Université Mohammed V
Faculté des Sciences
Rabat

Concepts de base de l'apprentissage profond

Mehdi Bouskri

mehdi_bouskri@um5.ac.ma

Scalaires (tenseurs 0D)

- Un tenseur qui ne contient qu'un seul nombre est appelé un scalaire, tenseur scalaire ou encore tenseur 0-dimensionnel.
- Dans Numpy, un float32 ou float64 est un tenseur scalaire ,ou tableau scalaire. Vous pouvez afficher le nombre d'axes d'un tenseur Numpy via l'attribut **ndim.**
- Le nombre d'axes d'un tenseur est également appelé son rang.

Vecteurs (tenseurs 1D)

- Un tableau de nombres est appelé un vecteur, ou un tenseur 1D. On dit qu'un tenseur 1D a exactement un axe.
- La "dimensionnalité" peut désigner soit le nombre d'entrées le long d'un axe spécifique, soit le nombre d'axes dans un tenseur ce qui peut parfois créer une certaine confusion.
- il est plus correct de parler d'un "tenseur de rang 1" (le rang d'un tenseur étant le nombre d'axes).

Matrices (tenseurs 2D)

- Un tableau de vecteurs est une matrice, ou un tenseur de rang 2, donc elle possède deux axes, souvent désignés par lignes et colonnes.

Tenseurs 3D et tenseurs d'haute dimension

- Si vous regroupez ces matrices dans un nouveau tableau, vous obtenez un tenseur de rang 3, que vous pouvez visuellement interpréter comme un cube de nombres.

- En plaçant des tenseurs 3D dans un tableau, vous pouvez créer un tenseur 4D, et ainsi de suite.

- Dans l'apprentissage profond, vous manipulerez généralement des tenseurs de 0D à 4D, bien que vous puissiez aller jusqu'à 5D si vous traitez des données vidéo.

Principaux attributs des tenseurs

Un tenseur est défini par 3 attributs essentiels :

- Le nombre d'axes qu'il possède, son rang . Par exemple, un tenseur 3D a 3 axes, et une matrice a un rang de 2 axes. Ceci est également appelé le ndim du tenseur, dans la bibliothèque Numpy.

- Sa forme (shape). Il s'agit d'un tuple d'entiers qui décrit combien de dimensions le tenseur a le long de chaque axe.

- Son type de données (généralement appelé dtype dans les bibliothèques Python). C'est le type des données contenues dans le tenseur.

Manipulation des tenseurs dans Numpy

Trancher un tenseur (Tensor slicing)

- La sélection des éléments spécifiques d'un tenseur s'appelle le tranchage du tenseur
- En général, on peut choisir entre deux indices quelconques le long de chaque axe
- Il est également possible d'utiliser des indices négatifs. Comme les indices négatifs dans les listes, ils indiquent une position par rapport à la fin de l'axe.

Manipulation des tenseurs dans Numpy

Tranchage d'un tenseur en lots (Slicing a tensor into batches)

- Les modèles d'apprentissage profond ne traitent pas un ensemble de données entier en une seule fois, mais ils décomposent les données en petits lots

- Lorsqu'on considère un tel tenseur de lot, le premier axe (axe 0) est appelé "axe de lot" ou "dimension de lot". Il s'agit d'un terme que vous rencontrerez fréquemment lors de l'utilisation de Keras ou d'autres bibliothèques d'apprentissage profond.

Exemples concrets de tenseurs de données

Les données que vous manipulerez entreront presque toujours dans l'une des catégories suivantes:

- Données vectorielles : Tenseurs 2D de forme: (éléments, caractéristiques)
- Données de séries temporelles ou données séquentielles : Tenseurs 3D de forme : (éléments, timesteps, caractéristiques)
- Images : Tenseurs 4D de forme: (éléments, largeur, hauteur, canaux)
- Vidéos : Tenseurs 5D de forme: (éléments, images, largeur, hauteur, canaux)

Données vectorielles

- Dans un tel ensemble de données, chaque point de données peut être codé comme un vecteur, et donc un lot de données sera codé comme un tenseur 2D, où le premier axe est l'axe des éléments et le deuxième axe est l'axe des caractéristiques.

- Un ensemble de données où nous considérons pour chaque personne son âge, son code postal et son revenu. Chaque personne peut être caractérisée comme un vecteur de 3 valeurs, et ainsi un jeu de données complet de 100 000 personnes peut être stocké dans un tenseur 2D de forme (100000, 3).

Données séquentielles

- Lorsque le temps a de l'importance dans nos données, il est logique de les stocker dans un tenseur 3D avec un axe temporel spécifique.

- Chaque élément peut être codé comme une séquence de vecteurs (un tenseur 2D), et donc un lot de données sera codé comme un tenseur 3D. Par convention, l'axe temporel sera toujours le deuxième axe (axe d'indice 1).

Données séquentielles

- Considérons un ensemble de données sur les cours des actions. Chaque minute, on stocke le prix actuel de l'action, le prix le plus élevé de la dernière minute et le prix le plus bas de la dernière minute. Ainsi, chaque minute est codée sous la forme d'un vecteur, une journée entière de négociation est codée sous la forme d'un tenseur 2D de forme (390, 3) (il y a 390 minutes dans une journée de négociation), et 250 jours de données peuvent être stockés dans un tenseur 3D de forme (250, 390, 3). Ici, chaque élément correspond à un jour de données.

Données d'images

- Les images ont généralement trois dimensions : la largeur, la hauteur et la profondeur de couleur. Bien que les images en niveaux de gris n'aient qu'un seul canal de couleur et puissent donc être stockées dans des tenseurs 2D. Par convention, les tenseurs d'images sont toujours 3D, avec un canal de couleur à une dimension pour les images en niveaux de gris.

- Un lot de 128 images en niveaux de gris de taille 256x256 peut être stocké dans un tenseur 4D de forme (128, 256, 256, 1), et un lot de 128 images en couleurs peut être stocké dans un tenseur 4D de forme (128, 256, 256, 3).

Données vidéo

- Les données vidéo sont l'un des rares types de données pour lesquelles on aura besoin de tenseurs 5D. Une vidéo peut être considérée comme une séquence d'images, chaque image étant une image en couleur.

- Puisque chaque image peut être stockée dans un tenseur 3D, une séquence d'images peut être stockée dans un tenseur 4D, et donc un lot de différentes vidéos peut être stocké dans un tenseur 5D de forme: (éléments, images, largeur, hauteur, canaux)

Données vidéo

- Par exemple, un clip vidéo de 60 secondes, de taille 256x144, avec 4 images par seconde, comptera 240 images. Un lot de 4 clips vidéo de ce type serait stocké dans un tenseur 5D de forme: (4, 240, 256, 144, 3).

les opérations tensorielles

- De même que tout programme informatique peut être réduit à un petit ensemble d'opérations sur des entrées binaires, toutes les transformations effectuées par les réseaux de neurones profonds peuvent être réduites à une série d'opérations tensorielles appliquées à des tenseurs de données numériques. Par exemple, il est possible d'additionner des tenseurs, de multiplier des tenseurs, et ainsi de suite.

les opérations tensorielles

Considérons une couche qui ressemble à ceci:

```
keras.layers.Dense(6, activation='relu’)
```

Cette couche peut être interprétée comme une fonction qui prend en entrée un tenseur 2D et renvoie un autre tenseur 2D. Plus précisément, la fonction suivante:

output = relu(dot(W, input) + b)

les opérations tensorielles

Nous avons ici trois opérations tensorielles :

- un produit scalaire (dot) entre le tenseur d'entrée et un tenseur nommé W;
- une addition (+) entre le tenseur 2D résultant et un vecteur b;
- une opération relu(x) qui est simplement max(x,0).

Opérations par éléments

- L'opération relu et l'addition sont des opérations par élément, c'est-à-dire des opérations qui sont appliquées indépendamment à chaque entrée des tenseurs considérés.

- Cela signifie que ces opérations peuvent être mises en œuvre de manière parallèle, ce que l'on appelle implémentations vectorisées.

Une implémentation simple d'une opération relu par élément

```
def naive_relu(x):
    assert len(x.shape) == 2
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

Une implémentation simple de l'addition par éléments

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

Opérations par éléments

- Selon le même principe, on peut faire des multiplications, des soustractions et ainsi de suite.

- En pratique, ces opérations sont disponibles en tant que fonctions Numpy intégrées et optimisées, qui elles-mêmes délèguent les tâches les plus lourdes vers une implémentation BLAS (Basic Linear Algebra Subprograms).

Opérations par éléments

Dans Numpy, on peut donc faire les opérations suivantes, et le résultat sera rapide :

```
import numpy as np
# addition par éléments
z = x + y
# relu par élément
z = np.maximum(z, 0.)
```

Broadcasting

- Dans l'implémentation de naive_add , nous ne supportons que l'addition de tenseurs 2D de forme identique. Mais dans la couche dense que nous avons vue précédemment, on ajout un tenseur 2D à un vecteur.

- Dans le cas ou les tenseurs ont des formes différentes, si possible et s'il n'y a pas d'ambiguïté, le plus petit tenseur sera "broadcasté" pour correspondre à la forme du grand tenseur.

Broadcasting

Le broadcasting se fait en deux étapes :

- des axes sont ajoutés au petit tenseur pour correspondre au ndim du grand tenseur
- le petit tenseur est ensuite répété le long de ces nouveaux axes, pour correspondre à la forme complète du grand tenseur.

Broadcasting

Considérons x avec la forme (32, 10) et y avec la forme (10).

- D'abord, on ajoute un premier axe vide à y, dont la forme devient (1, 10).
- Ensuite, nous répétons y 32 fois le long de ce nouvel axe, de sorte que nous obtenons un tenseur Y de forme (32, 10) , où Y[i, :] == y pour i dans l'intervalle [1, 32].
- À ce point, nous pouvons procéder à l'ajout de x et Y, puisqu'ils ont la même forme.

Une implémentation simple de l'addition matrice-vecteur

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

Le produit scalaire

- L'opération du produit scalaire, également appelée "produit tensoriel" (à ne pas confondre avec le produit par élément) est la plus courante et la plus utile des opérations tensorielles.

- Le produit élémentaire est réalisé avec l'opérateur * dans Numpy, Keras, Theano et TensorFlow. Le produit scalaire utilise une syntaxe différente dans TensorFlow, mais dans Numpy et Keras, il est réalisé avec l'opérateur standard dot :

```
z = np.dot(x, y)
```

Une implémentation simple du produit scalaire

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

Réorganisation des tenseurs

- Réorganiser un tenseur signifie réorganiser ses lignes et ses colonnes afin de correspondre à une forme cible. Le tenseur remodelé aura évidemment le même nombre total de coefficients que le tenseur initial.

- Un cas particulier de remodelage que l'on rencontre fréquemment est la transposition. Transposer une matrice signifie échanger ses lignes et ses colonnes

L'optimisation par gradient

Revenant à la couche qu'on a interprétée comme la fonction suivante:

output = relu(dot(W, input) + b)

Dans cette expression, W et b sont des tenseurs qui sont des attributs de la couche. Ils sont appelés les "poids", ou "paramètres entraînables" de la couche.

Ces poids contiennent les informations que le réseau a apprit à partir de l'exposition aux données de l'entraînement.

L'optimisation par gradient

- Initialement, ces matrices de poids sont remplies de petites valeurs aléatoires (une étape appelée initialisation aléatoire).
- Bien sûr, il n'y a aucune raison de supposer que *relu(dot(W, input) + b)*, lorsque *W* et *b* sont aléatoires, donnera des représentations utiles.
- Ensuite, il s'agit d'ajuster progressivement ces poids, en fonction d'un signal de retour.
- Cet ajustement progressif, également appelé entrainement, constitue essentiellement l'apprentissage qui caractérise l'apprentissage automatique.

La boucle d'apprentissage

Répétez autant de fois que nécessaire :

1. Tirer un lot de données d'apprentissage x et les cibles correspondantes y
2. Exécuter le réseau sur x ("forward pass"), obtenir des prédictions y_pred
3. Calculer la perte du réseau sur le lot, mesurer la différence entre y_pred et y
4. Mettre à jour tous les poids du réseau d'une manière qui réduise légèrement la perte dans ce lot.

La dérivée

Considérons une fonction continue et uniforme: $f(x)=y$
qui fait correspondre un nombre réel x à un nouveau nombre réel y .
Puisque la fonction est continue, une petite variation de x ne peut produire qu'une petite variation de y.

Disons que vous augmentez x d'un petit facteur *epsilon_x*, cela implique une petite modification *epsilon_y* de y:

$$f(x + epsilon\_x) = y + epsilon\_y$$

Lorsque *epsilon_x* est suffisamment petit, autour d'un certain point p, il est possible d'approximer f comme une fonction linéaire de pente a :

$$f(x + epsilon\_x) = y + a * epsilon\_x$$

La dérivée

La pente *a* est appelée la dérivée de *f* en *p*. Si *a* est négatif, cela signifie que le petit changement de x autour de *p* entrainerait une diminution de *f(x)*, et si *a* est positif, alors ce changement de x entrainerait une augmentation de *f(x)*.

De plus, la valeur absolue de a (l'amplitude de la dérivée) nous indique à quelle vitesse cette augmentation ou cette diminution se produirait.

La dérivée d'une opération tensorielle

Un gradient est la dérivée d'une opération tensorielle. Il s'agit de la généralisation du concept de dérivée aux fonctions d'entrées multidimensionnelles.

Considérons un vecteur d'entrée x, une matrice W, une cible *y* et une fonction de perte *loss* . Nous utilisons W pour calculer un candidat cible y_pred, et nous calculons la perte entre le candidat y_pred et la cible y :

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y)
```

La dérivée d'une opération tensorielle

Si les données d'entrée x et y sont fixées, cela peut être interprété comme une fonction qui associe des valeurs de W à des valeurs de perte :

```
loss_value = f(W)
```

Supposons que la valeur courante de W est W0 . Alors la dérivée de f au point W0, est un tenseur gradient(f)(W0) de même forme que W , où chaque coefficient gradient(f)(W0)[i, j] indique la direction et l'amplitude du changement de loss_value que l'on observerait en modifiant W0[i, j] . Ce tenseur gradient(f)(W0) est le gradient de la fonction f(W) = loss_value dans W0 .

La dérivée d'une opération tensorielle

On peut diminuer f(W) en déplaçant W dans la direction opposée au gradient, par exemple W1 = W0 - step * gradient(f)(W0).

Cela signifie simplement "aller dans le sens opposé à la courbure", ce qui intuitivement devrait vous faire descendre plus bas sur la courbe.

Notez que le facteur step est nécessaire parce que le gradient(f)(W0) ne s'approche de la courbure que lorsque on est proche de W0 , donc on ne veut pas trop s'éloigner de ce point.

Descente de gradient stochastique

Appliqué à un réseau de neurones, cela implique de trouver la combinaison de valeurs de poids qui donne la plus petite fonction de perte possible. Cela se fait par résoudre l'équation suivante :

gradient(f)(W) = 0

Il s'agit d'une équation polynomiale de N variables, où N est le nombre de coefficients dans le réseau.

Descente de gradient stochastique

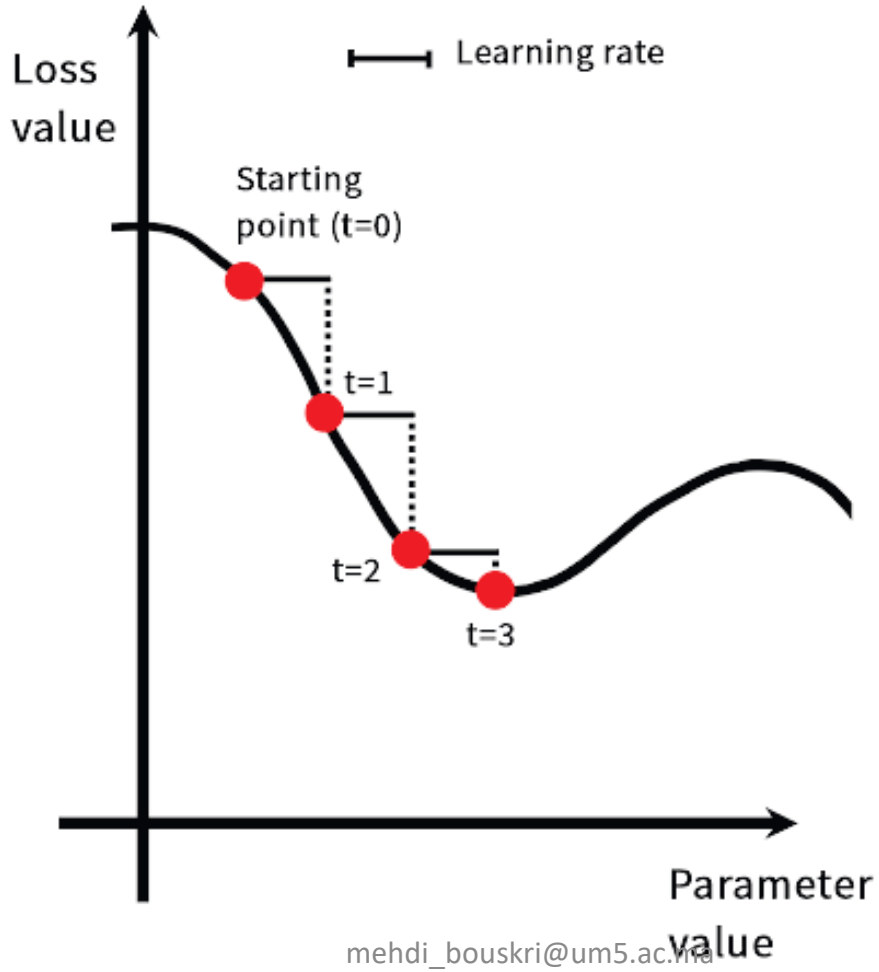
4.1) Calculer le gradient de la perte par rapport aux paramètres du réseau.("backward pass")

4.2) Déplacer un peu les paramètres dans la direction opposée au gradient, par exemple : `W -= step * gradient`, diminuant ainsi un peu la perte sur le lot.

Ce que nous venons de décrire est appelé "mini-batch Stochastic Gradient Descent" (mini-batch SGD). Le terme stochastique fait référence au fait que chaque lot de données est tiré aléatoirement.

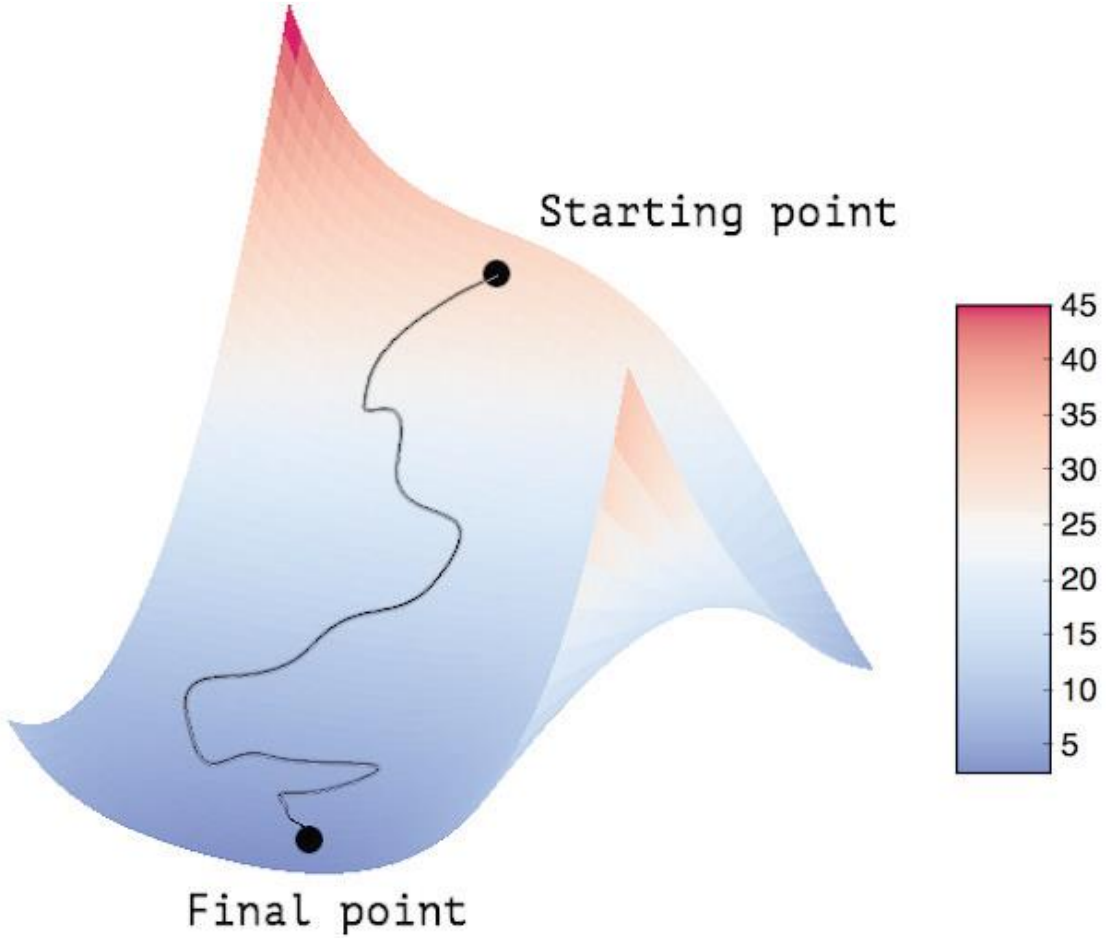
Descente de gradient stochastique

Figure1 :



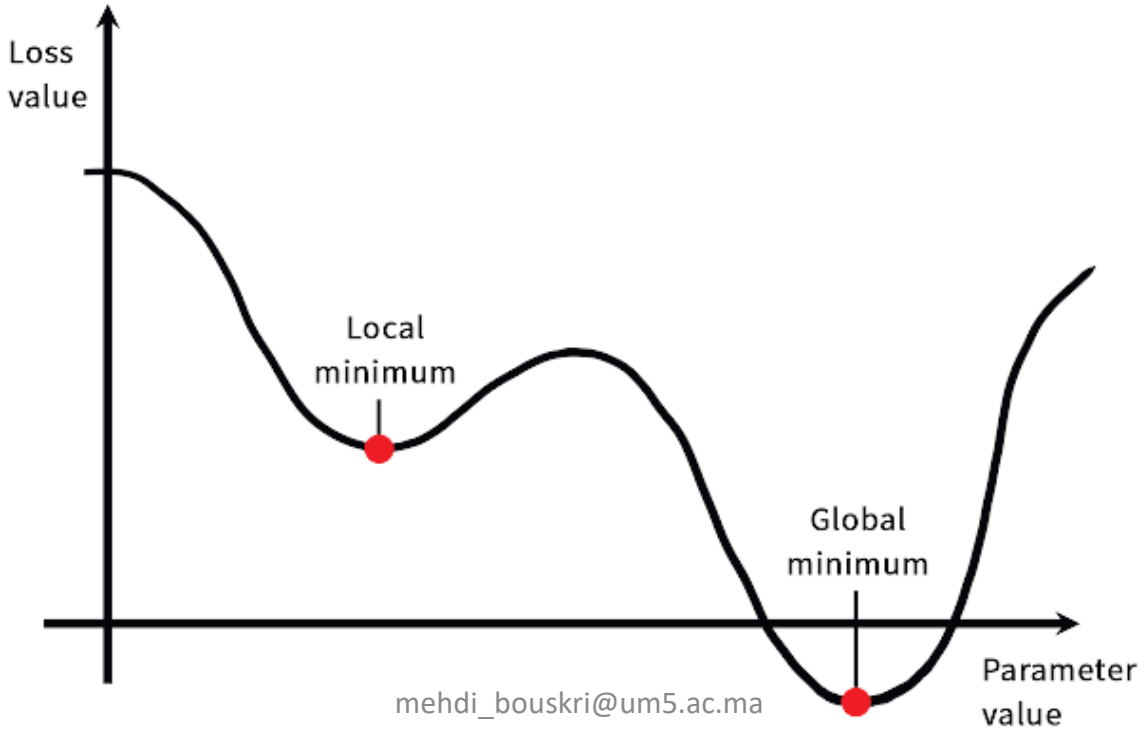
Descente de gradient stochastique

Figure 2:



SGD avec moment

Le concept de moment, qui est utilisé dans de nombreuses variantes du SGD avec moment, permet de résoudre deux problèmes liés à la SGD : la vitesse de convergence et les minimums locaux.



SGD avec moment

En pratique, cela signifie que la mise à jour du paramètre W est basée non seulement sur la valeur actuelle du gradient, mais aussi sur la mise à jour précédente du paramètre, comme dans cette implémentation simple :

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
     w, loss, gradient = get_current_parameters()
     velocity = past_velocity * momentum + learning_rate *
gradient
     w = w + momentum * velocity - learning_rate * gradient
     past_velocity = velocity
     update_parameter(w)
```

Enchaînement de dérivées : l'algorithme de rétropropagation

En pratique, une fonction de réseau de neurone est constituée de nombreuses opérations tensorielles enchaînées, chacune d'entre elles ayant une dérivée simple et connue.

Par exemple, il s'agirait d'un réseau f composé de trois opérations tensorielles a, b et c, avec des matrices de poids W1, W2 et W3 :

$$f(W1, W2, W3) = a(W1, b(W2, c(W3)))$$

Enchaînement de dérivées : l'algorithme de rétropropagation

Une telle chaîne de fonctions peut être dérivée en utilisant la "règle de la chaîne" :

$$f(g(x)) = f'(g(x)) * g'(x)$$

L'application de la règle de la chaîne au calcul des valeurs du gradient d'un réseau neuronal donne lieu à un algorithme appelé rétropropagation (backpropagation).

La rétropropagation

La rétropropagation commence par la valeur de perte finale et remonte des couches supérieures vers les couches inférieures, en appliquant la règle de la chaîne pour calculer la contribution de chaque paramètre à la valeur de perte.

On utilise des frameworks modernes capables de "différencier symboliquement", comme TensorFlow, pour implémenter les réseaux. Cela signifie que, étant donné une chaîne d'opérations avec une dérivée connue, on peut calculer un gradient pour la chaîne (en appliquant la règle de la chaîne) qui fait correspondre les valeurs des paramètres de la fonction du réseau aux valeurs du gradient. Lorsque vous avez accès à une telle fonction, le " backward pass" se réduit à un appel à cette fonction de gradient.

Exemple sur MNIST dataset

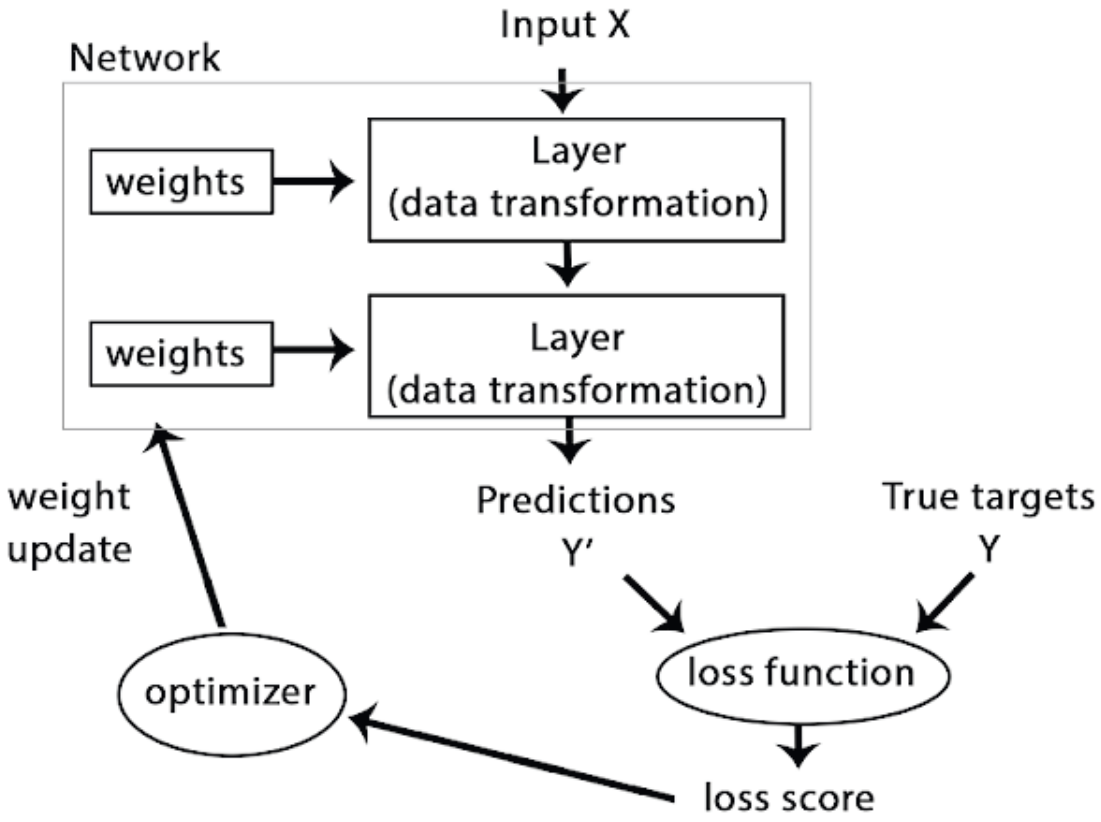
Anatomie d'un réseau de neurones

Comme nous l'avons vu, l'entrainement d'un réseau de neurones porte sur les éléments suivants :

- Les couches, qui sont combinées en un réseau (ou un modèle).
- Les données d'entrée et les cibles correspondantes.
- la fonction de perte, qui définit le signal de retour utilisé pour l'apprentissage.
- L'optimisateur, qui détermine la manière dont l'apprentissage se déroule.

Anatomie d'un réseau de neurones

On peut visualiser leur interaction de la manière suivante :



Les couches :
des éléments essentiels de l'apprentissage profond

Une couche est un module de traitement de données qui prend en entrée un ou plusieurs tenseurs et en sort un ou plusieurs.

La plupart des couches ont un état qui est les poids de la couche, représentés par un ou plusieurs tenseurs appris par descente de gradient, et qui contiennent la "connaissance" du réseau.

Différentes couches sont appropriées pour différents formats de tenseurs et différents types de traitement de données. Par exemple, les données vectorielles simples sont souvent traitées par des couches "entièrement connectées".

Les couches :
des éléments essentiels de l'apprentissage profond

Les données de séquence, stockées dans des tenseurs 3D, sont généralement traitées par des couches "récurrentes" comme des couches LSTM.

Les données d'image, stockées dans des tenseurs 4D, sont généralement traitées par des couches de convolution de deux dimensions.

Les couches :
des éléments essentiels de l'apprentissage profond

On peut considérer les couches comme les briques de l'apprentissage profond, une métaphore rendue explicite par des frameworks comme Keras.

La construction de modèles d'apprentissage profond dans Keras se fait en assemblant des couches compatibles pour former des pipelines de transformation de données utiles.

La notion de "compatibilité des couches" fait référence ici spécifiquement au fait que chaque couche n'acceptera que des tenseurs d'entrée d'une certaine forme, et retournera des tenseurs de sortie d'une forme particulière.

Les couches

Prenons l'exemple suivant :

```
from keras import layers
layer = layers.Dense(32, input_shape=(128,))
```

Nous avons créé une couche qui n'acceptera en entrée que des tenseurs 2D dont la deuxième dimension est 128, la première dimension n'est pas spécifiée et donc n'importe quelle valeur serait acceptée.

Cette couche retournera un tenseur où la deuxième dimension a été transformée pour être 32.

Les couches

Cette couche ne peut être connectée qu'à une autre qui attend des vecteurs à 32 dimensions comme entrée.

Lorsque vous utilisez Keras, vous n'avez pas besoin de vous occuper de la compatibilité, car les couches que vous ajoutez à vos modèles sont construites dynamiquement pour correspondre à la forme de la couche suivante. Par exemple:

```
model = models.Sequential()
model.add(layers.Dense(32, input_shape=(128,)))
model.add(layers.Dense(32))
```

Les modèles : des réseaux de couches

Un modèle d'apprentissage profond est simplement un graphe dirigé de couches. L'exemple le plus courant serait une pile linéaire de couches, qui relie une entrée unique à une sortie unique.

Cependant, en avançant, vous serez exposé à une variété plus large de topologies de réseau. Parmi les plus courantes, on trouve :

- Réseaux à deux branches
- Réseaux à têtes multiples
- Inception blocks

Les modèles : des réseaux de couches

La topologie d'un réseau définit un espace d'hypothèses. Alors que l'apprentissage automatique est défini comme la recherche de représentations utiles de certaines données d'entrée, dans un espace prédéfini de possibilités, en utilisant les informations fournies par un signal de feedback.

En choisissant une topologie de réseau, vous avez limité votre espace des possibilités (espace des hypothèses) à une série spécifique d'opérations tensorielles, en mettant en correspondance les entrées et les sorties.

Ce que vous chercherez alors, c'est un bon ensemble de valeurs pour les tenseurs de poids impliqués dans ces opérations tensorielles.

Les modèles : des réseaux de couches

Choisir la bonne architecture de réseau est plus un art qu'une science, et bien qu'il existe quelques bonnes pratiques et principes sur lesquels vous pouvez vous appuyer, seule la pratique peut réellement vous aider à trouver le bon modèle de réseau de neurones.

Fonctions de perte et optimisateurs

Une fois l'architecture du réseau est définie, il nous reste à choisir deux autres éléments :

- La fonction de perte (ou fonction objective), c'est-à-dire la quantité qui sera minimisée pendant l'apprentissage. Elle représente une mesure de réussite dans la tâche à accomplir.

- L'optimisateur, qui détermine comment le réseau va être actualisé par rapport à la fonction de perte. Il met en œuvre une variante spécifique de la descente de gradient stochastique.

Le choix des fonctions de perte

Il est extrêmement important de choisir la fonction objective adaptée au problème : votre réseau prendra tous les raccourcis possibles pour la minimiser. Par conséquent, si la fonction de perte ne correspond pas entièrement à la réussite réelle de la tâche à accomplir, votre réseau produira des résultats indésirables.

Lorsqu'il s'agit de problèmes courants tels que la classification, la régression ou la prédiction de séquences, il existe des directives simples que vous pouvez suivre pour choisir la bonne fonction de perte.

Le choix des fonctions de perte

Par exemple on peut utiliser la cross-entropy binaire pour un problème de classification à deux classes, la cross-entropy catégorielle pour un problème de classification à plusieurs classes, l'erreur quadratique moyenne pour un problème de régression, et la CTC pour un problème d'apprentissage de séquence.

Le développement de vos propres fonctions objectives ne sera nécessaire que lorsque vous travaillerez sur des problèmes spécifiques ou de recherche nouveaux.

Le choix des optimisateurs

Le choix d'un mauvais optimisateur peut avoir un impact négatif sur les performances de votre modèle d'apprentissage automatique.

Généralement, vous pouvez choisir le bon optimisateur en :

- Trouvant un article de recherche en rapport avec votre sujet et en commençant par utiliser le même optimisateur.
- Comparant les propriétés de votre ensemble de données aux forces et faiblesses des différents optimisateurs.

Le choix des optimisateurs

Optimisateur	État de mémoire	Paramètres réglables	Forces	Faiblesses
SGD	0	1	La meilleure généralisation possible	Sujet aux des minimums locaux, sensible à l'initialisation et au choix du taux d'apprentissage.
SGD avec moment	4n	2	Accélère dans les directions de descente régulière	sensible à l'initialisation du taux d'apprentissage et du moment
AdaGrad	4n	1	Fonctionne bien sur les données avec des caractéristiques dispersées, décroît automatiquement le taux d'apprentissage.	Se généralise moins bien, converge vers des minima aigus, le gradient peut disparaître
RMSprop	4n	3	Fonctionne bien sur les données avec des caractéristiques dispersées, moment intégré.	Généralisation plus mauvaise, convergence vers des minima aigus.
Adam	8n	3	Fonctionne bien sur des données avec des caractéristiques dispersées, bons paramètres par défaut, décroît automatiquement le taux d'apprentissage.	Généralisation plus mauvaise, convergence vers des minima aigus, nécessite beaucoup de mémoire pour l'état.
AdamW	8n	3	Amélioré par rapport à Adam en termes de généralisation, bassin plus large d'hyperparamètres optimaux.	Nécessite beaucoup de mémoire pour l'état
LARS	4n	3	Fonctionne bien sur les grands lots (jusqu'à 32k), compense la disparition et l'explosion des gradients, moment intégré.	Le calcul de la norme du gradient pour chaque couche peut être inefficace.

Évaluation des modèles d'apprentissage automatique

Dans le domaine de l'apprentissage automatique, notre objectif est de réaliser des modèles qui généralisent, c'est-à-dire qui fonctionnent bien sur des données jamais vues auparavant, et le surajustement est l'obstacle principal. Nous ne pouvons contrôler que ce que nous pouvons observer, il est donc crucial de pouvoir mesurer de manière fiable le pouvoir de généralisation de notre modèle.

Ensembles d'entrainement, de validation et de test

L'évaluation d'un modèle implique toujours la division des données en trois ensembles : l'ensemble d'apprentissage, l'ensemble de validation et l'ensemble de test. Vous effectuez l'apprentissage sur les données d'entrainement, et vous évaluez votre modèle sur les données de validation. Une fois que votre modèle est prêt à être utilisé, vous le testez une dernière fois sur les données de test.

Ensembles d'entrainement, de validation et de test

le développement d'un modèle implique toujours le réglage de sa configuration, par exemple choisir le nombre de couches ou la taille des couches (ce que l'on appelle les "hyperparamètres" du modèle, pour les distinguer des "paramètres", qui sont les poids du réseau).

Vous effectuerez ce réglage en utilisant comme signal de retour la performance du modèle sur les données de validation, de sorte que ce réglage est en fait une forme d'apprentissage : une recherche d'une bonne configuration dans un certain espace de paramètres.

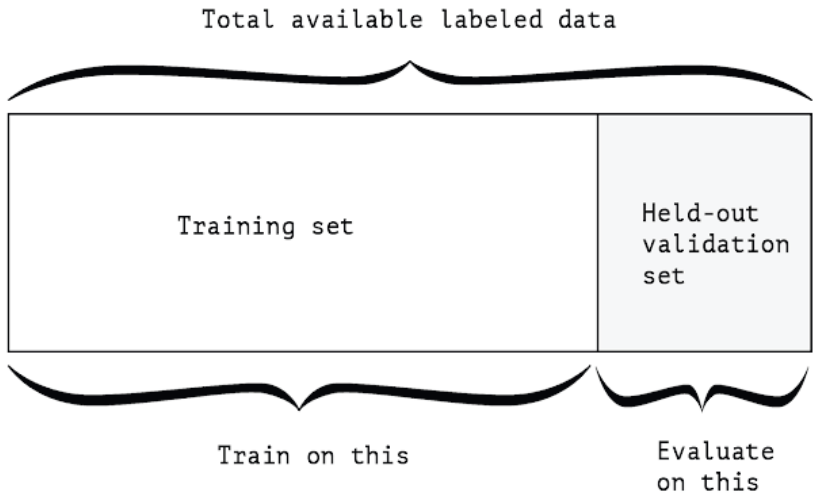
Par conséquent, l'ajustement de la configuration du modèle en fonction de ses performances peut rapidement résulter en un surajustement à l'ensemble de validation, même si votre modèle n'est jamais entraîné directement sur celui-ci.

Ensembles d'entrainement, de validation et de test

Puisque ce qui vous intéresse, ce sont les performances sur des données totalement nouvelles, et non les données de validation, vous avez besoin d'un ensemble de données totalement différent, jamais vu auparavant, pour évaluer votre modèle : l'ensemble de données de test.

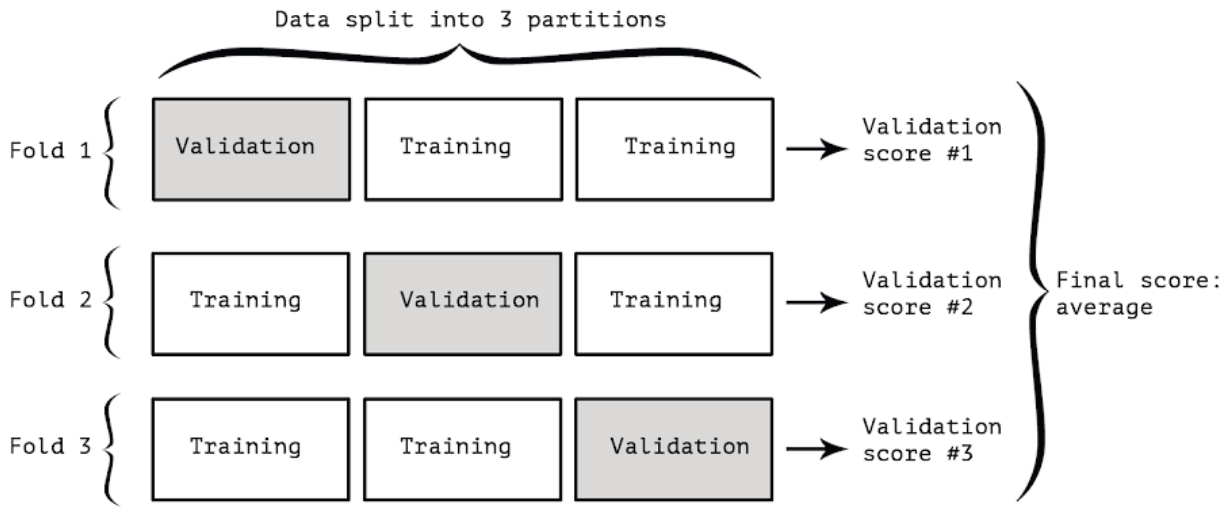
Validation simple
(Simple hold-out validation)

Afin d'éviter les fuites d'informations, vous ne devez pas ajuster votre modèle en fonction de l'ensemble de test, et vous devez donc réserver un ensemble de validation.



Validation par K-fold

Divisez vos données en K partitions de taille égale. Pour chaque partition i, entraînez un modèle sur les N-1 partitions restantes, et évaluez-le sur la partition i. Votre score final sera alors la moyenne des K scores obtenus.



Validation par K-fold itérative avec randomisation

Il s'agit d'appliquer la validation par K-fold plusieurs fois, en mélangeant les données à chaque fois avant de les diviser en K-folds.

Votre score final sera la moyenne des scores obtenus à chaque passage de la validation K-fold. Notez que vous finissez par l'apprentissage et l'évaluation de P * K modèles, où P est le nombre d'itérations que vous utilisez.

Prétraitement des données, ingénierie et apprentissage des caractéristiques

De nombreuses techniques de prétraitement des données et d'ingénierie des caractéristiques sont spécifiques à certains domaines (par exemple, spécifiques aux données textuelles ou aux données d'image). Nous allons les aborder au fur et à mesure que nous les rencontrerons dans des exemples pratiques. Pour l'instant, nous allons rappeler les bases communes à tous les types de données.

Vectorisation

Toutes les entrées et les cibles d'un réseau de neurones doivent être des tenseurs de données numériques (ou, dans certains cas, des tenseurs d'entiers). Quelles que soient les données à traiter (sons, images, textes), il faut d'abord les transformer en tenseurs, cette étape est appelée la "vectorisation des données".

Normalisation des valeurs

En général, il n'est pas judicieux de fournir à un réseau de neurones des données qui prennent des valeurs relativement "grandes" (par exemple, des entiers à plusieurs chiffres, ce qui est beaucoup plus grand que les valeurs initiales prises par les poids d'un réseau), ou des données hétérogènes, où une caractéristique serait dans l'intervalle 0-1 et une autre dans l'intervalle 100-200. Cela peut déclencher de grandes mises à jour du gradient qui empêcheront votre réseau à converger.

Normalisation des valeurs

Pour faciliter l'apprentissage de votre réseau, vos données doivent :

- Prendre de "petites" valeurs : en général, la plupart des valeurs doivent être comprises entre 0 et 1.
- Être homogènes, c'est-à-dire que toutes les caractéristiques doivent prendre des valeurs dans le même intervalle.

Normalisation des valeurs

En plus, la pratique de normalisation plus stricte suivante est commune et pourrait être utile, bien qu'elle ne soit pas toujours nécessaire (par exemple, nous ne l'avons pas fait dans MNIST dataset):

- Normaliser chaque caractéristique indépendamment pour avoir une moyenne de 0.
- Normaliser chaque caractéristique indépendamment pour avoir un écart-type de 1.

Traitement des valeurs manquantes

En général, avec les réseaux de neurones, il est acceptable d'entrer les valeurs manquantes comme étant 0, sous la condition que 0 n'est pas déjà une valeur significative.

Le réseau apprendra, en étant exposé aux données, que la valeur 0 signifie simplement "données manquantes" et commencera à ignorer cette valeur.

Traitement des valeurs manquantes

Cependant, notez que si vous vous attendez à des valeurs manquantes dans les données de test mais que le réseau a été entraîné sur des données sans aucune valeur manquante, alors le réseau n'aura pas appris à ignorer les valeurs manquantes.

Dans ce cas, vous devez générer artificiellement des échantillons d'entraînement avec des entrées manquantes : copiez simplement plusieurs fois certains échantillons d'entraînement et supprimez certaines des caractéristiques qui, selon vous, sont susceptibles de ne pas exister dans les données de test.

Ingénierie des caractéristiques

L'ingénierie des caractéristiques consiste à utiliser vos propres connaissances sur les données et sur l'algorithme d'apprentissage automatique utilisé (dans notre cas, un réseau de neurones) pour améliorer le fonctionnement de l'algorithme en appliquant des transformations aux données avant qu'elles ne soient introduites dans le modèle.

Les données doivent être présentées au modèle de manière à faciliter le travail de ce dernier. Un exemple intuitif est le suivant : supposons que nous essayons de développer un modèle capable de prendre en entrée l'image d'une horloge et d'en sortir l'heure.

Le surajustement et le sous-ajustement

Le problème fondamental de l'apprentissage automatique est la tension entre l'optimisation et la généralisation.

Le but est d'obtenir une bonne généralisation, mais vous ne contrôlez pas la généralisation; vous pouvez seulement ajuster le modèle en fonction de ses données d'entraînement.

Le surajustement et le sous-ajustement

Au début de de l'entraînement, l'optimisation et la généralisation sont corrélées positivement : plus la perte sur les données d'entraînement est faible, plus la perte sur les données de test est faible. Durant cette période, on dit que le modèle est sous-ajusté. Le réseau n'a pas encore modélisé tous les modèles pertinents dans les données d'apprentissage.

Après un certain nombre d'itérations, la généralisation cesse de s'améliorer, les métriques de validation commencent alors à se dégrader, on dit que le modèle commence alors à être surajusté, c'est-à-dire qu'il commence à apprendre des modèles qui sont spécifiques aux données d'apprentissage mais qui ne sont pas pertinents lorsqu'il s'agit de nouvelles données.

Le surajustement et le sous-ajustement

Pour éviter qu'un modèle n'apprenne pas des modèles trompeurs ou non pertinents trouvés dans les données de l'entraînement, la meilleure solution est bien sûr d'obtenir plus de données d'entraînement. Un modèle entraîné sur un grand nombre de données sera naturellement mieux généralisé.

Lorsque cela n'est plus possible, la meilleure solution suivante consiste à moduler la quantité d'informations que votre modèle est autorisé à stocker ou à ajouter des contraintes sur les informations qu'il est autorisé à stocker.

Le traitement permettant de prévenir le surajustement de cette manière s'appelle la régularisation . Nous allons aborder certaines techniques de régularisation parmi les plus courantes.

Réduire la taille du réseau

La façon la plus simple pour éviter le surajustement est de réduire la taille du modèle, c'est-à-dire le nombre de paramètres entrainable dans le modèle ,qui est déterminé par le nombre de couches et le nombre d'unités par couche.

Malheureusement, il n'y a pas de formule spéciale pour déterminer le bon nombre de couches ou la taille correcte de chaque couche. Vous devrez évaluer un ensemble de différentes architectures (sur votre ensemble de validation, et non pas sur votre ensemble de test) afin de trouver la bonne taille de modèle pour vos données.

Réduire la taille du réseau

Le processus général pour trouver une taille de modèle appropriée consiste à commencer avec relativement peu de couches et de paramètres, et à augmenter la taille des couches ou à ajouter de nouvelles jusqu'à ce que vous constatez des rendements décroissants en ce qui concerne la perte de validation.

Régularisation des poids

Une solution fréquente pour atténuer le surajustement est de mettre des contraintes sur la complexité d'un réseau en forçant ses poids à ne prendre que de petites valeurs, ce qui rend la distribution des valeurs de poids plus régulière.

Cela se fait en ajoutant à la fonction de perte du réseau un coût associé à la présence de poids importants. Ce coût se présente sous deux formes :

- Régularisation L1, où le coût ajouté est proportionnel à la valeur absolue des coefficients de poids. (L1 norm)
- Régularisation L2, où le coût ajouté est proportionnel au carré de la valeur des coefficients de poids. (L2 norm ou dégradation du poids)

Régularisation des poids

Dans Keras, la régularisation des poids est ajoutée en passant des instances de régularisation aux couches en tant qu'arguments:

```
from keras import regularizers

model = models.Sequential()
model.add(layers.Dense(6,kernel_regularizer=regularizers.l2(
0.001), activation='relu'))
```

Régularisation L1 et L2 en même temps:

```
regularizers.l1_l2(l1=0.001, l2=0.001)
```

L'ajout du dropout

Le dropout est l'une des techniques de régularisation les plus efficaces utilisées pour les réseaux de neurones. Elle consiste à "abandonner" de manière aléatoire (c'est-à-dire à mettre à zéro) un certain nombre de caractéristiques de sortie de la couche pendant l'apprentissage.

Le taux de dropout est la fraction des caractéristiques qui sont mises à zéro; il est généralement fixé entre 0,2 et 0,5.

Au moment du test, aucune unité n'est éliminée, mais les valeurs de sortie de la couche sont réduites par un facteur égal au taux de dropout, de façon à compenser le fait que plus d'unités sont actives par rapport à l'apprentissage.

L'ajout du dropout

Au moment de l'apprentissage, nous éliminerions au hasard 50% des paramètres de la couche par :

```
model.add(layers.Dense(6, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(6, activation='relu'))
```

Le processus général de l'apprentissage automatique

Ce qui suit est un plan général que vous pouvez utiliser pour aborder et résoudre n'importe quel problème d'apprentissage automatique, en rassemblant les différents concepts que nous avons vus.

Définir le problème et rassembler le jeu de données

Tout d'abord, vous devez définir le problème à résoudre :

- Quelles seront vos données d'entrée ? Que tenterez-vous de prédire ? Vous ne pouvez apprendre quelque chose que si vous avez des données d'entraînement disponibles, par exemple vous ne pouvez apprendre à classer le sentiment des critiques de films que si vous avez à la fois des critiques de films et des annotations de sentiments disponibles.

- Quel est le type de problème auquel vous êtes confronté : s'agit-il d'une classification binaire ? Multi-classes ? Régression scalaire ? Régression vectorielle ? Ou autre chose, comme le clustering, la génération ou l'apprentissage par renforcement ? L'identification du type de problème guidera votre choix d'architecture de modèle, de fonction de perte, et ainsi de suite.

Définir le problème et rassembler le jeu de données

Vous ne pouvez pas passer à l'étape suivante tant que vous ne savez pas quelles sont vos entrées et sorties, et quelles données vous allez utiliser. Soyez conscient des hypothèses que vous formulez à cette étape :

- Vous faites l'hypothèse que vos sorties peuvent être prédites à partir de vos entrées.
- Vous faites l'hypothèse que vos données disponibles sont suffisamment informatives pour apprendre la relation entre les entrées et les sorties.

Jusqu'à ce que vous ayez un modèle fonctionnel, alors ce sont simplement des hypothèses, attendant d'être validées ou invalidées. On ne peut pas résoudre tous les problèmes ; ce n'est pas parce que vous avez rassemblé des éléments d'entrées X et de cibles Y, que X contient assez d'informations pour prédire Y.

Choisir une mesure de réussite

Pour contrôler quelque chose, vous devez être capable de l'observer. Pour réussir, vous devez définir ce que vous entendez par réussite.

Votre mesure du succès guidera le choix de votre fonction de perte, c'est-à-dire le choix de ce que votre modèle optimisera.

il n'est pas rare de définir votre propre métrique personnalisée pour mesurer le succès.

Adopter un protocole d'évaluation

Une fois que vous connaissez votre objectif, vous devez établir comment vous allez mesurer votre progrès actuels. Nous avons précédemment examiné trois protocoles d'évaluation courants :

- Maintien d'un ensemble de validation en réserve ; c'est la solution à adopter lorsque vous disposez de suffisamment de données.
- Effectuer une validation par K-fold ; c'est la solution à adopter lorsque vous disposez de peu d'échantillons pour que la validation simple soit fiable.
- Faire une validation par K-fold itérative ; ceci permet d'effectuer une évaluation très précise du modèle lorsque peu de données sont disponibles

Préparer les données

Une fois que vous savez sur quoi vous allez travailler, ce que vous allez optimiser, et comment évaluer votre approche, vous êtes presque prêt à commencer à entraîner le modèle. Mais d'abord, vous devez mettre en forme vos données de manière à ce qu'elles puissent être introduites dans un modèle d'apprentissage profond:

- Comme nous l'avons vu précédemment, vos données doivent être représentées sous forme de tenseurs.
- Les valeurs prises par ces tenseurs devraient pratiquement toujours être ramenées à des petites valeurs, par exemple dans l'intervalle [0,1] ou [-1,1].
- Si différentes caractéristiques prennent des valeurs dans des intervalles différents (données hétérogènes), les données doivent être normalisées.
- S'il est nécessaire il faut faire de l'ingénierie des caractéristiques.

Une fois que vos tenseurs de données d'entrée et de données cibles sont prêts, vous pouvez commencer à entraîner vos modèles.

Développer un modèle qui dépasse un modèle basique

Votre objectif à ce stade est d'atteindre la "puissance statistique", c'est-à-dire de développer un petit modèle qui est capable de dépasser un modèle aléatoire.

Notez qu'il n'est pas toujours possible d'atteindre la puissance statistique. Si vous ne pouvez pas dépasser un modèle de base aléatoire après avoir essayé plusieurs architectures raisonnables, il se peut que la réponse à la question que vous posez n'est pas réellement présente dans les données d'entrée.

Développer un modèle qui dépasse un modèle basique

En supposant que tout fonctionne correctement, il y a trois choix clés que vous devez faire afin de construire votre modèle:

- Choix de la fonction d'activation de la dernière couche. Cela permet d'établir des contraintes utiles sur la sortie du réseau.
- Choix de la fonction de perte. Elle doit correspondre au type de problème que vous essayez de résoudre.
- Choix de la configuration de l'optimisation : quel optimiseur utiliserez-vous ? Quel sera son taux d'apprentissage ?

Développer un modèle qui dépasse un modèle basique

Ce tableau peut vous aider à choisir une fonction d'activation de la dernière couche et une fonction de perte pour quelques types de problèmes courants :

Type de problème	Activation de la dernière couche	Fonction de perte
Classification binaire	Sigmoid	binary_crossentropy
Classification multi-classes, à un seul label	Softmax	categorical_crossentropy
Classification multi-classes, à multi-labels	sigmoid	binary_crossentropy
Régression vers des valeurs arbitraires	Sans activation	mse
Régression vers des valeurs entre 0 et 1	sigmoid	mse ou binary_crossentropy

développer un modèle avec surajustement

Une fois que vous avez obtenu un modèle ayant une puissance statistique, la question devient : votre modèle est-il assez puissant ? A-t-il suffisamment de couches et de paramètres pour modéliser correctement le problème en question ?

le modèle idéal est celui qui se situe juste à la frontière entre le sous-ajustement et le surajustement, entre la sous-capacité et la surcapacité. Pour déterminer cette frontière, il faut d'abord la dépasser.

Pour déterminer la taille du modèle dont vous aurez besoin, vous devez élaborer un modèle avec surajustement. C'est relativement simple :

- Ajoutez des couches.
- Augmentez la taille de ces couches.
- Faire l'apprentissage sur un nombre plus important d'époques.

développer un modèle avec surajustement

Surveillez toujours la perte de l'apprentissage et la perte de la validation, ainsi que les valeurs de l'apprentissage et de la validation pour toutes les métriques qui vous intéressent. Lorsque vous constatez que les performances du modèle sur les données de validation commencent à se dégrader, vous avez atteint le surajustement.

L'étape suivante consiste à commencer la régularisation de votre modèle, afin de vous rapprocher le plus possible du modèle idéal.

Régulariser le modèle et ajuster les hyperparamètres

C'est la partie qui vous prendra le plus de temps, vous allez modifier votre modèle à plusieurs reprises, l'entraîner, l'évaluer sur vos données de validation, le modifier à nouveau jusqu'à ce que votre modèle soit optimal.

Vous pouvez essayer ces méthodes :

- Ajoutez du dropout.
- Essayez différentes architectures, ajoutez ou supprimez des couches.
- Ajoutez une régularisation L1 ou L2.
- Essayez différents hyperparamètres (tels que le nombre d'unités par couche, le taux d'apprentissage de l'optimiseur) pour trouver la configuration optimale.
- Optionnellement, expérimentez avec l'ingénierie des caractéristiques, ajouter de nouvelles caractéristiques, supprimer celles qui ne semblent pas être informatives.

Régulariser le modèle et ajuster les hyperparamètres

Une fois que vous avez développé une configuration de modèle suffisamment performante, vous pouvez entraîner votre modèle final sur toutes les données disponibles (apprentissage et validation) et l'évaluer une dernière fois sur l'ensemble de test.

S'il apparaît que la performance sur l'ensemble de test est significativement plus faible que la performance mesurée sur les données de validation, cela peut signifier soit que votre procédure de validation n'était pas si fiable, soit que vous avez commencé à sur-ajuster aux données de validation tout en ajustant les paramètres du modèle.

Dans ce cas, vous pouvez passer à un protocole d'évaluation plus fiable (par exemple, la validation par K-fold itérative).