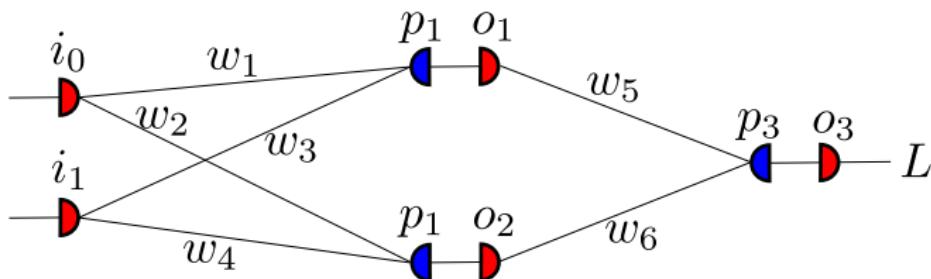
Il ne nous manque donc plus qu'à réussir à calculer les petites modifications à appliquer à θ pour faire diminuer $L^B(\theta)$. Cela se fait en calculant le gradient selon θ de $L^B(\theta)$:

$$\nabla_{\theta} L^B(\theta)$$

Remarque : le gradient est une généralisation de la dérivée aux fonctions de plusieurs variables. Pour les réseaux de neurones, il existe une façon particulière de calculer ce gradient : l'algorithme de rétropropagation du gradient (ou "backpropagation").

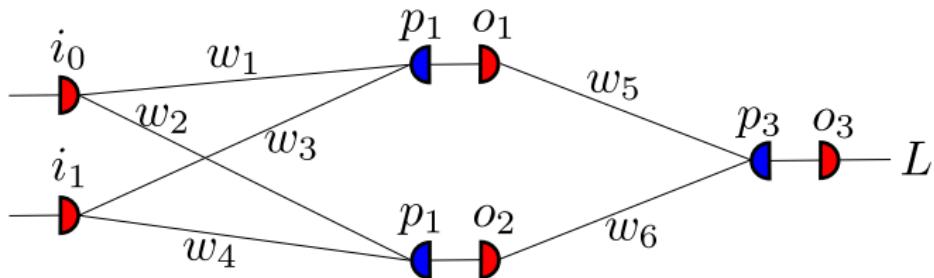
Rétropropagation du gradient

Exemple avec un petit réseau de neurones, dans lequel les neurones (p_1, o_1) , (p_2, o_2) et (p_3, o_3) ont des fonctions d'activations non-linéaires (typiquement : ReLU, tanh, sigmoïde, etc.) :



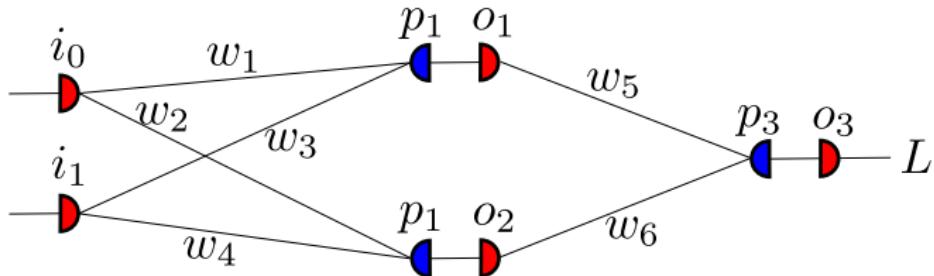
L est une fonction de coût qui dépend uniquement de o_3 , et dont on connaît la dérivée par rapport à o_3 : $\frac{\partial L}{\partial o_3}$.

Rétropropagation du gradient



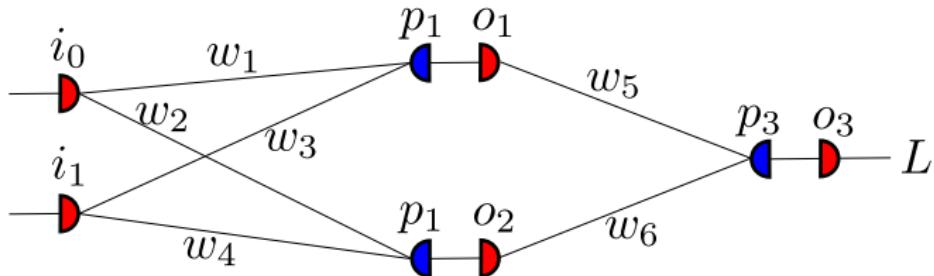
La première passe de l'algorithme (passe "forward") va de la gauche vers la droite, et calcule toutes les valeurs p_i et o_i .

Rétropropagation du gradient



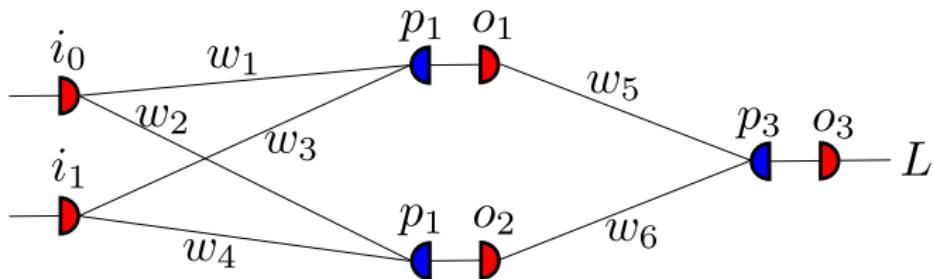
La seconde passe de l'algorithme va de droite à gauche, c'est celle qui va permettre le calcul du gradient. L'objectif est de connaître les dérivées partielles $\frac{\partial L}{\partial w_i}$ pour tous les paramètres w_i (il y a aussi en général des biais qu'on ne fait pas apparaître ici). Ces dérivées peuvent se traduire par la question suivante : de combien varie L si on fait varier w_i d'une valeur infinitésimale δw_i ?

Rétropropagation du gradient



Comme on connaît la dérivée des fonctions d'activation, on connaît tous les $\frac{\partial o_i}{\partial p_i}$. On peut donc commencer par calculer $\frac{\partial L}{\partial p_3} = \frac{\partial L}{\partial o_3} \frac{\partial o_3}{\partial p_3}$. Lorsqu'on connaît tous les $\frac{\partial L}{\partial p_i}$ pour une couche, on peut passer à la couche précédente.

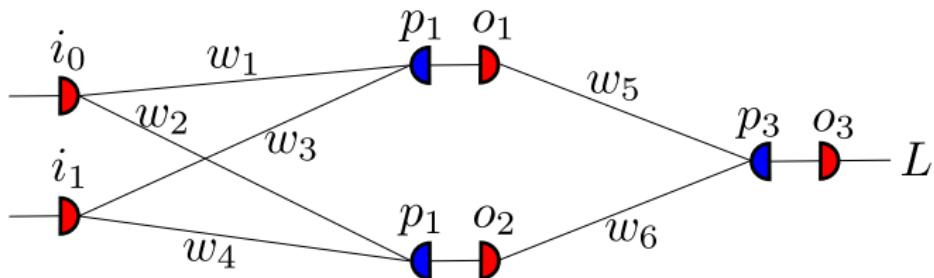
Rétropropagation du gradient



Par exemple, une variation de δw_5 sur w_5 provoque une variation de p_3 de $\delta w_5 o_1$. On en déduit que :

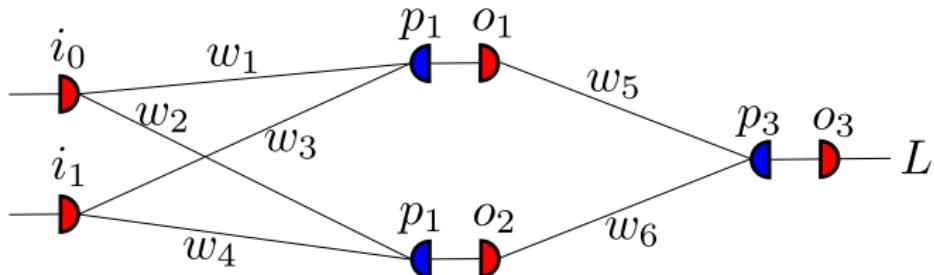
$$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial p_3} o_1$$

Rétropropagation du gradient



Pour calculer $\frac{\partial L}{\partial o_1}$, il suffit de regarder toutes les connections partant de o_1 . Ici, il n'y a que la connection de poids w_5 . Une variation de δo_1 fait varier p_3 de $\delta o_1 w_5$. On a donc : $\frac{\partial L}{\partial o_1} = \frac{\partial L}{\partial p_3} w_5$. Si il y avait eu plusieurs connections partant de o_1 , il aurait suffit de faire la somme de ces expressions pour obtenir $\frac{\partial L}{\partial o_1}$.

Rétropropagation du gradient



Une fois que l'on a obtenu $\frac{\partial L}{\partial o_1}$, et $\frac{\partial L}{\partial o_2}$, on obtient facilement $\frac{\partial L}{\partial p_1}$ et $\frac{\partial L}{\partial p_3}$. On peut alors recommencer le processus pour progresser de couche en couche vers la gauche.

Une fois la passe "backward" complète, toutes les valeurs $\frac{\partial L}{\partial w_i}$ sont connues, ce qui donne le gradient $\nabla_{\theta} L^B(\theta)$ (les paramètres θ sont ici les poids w_i).

Boucle d'entraînement

Nous sommes maintenant en mesure de réaliser les itérations de la boucle d'entraînement :

- ▶ Définir aléatoirement un batch
 $B = (x_0, y_0), (x_1, y_1), \dots, (x_k, y_k)$.
- ▶ Calculer $\nabla_{\theta} L^B(\theta)$, avec $L^B(\theta) = \sum_i -\log(f_{\theta}(x_i)(y_i))$.
- ▶ $\theta \leftarrow \theta - \eta \nabla_{\theta} L^B(\theta)$.
- ▶ Recommencer.

η est le taux d'apprentissage ("learning rate").



Boucle d'entraînement

Dans l'implémentation pytorch, c'est un "optimizer" qui se charge de calculer les gradients et de modifier les paramètres du réseau à chaque itération. Il peut se définir juste après la définition du réseau :

```
reseau1 = Reseau1().to(device)
optimizer_reseau1 = torch.optim.SGD(reseau1.parameters(),
    lr=learning_rate, momentum=momentum)
```

Boucle d'entraînement

Remarque : ici pour l'optimizer on a fait le choix le plus simple : "SGD" (stochastic gradient descent), avec un terme d'inertie ("momentum"). Il existe de nombreux méthodes de descente de gradient adaptatives, et avec des réseaux plus complexes on se tourne souvent vers des méthodes un peu plus avancées comme "Adam".

Le choix de la méthode de descente de gradient peut avoir un impact très important sur les performances.

Boucle d'entraînement

Voici la boucle d'entraînement dans le code :

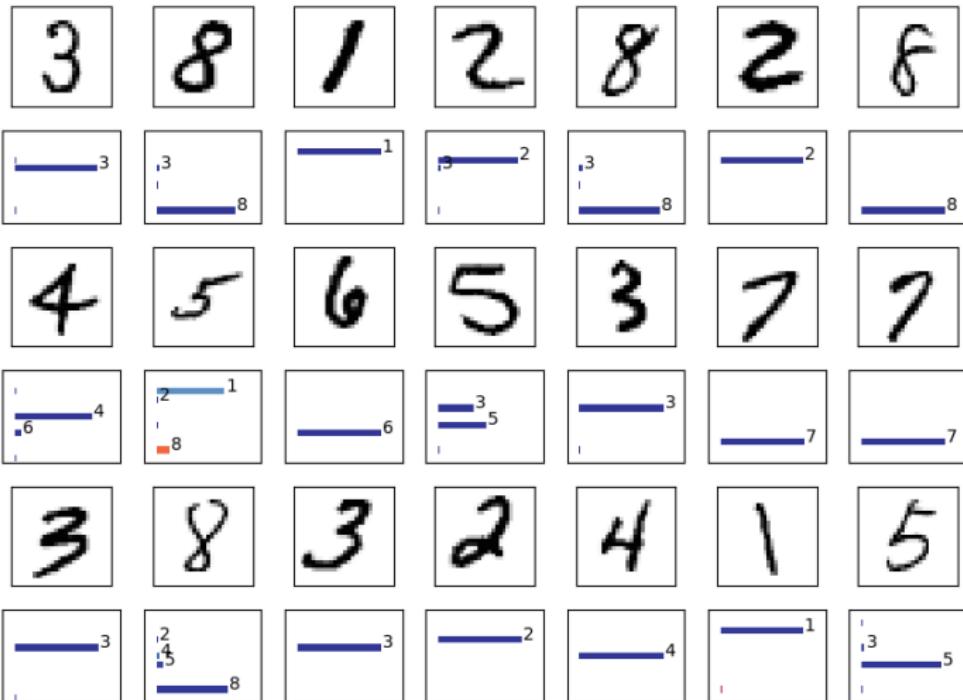
```
for i in range(nb_iterations):
    batch_idx, (data, targets) = next(train_loader)
    outputs = reseau1(data)
    loss = F.nll_loss(outputs, targets, reduction='sum')
    # loss = F.cross_entropy(outputs, targets)
    optimizer_reseau1.zero_grad()
    loss.backward()
    optimizer_reseau1.step()
```

Résultats

Juste après initialisation, le réseau, si on évalue le réseau sur un batch de l'ensemble de test (de taille 1000), on constate que le réseau effectue une prédiction correcte dans à peu près 10% des cas (en considérant la prédiction comme l'étiquette de plus haute probabilité), ce qui est logique pour une prédiction aléatoire avec 10 choix possibles au total (et donc 9 chances sur 10 de se tromper).

Après 1000 itérations sur des batchs de taille 60 (une "epoch" : toutes les données d'entraînement sont vues une fois), les prédictions sont cette fois correctes à environ 90% (toujours sur des données de test, qui n'ont pas été vues pendant l'entraînement).

Résultats



Résultats

Il est remarquable de voir qu'un réseau très simple peut mener à des résultats très corrects sur MNIST, mais c'est en partie dû à la simplicité de MNIST.

Et entraîner le réseau plus longtemps ne permet pas vraiment d'obtenir une précision nettement plus élevée que $90 \sim 92\%$.

Exemple avec MNIST

Architectures plus complexes

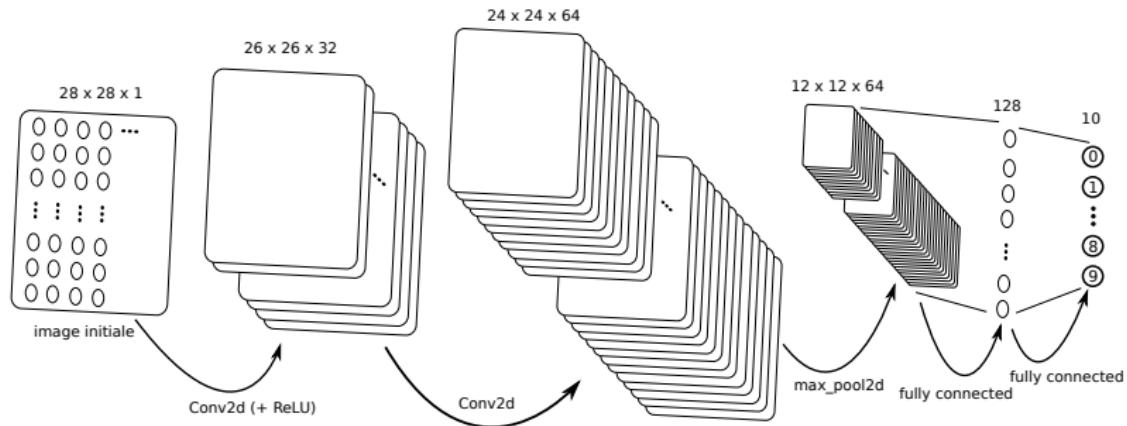
TD

GANs

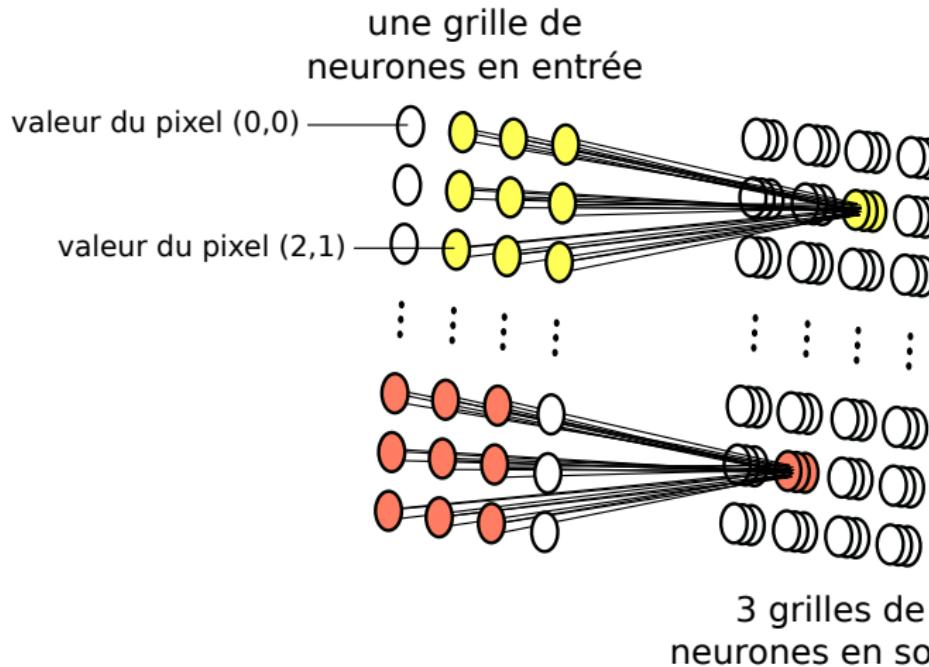
```
class Reseau2(nn.Module):
    def __init__(self):
        super(Reseau2, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.conv2(x)
        x = self.dropout1(F.max_pool2d(x, 2))
        x = torch.flatten(x, 1)
        x = self.dropout2(F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x
```

Convolutions, Max-pooling, Dropout



Couche de convolution (filtre de taille 3x3)



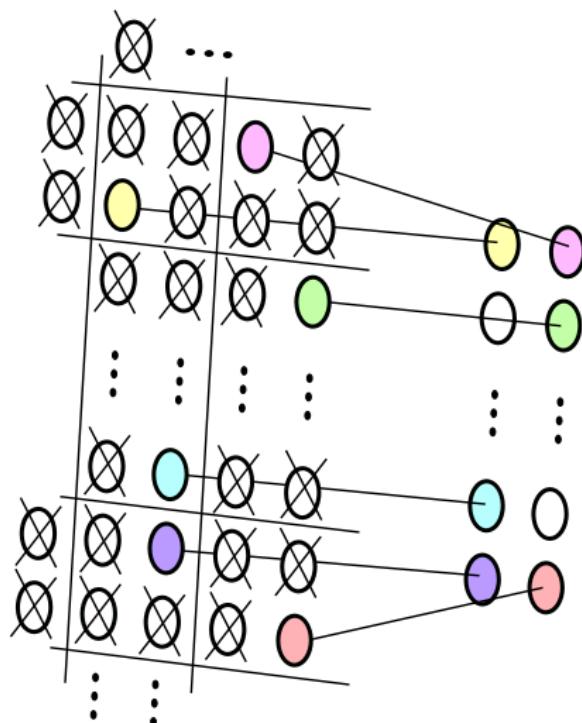
Couche de convolution (filtre de taille 3x3)

Les neurones jaunes et rouges forment ce qu'on appelle les filtres, où chaque neurone dans les grilles en sortie est relié à des neurones de son voisinage dans les grilles en entrée. Ici l'entrée de ces filtres (qu'on appelle champs réceptif) est de taille $3 \times 3 \times 1$ car il n'y a qu'une seule grille en entrée, mais les filtres ont toujours la même profondeur que l'entrée ; donc si l'entrée était composée de plusieurs grilles de neurones, par exemple 5, alors la taille des filtres serait $3 \times 3 \times 5$. Une particularité importante des réseaux convolutifs est que les filtres ont des *poids partagés*.

Concrètement, cela veut dire que sur le schéma, les 9×3 "fils" du haut (filtre jaune) ont exactement les mêmes poids que les 27 "fils" du bas (filtre rouge). Donc le nombre total de paramètres de toute la couche de convolution est en fait égal aux nombre de paramètres d'un seul filtre.



Max-pooling



Dropout

La technique du dropout consiste à "éteindre" aléatoirement un certain pourcentage des neurones d'une couche à chaque pas d'apprentissage. Cela rend les neurones moins dépendants les uns des autres et empêche le surapprentissage (ou "overfitting") qui se traduit par une très grande variabilité et de mauvais résultats sur des données nouvelles (c'est-à-dire une mauvaise capacité de généralisation).

Résultats

Avec l'architecture proposée, le temps d'apprentissage est nettement plus long (car il y a beaucoup plus de paramètres), mais il est possible d'atteindre en 1 ou 2 epochs une précision de 97~98%.

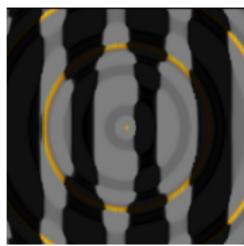
Exemple avec MNIST

Architectures plus complexes

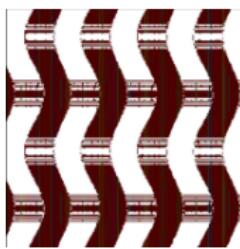
TD

GANs

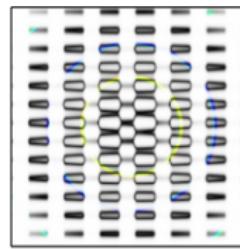
- ▶ Tester des réseaux simples et plus complexes sur la base de données CIFAR 10.
- ▶ Tenter d'obtenir plus de 50% de précision sur CIFAR 10.
- ▶ Trouver des images sur lesquelles le réseau se trompe beaucoup.
- ▶ Pour montrer que le réseau est loin d'apprendre comme un humain, introduire une corruption très légère des images et montrer que cela peut significativement réduire la qualité des prédictions du réseau (alors que les images restent très reconnaissables).
- ▶ Au contraire, on peut essayer de créer des images étranges mais pour lesquelles le réseau fait des prédictions fausses avec une grande confiance.



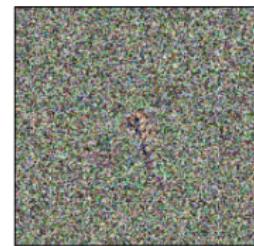
pingouin



guitare



télécommande



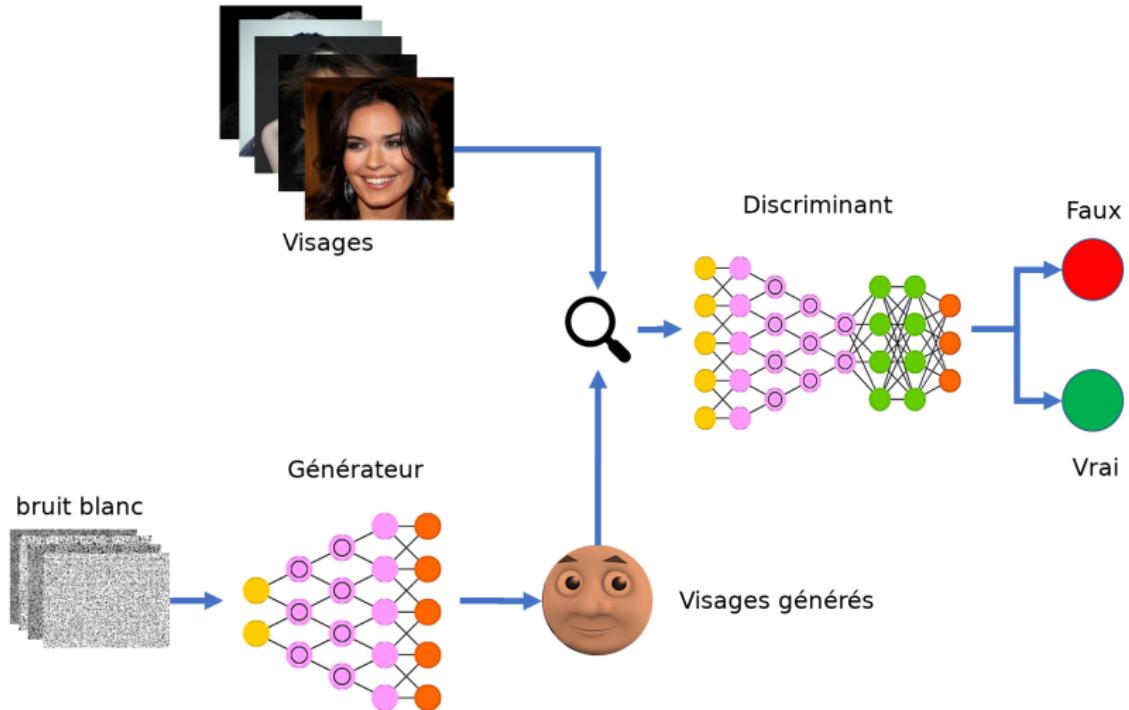
guépard

Exemple avec MNIST

Architectures plus complexes

TD

GANs





2014



2015



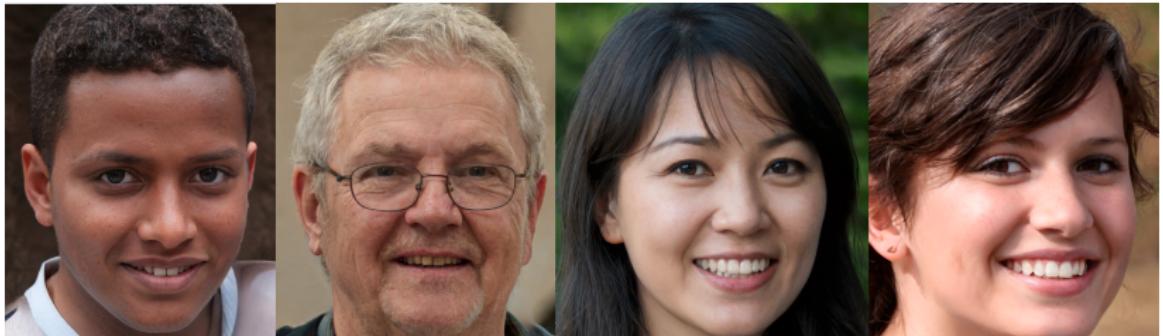
2016



2017



2019



<https://thispersondoesnotexist.com/>

