Agence Nationale de Statistique et de la Démographie

**ANSD**

---

Ecole Nationale de la Statistique et de l'Analyse Economique

**ENSAE Pierre NDIAYE**

---

# LES METHODES DE DEEP LEARNING AVEC R

---

Rédigé par :
Yatoute MINTOAMA
Elève ingénieur statisticien
économiste

Sous la supervision de :
Mouhamadou Hady DIALLO
Ingénieur statisticien, Data Scientist

25 avril 2022

# Contents

# Introduction

Le deep learning ou l'apprentissage en profondeur en français, est un sous-domaine spécifique du machine learning : une nouvelle approche de l'apprentissage de représentations à partir de données qui met l'accent sur l'apprentissage de couches successives de des représentations de plus en plus signifiantes. Le deep in deep learning ne fait référence à aucune sorte de compréhension plus profonde atteinte par l'approche ; il représente plutôt cette idée de couches successives de représentations. Le nombre de couches qui contribuent à un modèle de données est appelé la profondeur du modèle. D'autres noms appropriés pour le domaine auraient pu être l'apprentissage des représentations en couches et l'apprentissage des représentations hiérarchiques. L'apprentissage en profondeur moderne implique souvent des dizaines, voire des centaines de couches successives de représentations, et elles sont toutes apprises automatiquement à partir de l'exposition aux données de formation. Pendant ce temps, d'autres approches de l'apprentissage automatique ont tendance à se concentrer sur l'apprentissage d'une ou deux couches de représentations des données; par conséquent, elles sont parfois appelées apprentissage superficiel.

Dans l'apprentissage en profondeur, ces représentations en couches sont (presque toujours) apprises via des modèles appelés réseaux de neurones, structurés en couches littérales empilées les unes sur les autres. Le terme réseau de neurones fait référence à la neurobiologie, mais bien que certains des concepts centraux de l'apprentissage en profondeur aient été développés en partie en s'inspirant de notre compréhension du cerveau, les modèles d'apprentissage en profondeur ne sont pas des modèles du cerveau. Il n'y a aucune preuve que le cerveau implémente quelque chose comme les mécanismes d'apprentissage utilisés dans les modèles modernes d'apprentissage en profondeur. Vous pouvez rencontrer des articles de popscience proclamant que l'apprentissage en profondeur fonctionne comme le cerveau ou a été calqué sur le cerveau, mais ce n'est pas le cas. Il serait déroutant et contre-productif pour les nouveaux venus dans le domaine de penser que l'apprentissage en profondeur est lié de quelque manière que ce soit à la neurobiologie; vous n'avez pas besoin de ce linceul de mystique et de mystère "tout comme nos esprits", et vous pouvez tout aussi bien oublier quoi que ce soit vous avez peut-être lu des liens hypothétiques entre l'apprentissage en profondeur

et la biologie. Pour nos besoins, l'apprentissage en profondeur est un cadre mathématique pour apprendre des représentations à partir de données.

# I. Comprendre le fonctionnement du deep learning

Le deep learning consiste à mapper des entrées (telles que des images) à des cibles (telles que l'étiquette "chat"), ce qui se fait en observant de nombreux exemples d'entrées et de cibles. cette cartographie « input-to-target » est effectué par les réseaux de neurones profonds via une séquence profonde de transformations de données simples et que ces transformations de données sont apprises par exposition à des exemples. chaque couche neuronale transforme ses données d'entrée comme suit :

```
output = activate(dot(W, input) + b)
```

Dans cette expression, W et b sont des tenseurs qui sont des attributs de la couche. On les appelle les pondérations ou les paramètres entraînables de la couche (respectivement les attributs de noyau et de biais). Ces pondérations contiennent les informations apprises par le réseau à partir de l'exposition aux données d'entraînement. Bien sûr, il n'y a aucune raison de s'attendre à ce que activate(dot(W, input) + b), lorsque W et b sont aléatoires, produise des représentations utiles. Les représentations qui en résultent n'ont aucun sens, mais elles constituent un point de départ. Il s'agit ensuite d'ajuster progressivement ces pondérations, en fonction d'un signal de rétroaction. Cet ajustement progressif, également appelé formation, est essentiellement l'apprentissage qu'est l'apprentissage automatique.

Cela se produit dans ce qu'on appelle une boucle de formation, qui fonctionne comme suit. Répétez ces étapes en boucle, aussi longtemps que nécessaire :

1. Dessinez un lot d'échantillons d'apprentissage x et de cibles correspondantes y;
2. Exécutez le réseau sur x (une étape appelée la passe avant) pour obtenir des prédictions y_pred ;
3. Calculez la perte du réseau sur le lot, une mesure de l'inadéquation entre y_pred et y;
4. Mettez à jour tous les poids du réseau de manière à réduire légèrement la perte sur ce lot.

   Dans ce contexte, l'apprentissage signifie trouver un ensemble de valeurs pour les pondérations de toutes les couches d'un réseau, de sorte que le réseau mappe correctement les exemples d'entrées sur leurs cibles associées. La figure ci-dessous illustre un peu ce que nous venons de décrire.

Pour contrôler quelque chose, il faut d'abord être capable de l'observer. Pour contrôler la sortie d'un réseau de neurones, vous devez être en mesure de mesurer à quelle distance cette sortie est de ce que vous attendiez. C'est le travail de la fonction de perte du réseau,également appelée fonction objectif. **La fonction de perte(loss)** prend les prédictions du réseau et la véritable cible (ce que vous vouliez que le réseau produise) et calcule un score de distance, capturant la performance du réseau sur cet exemple spécifique. Cet score est fondamentalement utilisé comme un signal de rétroaction pour ajuster un peu la valeur des

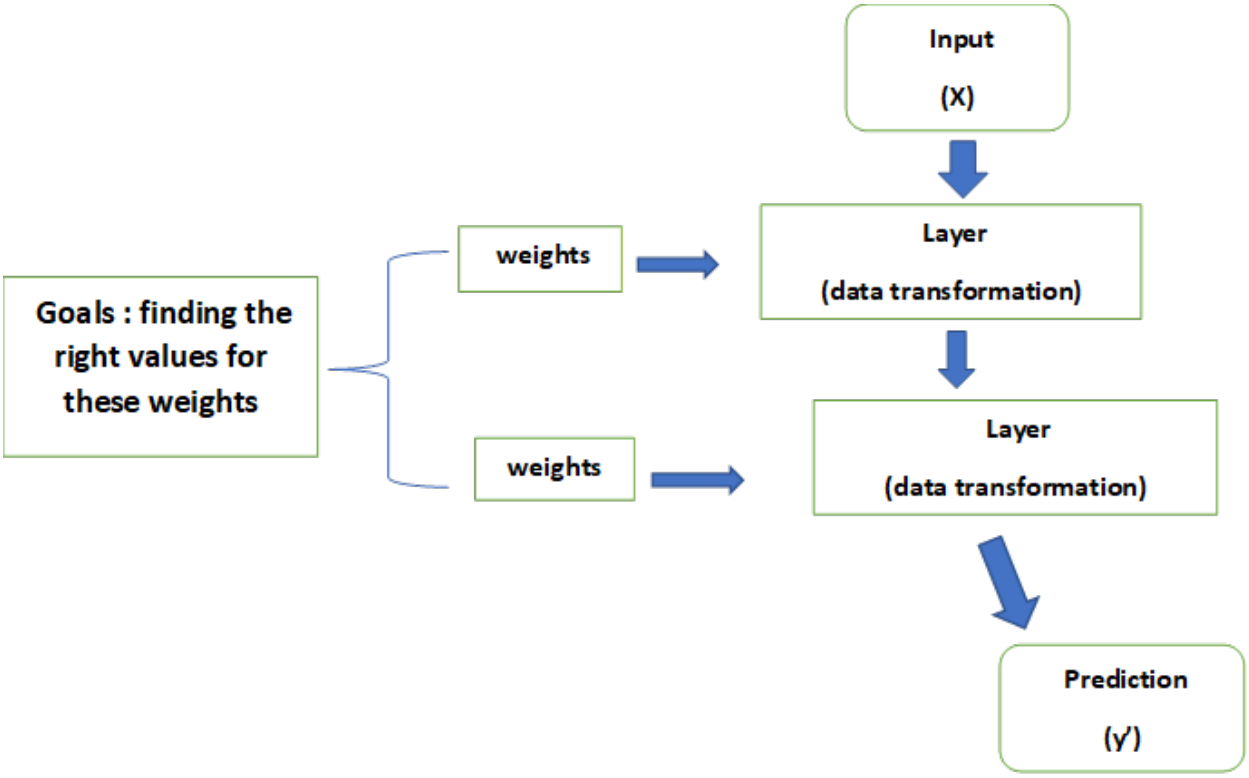


Figure 1: Paramétrage des poids d'un réseau de neurone

poids, dans une direction qui abaissera le score de perte. Cet ajustement est le travail de **l'optimiseur(optimizer)**.

Nous pouvons retenir de cette partie que :

- **Apprendre signifie trouver une combinaison de paramètres de modèle qui minimise une fonction de perte pour un ensemble donné d'échantillons de données d'apprentissage et leurs cibles correspondantes.**
- **L'apprentissage se produit en tirant des lots aléatoires d'échantillons de données et de leurs cibles, et en calculant le gradient des paramètres du réseau par rapport à la perte sur le lot. Les paramètres du réseau sont alors pour ajusté dans une direction qui abaissera le score de perte.**
- **Les deux concepts clé que nous verrons fréquemment dans les prochaines parties et qui permettent d'ajuster les paramètres du réseau sont : la perte(loss) et les optimiseurs(optimizer).**
- **La perte est la quantité que nous tenterons de de minimiser pendant l'entraînement, elle devrait donc représenter une mesure de réussite pour la tâche que vous essayez de résoudre.**

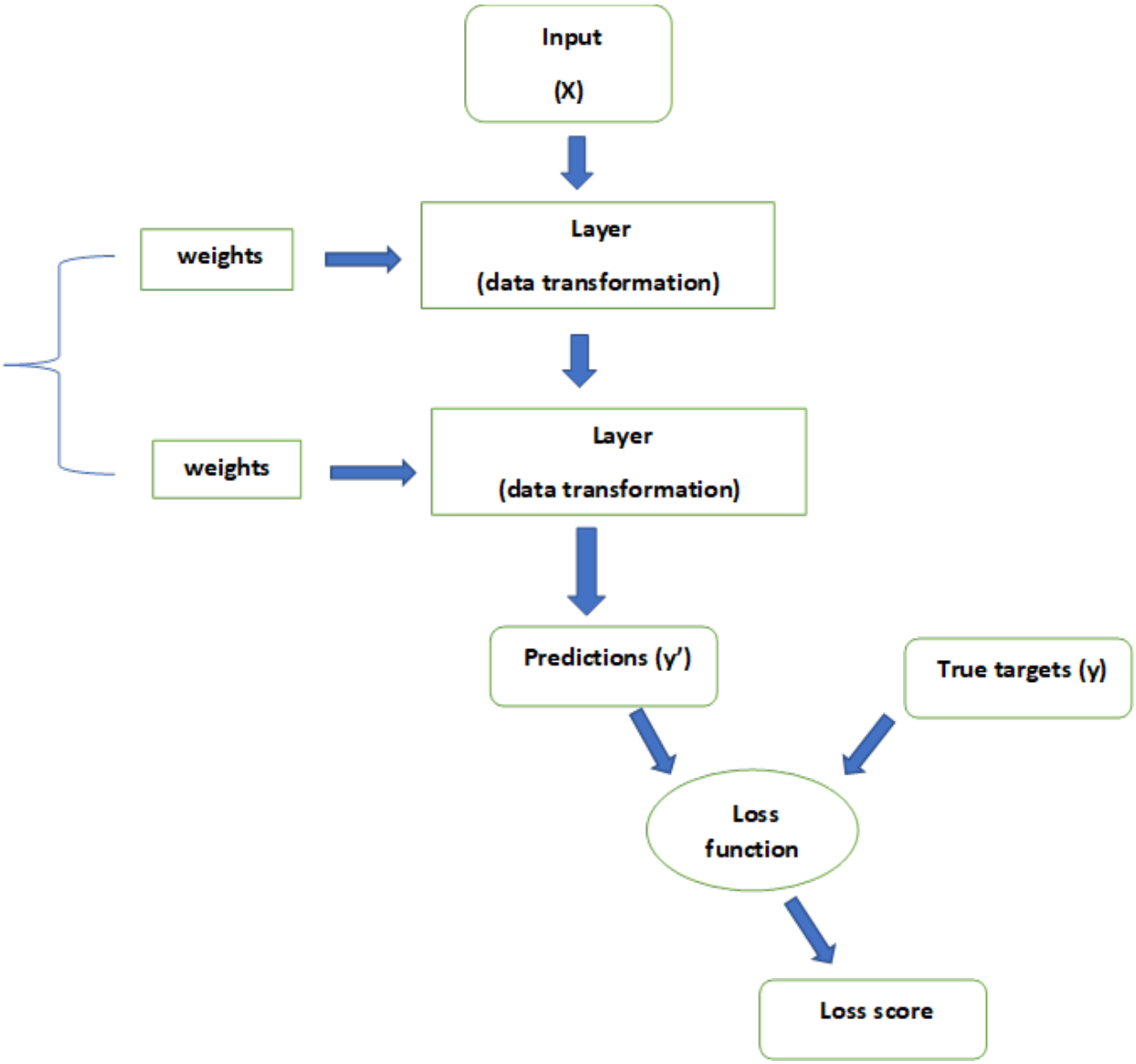


Figure 2: Une fonction de perte mesure la qualité de la sortie du réseau

- **L'optimiseur spécifie la manière exacte dont le gradient de la perte sera utilisé pour mettre à jour les paramètres: par exemple, il peut s'agir de l'optimiseur RMSProp, SGD avec momentum, etc.**

# II. Les réseaux de neurones

Dans ce chapitre, nous examinerons de plus près les composants de base des réseaux de neurones que nous avons présentés au chapitre 1 : les couches, les réseaux, les fonctions objectives et les optimiseurs.

Ensuite nous présenterons rapidement Keras, la bibliothèque d'apprentissage en profondeur que nous utiliserons tout au long de cet exposé. Nous allons configurer un poste de travail d'apprentissage en profondeur avec prise en charge de TensorFlow et Keras.

En fin nous allons plonger dans trois exemples d'introduction sur la façon d'utiliser les réseaux de neurones pour résoudre des problèmes réels :

- Classer les critiques de films comme positives ou négatives (classification binaire) ;
- Classement des dépêches par sujet (classification multiclasse) ;
- Estimation du prix d'une maison, compte tenu des données immobilières (régression)

À la fin de ce chapitre, vous serez en mesure d'utiliser les réseaux de neurones pour résoudre des problèmes simples tels que la classification et la régression sur des données vectorielles. Vous serez alors prêt à commencer à construire une compréhension plus théorique et fondée sur des principes de l'apprentissage en profondeur au chapitre 3.

## 1. Anatomie d'un réseau de neurone

L'apprentissage d'un réseau de neurones s'articule autour des objets suivants :

- les couches, qui sont combinées dans un réseau (ou modèle) ;
- les données d'entrée et les cibles correspondantes ;
- la fonction de perte, qui définit le signal de rétroaction utilisé pour l'apprentissage ;
- l'optimiseur, qui détermine le déroulement de l'apprentissage.

Nous pouvez visualiser leur interaction comme illustré à la figure ci-dessous : le réseau, composé de couches enchaînées, mappe les données d'entrée aux prédictions. La fonction de perte compare ensuite ces prédictions aux cibles, produisant une valeur de perte : une mesure de la mesure dans laquelle les prédictions du réseau correspondent à ce qui était attendu. L'optimiseur utilise cette valeur de perte pour mettre à jour les pondérations du réseau.

Examinons de plus près les couches, les réseaux, les fonctions de perte et les optimiseurs.

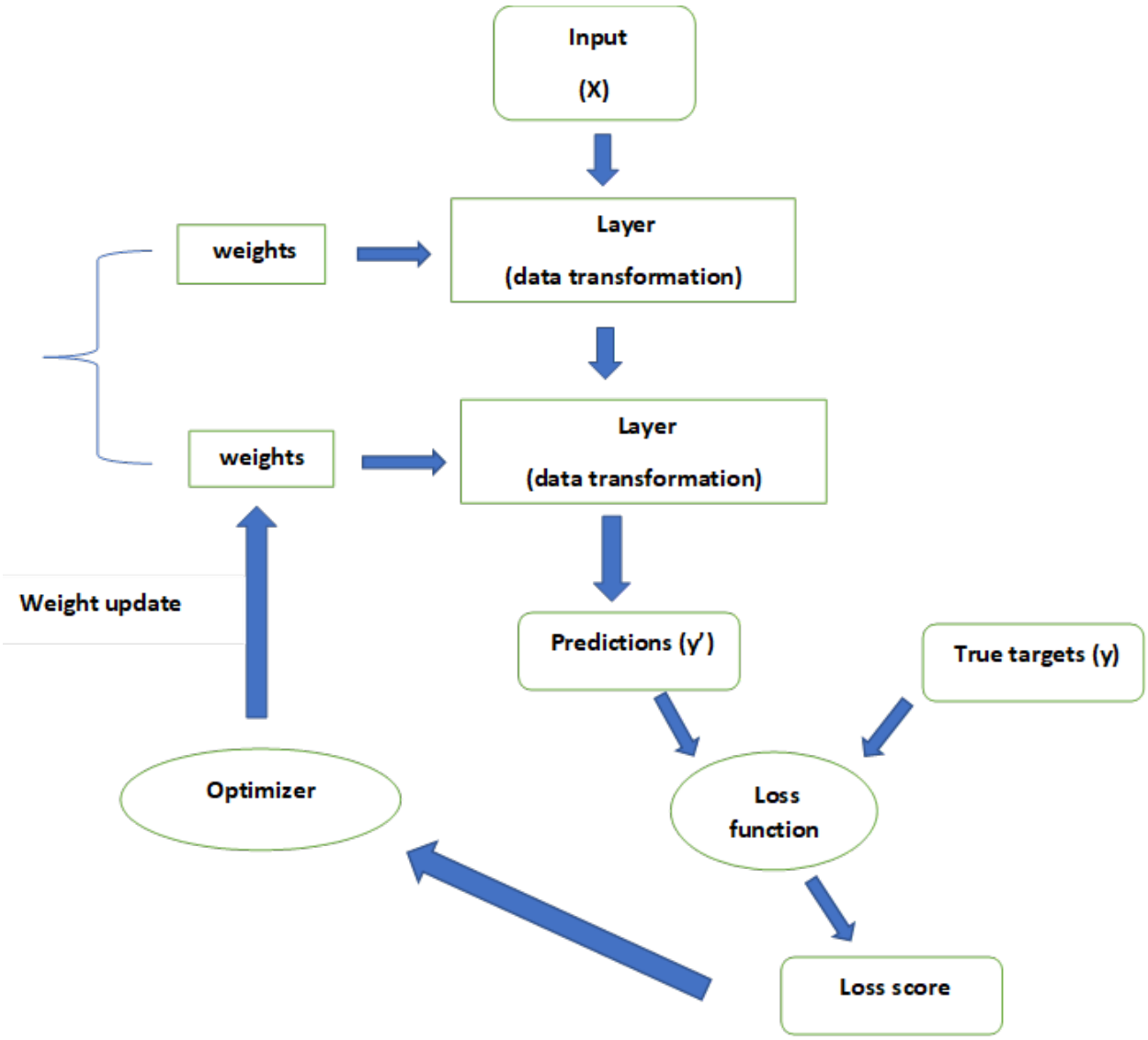


Figure 3: Relation entre le réseau, les couches, la fonction de perte et l'optimiseur

## 1.1. Les couches neuronales

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

Différentes couches sont appropriées pour différents formats de tenseurs et différents types de traitement de données. Par exemple, des données vectorielles simples, stockées dans des tenseurs de forme 2D (échantillons, caractéristiques), sont souvent traitées par des couches densément connectées, également appelées couches entièrement connectées ou denses (la fonction layer_dense dans Keras). Les données de séquence, stockées dans des tenseurs de forme 3D (échantillons, pas de temps, caractéristiques), sont généralement traitées par des couches récurrentes telles que layer_lstm. Les données d'image, stockées dans des tenseurs 4D, sont généralement traitées par des couches de convolution 2D (layer_conv_2d).

La création de modèles d'apprentissage en profondeur dans Keras se fait en associant des couches compatibles pour former des pipelines de transformation de données utiles.

Exemple : *layer = layer_dense(units = 32, input_shape = c(784))*

Dans l'exemple ci-dessus, nous avons créé une couche qui n'acceptera comme entrée que les tenseurs 2D où la première dimension est 784 (la dimension du lot, n'est pas spécifiée, et donc toute valeur serait acceptée). Cette couche renverra un tenseur où la première dimension a été transformée en 32.

## 1.2. Les modèles : lesréseaux de couches

Un modèle d'apprentissage en profondeur est un graphe orienté et acyclique de couches. L'instance la plus courante est une pile linéaire de couches, mappant une seule entrée à une seule sortie.

Choisir la bonne architecture de réseau est plus un art qu'une science; et bien qu'il existe des pratiques exemplaires et des principes sur lesquels vous pouvez compter, seule la pratique peut vous aider à devenir un véritable architecte de réseau neuronal.

## 1.3. Fonctions de perte et optimiseurs : clés pour configurer le processus d'apprentissage

Une fois l'architecture réseau définie, il reste encore deux choses à choisir :

- Fonction de perte (fonction objectif) : la quantité qui sera minimisée pendant la formation. Il représente une mesure de succès pour la tâche à accomplir.
- Optimiseur : détermine comment le réseau sera mis à jour en fonction de la fonction de perte. Il implémente une variante spécifique de descente de gradient stochastique (SGD).

Un réseau neuronal qui a plusieurs sorties peut avoir plusieurs fonctions de perte (une par sortie). Mais le processus de descente de gradient doit être basé sur une seule valeur de perte scalaire; ainsi, pour les réseaux multipertes, toutes les pertes sont combinées (via la moyenne) en une seule quantité scalaire.

**NB : Choisir la bonne fonction objectif pour le bon problème est extrêmement important : votre réseau prendra tous les raccourcis possibles pour minimiser la perte;donc si l'objectif n'est pas entièrement corrélé avec le succès de la tâche à accomplir, votre réseau finira par faire des choses que vous n'auriez peut-être pas voulues.**

Heureusement, lorsqu'il s'agit de problèmes courants tels que la classification, la régression et la prédiction de séquence, il existe des directives simples que vous pouvez suivre pour choisir la bonne perte. Par exemple, vous utiliserez :

- l'entropie croisée binaire pour un problème de classification à deux classes ;
- l'entropie croisée catégorielle pour un problème de classification à plusieurs classes ;
- l'erreur quadratique moyenne pour un problème de régression ;
- la classification temporelle connexionniste (CTC) pour un problème d'apprentissage de séquence, etc.

Ce n'est que lorsque vous travaillez sur des problèmes de recherche véritablement nouveaux que vous devrez développer vos propres fonctions objectives.

## 2. Introduction au keras

Keras est un cadre d'apprentissage en profondeur qui offre un moyen pratique de définir et de former presque tous les types de modèles d'apprentissage en profondeur. Il a été initialement développé pour les chercheurs, dans le but de permettre une expérimentation rapide.

Keras possède les fonctionnalités clés suivantes :

- il permet au même code de s'exécuter de manière transparente sur le CPU ou le GPU ;
- il dispose d'une API conviviale qui facilite le prototypage rapide du deep learning ;
- il prend en charge les réseaux convolutifs (pour la vision par ordinateur que nous verrons au chapitre 3), les réseaux récurrents (pour le traitement des séquences) et toute combinaison des deux ;
- il prend en charge des architectures réseau arbitraires: modèles multi-entrées ou multi-sorties,partage de couches, partage de modèles, etc. Cela signifie que Keras est approprié pour construire essentiellement n'importe quel modèle d'apprentissage en profondeur.

Keras et son interface R sont distribués sous la licence permissive MIT, ce qui signifie qu'ils peuvent être librement utilisés dans des projets commerciaux. Le package Keras R est compatible avec les versions R 3.2 et supérieures. La documentation de l'interface R est disponible sur **https://keras.rstudio.com**. Le site Web principal du projet Keras se trouve à l'adresse **https://keras.io**.

## 2.1. Mise en place d'un poste de travail de deep learnin

Keras est une bibliothèque de niveau modèle, fournissant des blocs de construction de haut niveau pour développer des modèles d'apprentissage en profondeur. Il ne gère pas les opérations de bas niveau telles que la manipulation et la différenciation des tenseurs. Au lieu de cela, il s'appuie sur une bibliothèque de tenseurs spécialisée et bien optimisée pour ce faire, servant de moteur principal de Keras. Plutôt que de choisir une seule bibliothèque de tenseurs et de lier l'implémentation de Keras à cette bibliothèque, Keras gère le problème de manière modulaire ; ainsi, plusieurs moteurs différents peuvent être connectés de manière transparente à Keras. Actuellement, les trois implémentations existantes sont TensorFlow, Theano et Microsoft Cognitive Toolkit (CNTK).

Avant de pouvoir commencer à développer des applications d'apprentissage en profondeur, vous devez configurer votre poste de travail. Il est fortement recommandé, bien que pas strictement nécessaire, d'exécuter du code d'apprentissage en profondeur sur un GPU NVIDIA moderne. Certaines applications, en particulier le traitement d'images avec des réseaux convolutifs et le traitement de séquences avec des réseaux de neurones récurrents, seront atrocement lentes sur le processeur, même un processeur multicoeur rapide.

Que vous exécutiez localement ou dans le cloud, il est préférable d'utiliser un poste de travail Unix. Bien qu'il soit techniquement possible d'utiliser Keras sous Windows (les trois backends Keras prennent en charge Windows), nous ne le recommandons pas. Si vous êtes un utilisateur Windows, la solution la plus simple pour que tout fonctionne est de configurer un double démarrage Ubuntu sur votre machine. Cela peut sembler fastidieux, mais l'utilisation d'Ubuntu vous fera économiser beaucoup de temps et de problèmes à long terme.

## 2.2. Installation de Keras

Pour démarrer avec Keras, vous devez installer le package **keras R**, la bibliothèque principale de Keras et un moteur de tenseur backend (tel que TensorFlow). Pour une installation plus complète incluant des dépendances facultatives supplémentaires, il faut installer les packages tensorflow python (install_tensorflow()) et keras python (install_keras()).

Vous pouvez le faire comme suit :

```
require(devtools)
install_github("rstudio/reticulate",force =TRUE)
install_github("rstudio/tensorflow", force = TRUE)
install_github("rstudio/keras",force = TRUE)
```

```
library('reticulate')
library('tensorflow')
library('keras')
install_tensorflow()
install_keras()
```

Si vous souhaitez entraîner vos modèles de deep learning sur un GPU,vous pouvez installer la version basée sur GPU du moteur backend TensorFlow comme suit (si vous utilisez, bien sûr, un système avec un GPU NVIDIA et des bibliothèques CUDA et cuDNN correctement configurées.) :

```
install_keras(tensorflow = "gpu")
```

## 2.3. Développer avec Keras : un aperçu rapide

Le flux de travail typique de Keras se présente comme suit :

1. Définir les données d'entraînementÿ: tenseurs d'entrée et tenseurs cibles.
2. Configurer le processus d'apprentissage en choisissant une fonction de perte, un optimiseur et quelques indicateurs à surveiller.
3. Itérez sur les données d'apprentissage en appelant la méthode *fit()* au modèle.

- **Architecture d'un modèle de réseau de neurones** : Il existe deux manières de définir un modèle: en utilisant la fonction *keras_model_sequential()* (uniquement pour les piles linéaires de couches, qui est de loin l'architecture de réseau la plus courante) ou *l'API fonctionnelle* (pour les graphes acycliques dirigés de couches, qui vous permettent de construire complètement architectures arbitraires).

voici un modèle à deux couches défini à l'aide de *keras_model_sequential* (notez que nous transmettons la forme attendue des données d'entrée à la première couche) :

```
model <- keras_model_sequential() %>%
    layer_dense(units = 32, input_shape = c(784)) %>%
    layer_dense(units = 10, activation = "softmax")
```

Et voici le même modèle défini à l'aide de l'API fonctionnelle :

```
input_tensor <- layer_input(shape = c(784))

output_tensor <- input_tensor %>%
          layer_dense(units = 32, activation = "relu") %>%
          layer_dense(units = 10, activation = "softmax")
model <- keras_model(inputs = input_tensor, outputs = output_tensor)
```

Une fois l'architecture de votre modèle définie, peu importe si vous avez utilisé *keras_model_sequential* ou *l'API fonctionnelle*, toutes les étapes suivantes sont les même.

- **Etape de compilation** : Le processus d'apprentissage est configuré à l'étape de compilation, où vous spécifiez la ou les fonctions d'optimisation et de perte que le modèle doit utiliser, ainsi que les métriques que vous souhaitez surveiller pendant la formation. Voici un exemple avec une seule fonction de perte, qui est de loin le cas le plus courant.

```
model %>% compile(optimizer = optimizer_rmsprop(lr = 0.0001),loss = "mse",metrics = c("a
```

- **Processus d'apprentissage** : En fin, le processus d'apprentissage consiste à transmettre des tableaux de données d'entrée (et les données cibles correspondantes) au modèle via la méthode *fit()*, similaire à ce que vous feriez avec d'autres bibliothèques d'apprentissage automatique.

```
model %>% fit(input_tensor, target_tensor, batch_size = 128, epochs = 10)
```

# 3. Utilisation des réseaux de neurones pour résoudre des problèmes réels

## 3.1. Classification des critiques de cinéma : Une classification binaire

La classification à deux classes, ou classification binaire, peut être le type de problème d'apprentissage automatique le plus largement appliqué. Dans cet exemple, vous apprendrez à classer les critiques de films comme positives ou négatives, en fonction du contenu textuel des critiques.

### 3.1.1. Le jeu de données IMDB

Nous travaillerons avec l'ensemble de données IMDB : un ensemble de 50 000 critiques hautement polarisées de la base de données de films Internet. Ils sont divisés en 25 000 avis pour la formation et 25 000 avis pour les tests, chaque ensemble comprenant 50 % d'avis négatifs et 50 % d'avis positifs.

```
library(keras)
dataset_imdb(num_words = 10000)
imdb = dataset_imdb(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-%imdb
```

Les ensembles de données intégrés à Keras sont tous des listes imbriquées de données d'entraînement et de test. Ici, nous utilisons l'opérateur de multiaffectation *(%<-%)* du package *zeallot* pour décompresser la liste en un ensemble de variables distinctes. L'opérateur *%<-%* est automatiquement disponible chaque fois que le package R *Keras* est chargé.

Pour le plaisir voici comment décoder cette liste de mots :

```
library(keras)
dataset_imdb_word_index()
word_index = dataset_imdb_word_index()
reverse_word_index = names(word_index)
names(reverse_word_index) = word_index
decoded_review = sapply(train_data[[1]], function(index) {
    word = if (index >= 3) reverse_word_index[[as.character(index - 3)]]
    if (!is.null(word)) word else "?"
  })
```

### 3.1.2. Préparation des données

Vous ne pouvez pas alimenter des listes d'entiers dans un réseau de neurones. Vous devez transformer vos listes (les données sont de le foreme (list1,list2,list3,....) en tenseurs. Il y a deux façons de le faire :

1. Complétez vos listes pour qu'elles aient toutes la même longueur, transformez les en un tenseur entier de forme (samples, word_indices), puis utilisez comme première couche de votre réseau une couche capable de gérer de tels tenseurs entiers.

2. One-hot encode vos listes pour les transformer en vecteurs de 0 et de 1. Cela signifierait, par exemple, transformer la séquence [3, 5] en un vecteur à 10 000 dimensions qui serait tous des 0 sauf pour les indices 3 et 5, qui seraient des 1. Ensuite, vous pouvez utiliser comme première couche de votre réseau une couche dense, capable de gérer des données vectorielles à virgule flottante.

Partons avec cette dernière solution pour vectoriser les données, ce que vous ferez manuellement pour un maximum de clarté :

```
vectorize_sequences = function(sequences, dimension = 10000) {
  results = matrix(0, nrow = length(sequences),
                   ncol = dimension)
  for (i in 1:length(sequences))
    results[i, sequences[[i]]] = 1
  return(results)
}

# Recodage des données
```

```
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)

# Convertion des étiquettes d'entier en numérique.
y_train = as.numeric(train_labels)
y_test = as.numeric(test_labels)
```

Maintenant, les données sont prêtes à être introduites dans un réseau de neurones.

### 3.1.3. Construire du réseau de neurone

Les données d'entrée sont des vecteurs et les étiquettes sont des scalaires (1 et 0): c'est la configuration la plus simple que vous puissiez rencontrer. Un type de réseau qui fonctionne bien sur un tel problème est une simple pile de couches entièrement connectées (dense) avec des activations relu: layer_dense(units = 16, activation = "relu").

L'argument units passé à chaque couche dense (16) est le nombre d'unités cachées de la couche. Une unité cachée est une dimension dans l'espace de représentation de la couche. Chacune de ces couches denses avec une activation relu implémente la chaîne suivante d'opérations tensorielles :

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

Avoir 16 unités cachées signifie que la matrice de poids W aura une forme(16, input_dimension): le produit scalaire avec W projettera les données d'entrée sur un espace de représentation à 16 dimensions (puis vous ajouterez le vecteur de biais b et appliquerez l'opération relu) . Vous pouvez intuitivement comprendre la dimensionnalité de votre espace de représentation comme "la liberté que vous accordez au réseau lors de l'apprentissage des représentations internes".

Avoir plus d'unités cachées (un espace de représentation de plus grande dimension) permet à votre réseau d'apprendre des représentations plus complexes, mais cela rend le réseau plus coûteux en calcul et peut conduire à l'apprentissage de modèles indésirables (modèles qui amélioreront les performances sur les données d'apprentissage mais pas sur les données de test) .

Deux décisions d'architecture clés doivent être prises concernant un tel empilement de couches denses :

- Combien de couches utiliser ?
- Combien d'unités cachées choisir pour chaque couche ?

Pour l'instant, vous devrez nous confier le choix d'architecture suivant :

- Deux couches intermédiaires avec 16 unités cachées chacune ;

- Une troisième couche qui produira la prédiction scalaire concernant le sentiment du examen en cours.

```
library(keras)
model <- keras_model_sequential() %>%
 layer_dense(units = 16, activation = "relu",
             input_shape = c(10000)) %>%
 layer_dense(units = 16, activation = "relu") %>%
 layer_dense(units = 1, activation = "sigmoid")
```

Les couches intermédiaires utiliseront **relu** comme fonction d'activation, et la couche finale utilisera une activation **sigmoïde** afin de produire une probabilité (un score compris entre 0 et 1, indiquant la probabilité que l'échantillon ait la cible "1": quelle est la probabilité l'avis doit être positif). Un **relu** (unité linéaire rectifiée) est une fonction destinée à mettre à zéro les valeurs négatives (voir figure 3.4), tandis qu'un **sigmoïde** écrase des valeurs arbitraires dans l'intervalle [0, 1], produisant quelque chose qui peut être interprété comme une probabilité.

Sans une fonction d'activation telle que relu (également appelée non-linéarité), la couche dense consisterait en deux opérations linéaires, un produit scalaire et une addition : *output = dot(W, input) + b*

Ainsi, la couche ne pourrait apprendre que des transformations linéaires (transformations affines) des données d'entrée : l'espace d'hypothèse de la couche serait l'ensemble de toutes les transformations linéaires possibles des données d'entrée dans un espace à 16 dimensions. Un tel espace d'hypothèses est trop restreint et ne bénéficierait pas de plusieurs couches de représentations, car une pile profonde de couches linéaires implémenterait toujours une opération linéaire : l'ajout de couches supplémentaires n'étendrait pas l'espace d'hypothèses.

### 3.1.4. Configuration du modèle avec l'optimiseur rmsprop et la fonction de perte *binary_crossentropy*

Afin d'accéder à un espace d'hypothèses beaucoup plus riche qui bénéficierait de représentations profondes, vous avez besoin d'une fonction de non-linéarité ou d'activation. relu est la fonction d'activation la plus populaire dans l'apprentissage en profondeur, mais il existe de nombreux autres candidats, qui portent tous des noms tout aussi étranges : **prelu, elu, etc**.

Enfin, vous devez choisir une **fonction de perte** et **un optimiseur**. Parce que vous êtes confronté à un problème de classification binaire et que la sortie de votre réseau est une probabilité (vous terminez votre réseau avec une couche unitaire avec une activation **sigmoïde**), il est préférable d'utiliser la perte binary_crossentropy. Ce n'est pas le seul choix viable : vous pouvez utiliser, par exemple, **mean_squared_error**. Mais **l'entropie croisée** est généralement le meilleur choix lorsque vous avez affaire à des modèles qui génèrent des probabilités. La **crossentropie** est une quantité du domaine de la théorie de l'information qui mesure la distance entre les distributions de probabilité ou, dans ce cas, entre la distribution de la vérité terrain et vos prédictions.

```
model %>% compile(optimizer = "rmsprop",
      loss = "binary_crossentropy",
      metrics = c("accuracy"))
```

On peut également utiliser des fonctions de pertes et de métriques personnalisées :

```
model %>% compile(
  optimizer = optimizer_rmsprop(lr = 0.001),
    loss = loss_binary_crossentropy,
     metrics = metric_binary_accuracy)
```

### 3.1.5. Echantillon de validation du modèle

Afin de surveiller pendant la formation la précision du modèle sur des données qu'il n'a jamais vues auparavant, nous allons créer un ensemble de validation en séparant un sous ensemble d'échantillons des données de formation d'origine.

```
val_indices = 1:10000
x_val = x_train[val_indices,]
partial_x_train = x_train[-val_indices,]
y_val = y_train[val_indices]
partial_y_train = y_train[-val_indices]
```

### 3.1.6. Formation de notre modèle

Nous allons maintenant entraîner le modèle pour 20 epochs (20 itérations sur tous les échantillons des tenseurs x_train et y_train), en mini-lots de 512 échantillons. En même temps, nous surveillerons la perte et la précision sur l'échantillon de validations. Pour ce faire, passons les données de validation en tant qu'argument validation_data.

```
history = model %>% fit(partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val))
plot(history)
```

L'objet history comprend des paramètres utilisés pour ajuster le modèle (history$params)$ainsiquedesdonne

Il contient également a une méthode plot() qui vous permet de visualiser la formation et métriques de validation par epochs :

```
plot(history)
```

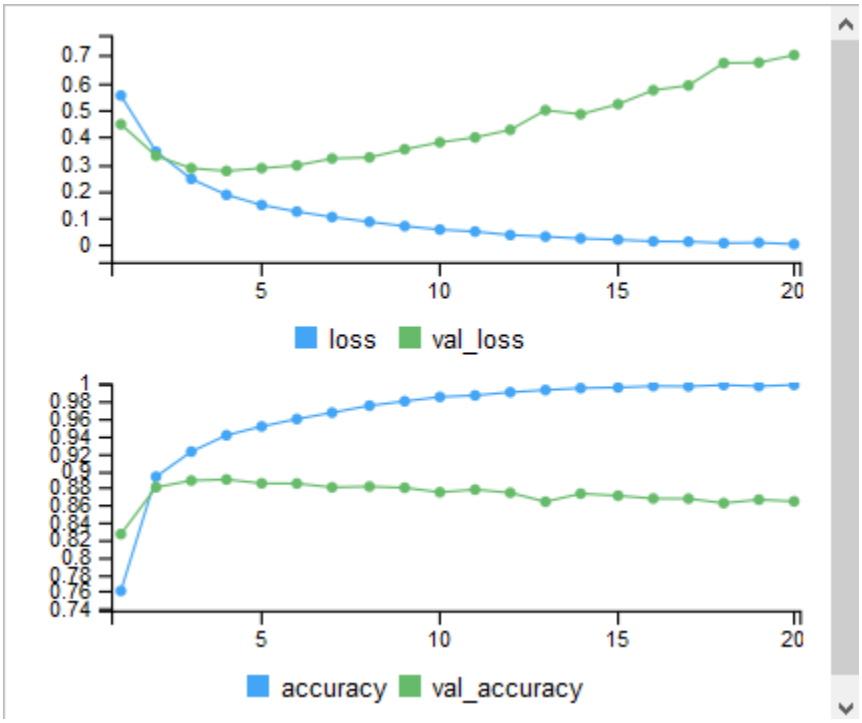


Figure 4: Performances du modèle par époque

Comme vous pouvez le constater, la perte d'entraînement diminue à chaque époque et la précision de l'entraînement augmente à chaque époque. Mais ce n'est pas le cas pour la perte de validation et la précision. Elles semblent culminer à la quatrième époque. Ceci est un exemple de ce contre quoi nous avons mis en garde plus tôt : un modèle qui fonctionne mieux sur les données d'apprentissage n'est pas nécessairement un modèle qui fonctionnera mieux sur des données qu'il n'a jamais vues auparavant.

En termes précis, ce que vous voyez est un surajustement : après la quatrième époque, vous suroptimisez les données d'entraînement et vous finissez par apprendre des représentations spécifiques aux données d'entraînement et ne généralisez pas aux données en dehors de l'entraînement.

Dans ce cas, pour éviter le surajustement, vous pouvez arrêter l'entraînement après quatre époques.

### 3.1.7. Evaluation du modèle sur des nouveaux échantillons

Entraînons à nouveau notre réseau à partir de zéro pendant quatre epochs, puis évaluons-le sur les données de test.

```
# 1. Construction du réseau
library(keras)
model = keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu",
```

```
  input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
# 2. Configuration du réseau
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
# 3. Formation du réseau
model %>% fit(x_train, y_train, epochs = 4,
              batch_size = 512)
# 4. Evaluation du réseau su l'échantillon test
results = model %>% evaluate(x_test, y_test)
```

### 3.1.8. Utiliser un réseau formé pour générer des prédictions sur de nouvelles données

Après avoir formé un réseau, vous souhaiterez l'utiliser dans un cadre pratique. Vous pouvez générer la probabilité que les avis soient positifs en utilisant la méthode *predict()* :

```
model %>% predict(x_test[1:10,])
```

Comme on peut le voir, le réseau est confiant pour certains échantillons (0,99 ou plus, ou 0,01 ou moins) mais moins confiants pour les autres (0,7;0,2).

### 3.1.9. Améliorer les performances du model

Les expériences suivantes nous aideront à nous convaincre que les choix d'architecture que nous avons faits sont tous assez raisonnables, même s'il y a encore place à l'amélioration :

- Nous avons utilisé deux couches masquées. On peut d'utiliser une ou trois couches masquées et voir comment cela affecte la validation et la précision des tests.
- On peut essayer d'utiliser des calques avec plus d'unités masquées ou moins d'unités masquéesÿ: 32 unités, 64 unités, etc.
- Essayer d'utiliser la fonction de perte mse au lieu de binary_crossentropy.
- Essayer d'utiliser l'activation tanh (une activation populaire au début des réseaux de neurones) au lieu de relu.

### 3.1.10. Recommandation

Voici ce que vous devez retenir de cet exemple :

- Vous devez généralement effectuer un peu de prétraitement sur vos données brutes afin de pouvoir les alimenter, sous forme de tenseurs, dans un réseau de neurones. Les séquences de mots peuvent être encodées sous forme de vecteurs binaires, mais il existe également d'autres options d'encodage.
- On peut utiliser des piles de couches denses avec des activations relu pour résoudre un large éventail de problèmes (y compris la classification des sentiments), et vous les utiliserez probablement fréquemment.
- Dans un problème de classification binaire (deux classes de sortie), votre réseau doit se terminer par une couche dense avec une unité et une activation sigmoïde : la sortie de votre réseau doit être un scalaire compris entre 0 et 1, codant une probabilité.
- Avec une telle sortie sigmoïde scalaire sur un problème de classification binaire, la fonction de perte que vous devez utiliser est binary_crossentropy.
- L'optimiseur rmsprop est généralement un bon choix, quel que soit votre problème. C'est un souci de moins pour vous.
- Au fur et à mesure qu'ils s'améliorent sur leurs données d'entraînement, les réseaux de neurones finissent par se sur-adapter et finissent par obtenir des résultats de plus en plus mauvais sur des données qu'ils n'ont jamais vues auparavant. Assurez-vous de toujours surveiller les performances sur les données qui ne font pas partie de l'ensemble d'apprentissage.

## 3.2. Classification des fils de presse : Une classification multiclasse

Dans la section précédente, nous avons vu comment classer les entrées vectorielles en deux classes mutuellement exclusives à l'aide d'un réseau neuronal densément connecté. Mais que se passe-t-il lorsque vous avez plus de deux classes ?

Dans cette section, nous allons créer un réseau pour classer les fils de presse de Reuters en 46 sujets mutuellement exclusifs. Étant donné que nous avons de nombreuses classes, ce problème est une instance de classification multiclasse; et comme chaque point de données doit être classé dans une seule catégorie, le problème est plus spécifiquement une instance de classification à étiquette unique et multiclasse. Si chaque point de données pouvait appartenir à plusieurs catégories (dans ce cas, des sujets), vous seriez confronté à un problème de classification multiétiquette et multiclasse.

### 3.2.1. Le jeu de données Reuters

C'est un ensemble de courts fils de presse et leurs sujets, publié par Reuters en 1986. Il s'agit d'un ensemble de données de jouets simple et largement utilisé pour la classification de texte. Il y a 46 sujets différents; certains sujets sont plus représentés que d'autres, mais chaque sujet a au moins 10 exemples dans l'ensemble de formation.

```r
# chargement de la base
library(keras)
reuters <- dataset_reuters(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% reuters
```

### 3.2.2. Préparation des données

- Nous pouvons vectoriser les données avec exactement le même code que dans l'exemple précédent.

```r
vectorize_sequences <- function(
sequences, dimension = 10000) {
    results = matrix(0, nrow = length(sequences), ncol = dimension)
    for (i in 1:length(sequences))
       results[i, sequences[[i]]] = 1
       return(results)
}
x_train <- vectorize_sequences(train_data)
x_test <- vectorize_sequences(test_data)
```

- Pour vectoriser les étiquettes, nous utilisons l'encodage Onehot,un format largement utilisé pour les données catégorielles, également appelé encodage catégorique.

```r
one_hot_train_labels <- to_categorical(train_labels)
one_hot_test_labels <- to_categorical(test_labels)
```

### 3.2.3. Construire du réseau de neurone

Ce problème de classification de sujet ressemble au problème de classification précédent des critiques de films : dans les deux cas, vous essayez de classer de courts extraits de texte. Mais il y a ici une nouvelle contrainte : le nombre de classes de sortie est passé de 2 à 46. La dimensionnalité de l'espace de sortie est beaucoup plus grande.

Dans une pile de couches denses comme celle que nous avons utilisée, chaque couche ne peut accéder qu'aux informations présentes dans la sortie de la couche précédente. Si une couche laisse tomber des informations pertinentes pour le problème de classification, ces informations ne peuvent jamais être récupérées par les couches suivantes : chaque couche peut potentiellement devenir un goulot d'étranglement d'informations. Dans l'exemple de la critique de film, nous avons utilisé des couches intermédiaires à 16 dimensions, mais un espace à 16 dimensions peut être trop limité pour apprendre à séparer 46 classes différentes : de telles petites couches peuvent agir comme des goulots d'étranglement d'informations, supprimant de manière permanente les informations pertinentes.

Pour cette raison, nous utiliserons des calques plus grands. Allons-y avec 64 unités.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu",
              input_shape = c(10000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 46, activation = "softmax")
```

Il y a deux autres choses à noter à propos de cette architecture :

- Nous terminons le réseau avec une couche dense de taille 46. Cela signifie que pour chaque échantillon d'entrée, le réseau produira un vecteur à 46 dimensions. Chaque entrée dans ce vecteur (chaque dimension) encodera une classe de sortie différente.
- La dernière couche utilise une activation **softmax**. Cela signifie que le réseau produira une distribution de probabilité sur les 46 classes de sortie différentes.

La meilleure fonction de perte à utiliser dans ce cas est **categorical_crossentropy**. Il mesure la distance entre deux distributions de probabilité : ici, entre la distribution de probabilité sortie par le réseau et la vraie distribution des étiquettes. En minimisant la distance entre ces deux distributions, nous entraînons le réseau à produire quelque chose d'aussi proche que possible des véritables étiquettes.

```
model %>% compile(
optimizer = "rmsprop",
loss = "categorical_crossentropy",
metrics = c("accuracy")
)
```

### 3.2.4. Formation du modèle

- Nous séparons 1 000 échantillons dans les données d'apprentissage à utiliser comme ensemble de validation.

```
val_indices = 1:1000
x_val = x_train[val_indices,]
partial_x_train = x_train[- val_indices,]
y_val = one_hot_train_labels[val_indices,]
partial_y_train = one_hot_train_labels[- val_indices,]
```

- Maintenant, formons le réseau pendant 20 époques.

```
history <- model%>% fit(
      partial_x_train,
      partial_y_train,
      epochs = 20,
      batch_size = 512,
      validation_data = list(x_val, y_val)
)
```

- Et enfin, nous pouvons afficher ses courbes de perte et de précision

```
plot(history)
```

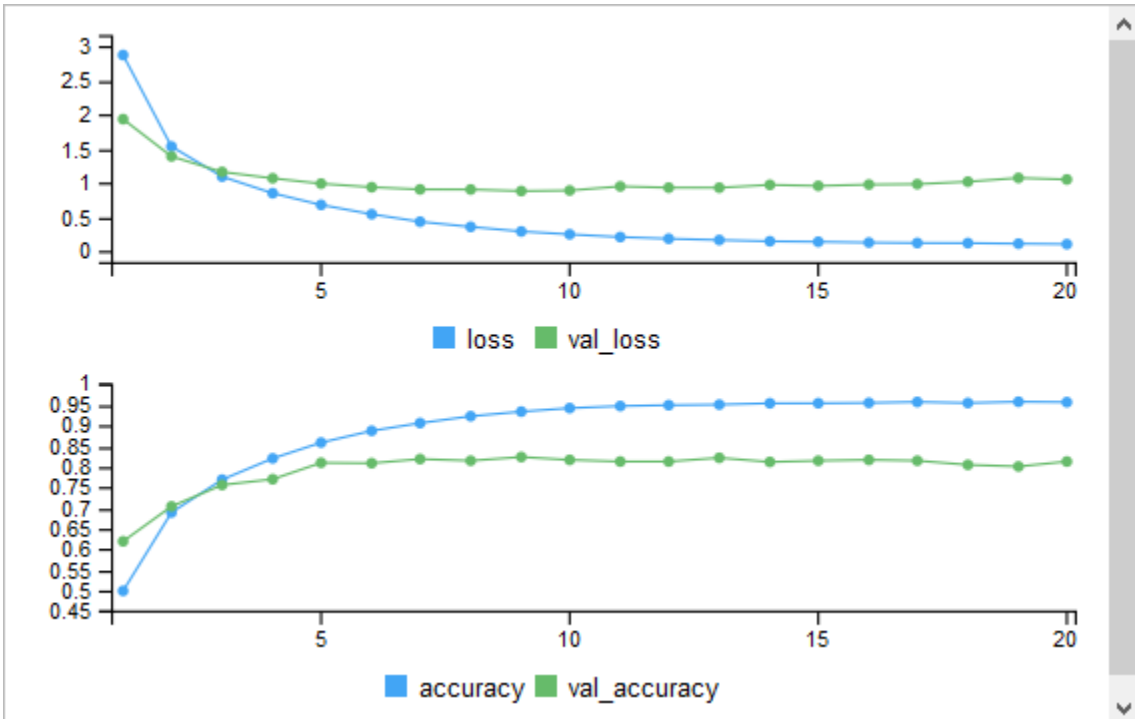


Figure 5: Performances du modèle par époque

- Le réseau commence à sur-adapter après neuf époques. Entraînons un nouveau réseau à partir de zéro pendant neuf époques, puis évaluons-le sur l'ensemble de test.

```
# construction du modèle
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu",
              input_shape = c(10000)) %>%
  layer_dense(units = 64,
              activation = "relu") %>%
```

```
  layer_dense(units = 46, activation = "softmax")

# Compilation du modèle
model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

# formation du modèle
history <- model%>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 9,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

# Evaluation du modèle sur l'échantillon test
results <- model %>% evaluate(x_test,
                        one_hot_test_labels)
results
```

Cette approche atteint une précision de $\sim 78{,}2\%$. Ce qui semble assez bon.

### 3.2.5. Récapitulation et recommandations

- **Si vous essayez de classer des points de données parmi N classes, votre réseau doit se terminer par une couche dense de taille N.**
- **Dans un problème de classification à étiquette unique et multiclasse, votre réseau doit se terminer par une activation softmax afin qu'il produise une distribution de probabilité sur les N classes de sortie.**
- **L'entropie croisée catégorielle (loss = "categorical_crossentropy") est presque toujours la fonction de perte que vous devez utiliser pour de tels problèmes. Il minimise la distance entre les distributions de probabilité émises par le réseau et la vraie distribution des cibles.**
- **Il existe deux manières de gérer les étiquettes dans la classification multi-classe :**

**– Encodage des étiquettes via l'encodage catégoriel (également connu sous le nom d'encodage onehot) et en utilisant categorical_crossentropy comme fonction de perte.**

**– Encodage des étiquettes sous forme d'entiers et utilisation de la fonction de perte sparse_categorical_crossentropy**

- **Si vous avez besoin de classer des données dans un grand nombre de catégories, vous devez éviter de créer des goulots d'étranglement d'informations dans votre réseau en raison de couches intermédiaires qui sont trop petits.**

## 3.3. Prévoir les prix des logements : un exemple de régression

Dans cette section, nous tenterons de prédire le prix médian des maisons dans une banlieue donnée de Boston au milieu des années 1970, en fonction de points de données sur la banlieue à l'époque, tels que le taux de criminalité, le taux de la taxe foncière locale, etc. L'ensemble de données que nous utiliserons présente une différence intéressante par rapport aux deux exemples précédents. Il a relativement peu de points de données : seulement 506, répartis entre 404 échantillons d'apprentissage et 102 échantillons de test. Et chaque caractéristique dans les données d'entrée (par exemple, le taux de criminalité) a une échelle différente. Par exemple, certaines valeurs sont des proportions, qui prennent des valeurs entre 0 et 1 ; d'autres prennent des valeurs entre 1 et 12, d'autres entre 0 et 100, etc.

### 3.3.1. Chargement de l'ensemble de données sur le logement de Boston

```
library(keras)
dataset <- dataset_boston_housing()
c(c(train_data, train_targets), c(test_data, test_targets)) %<-% dataset
str(train_data)
str(test_data)
```

Nous avons 404 échantillons d'apprentissage et 102 échantillons de test, chacun avec 13 caractéristiques numériques, telles que le taux de criminalité par habitant, le nombre moyen de pièces par logement, l'accessibilité aux autoroutes, etc.

Les cibles sont les valeurs médianes des maisons occupées par leur propriétaire, en milliers de dollars.

### 3.3.2. Préparation des données

Il serait problématique d'alimenter un réseau de neurones avec des valeurs qui prennent toutes des plages très différentes. Le réseau pourrait peut-être s'adapter automatiquement à ces données hétérogènes, mais cela rendrait certainement l'apprentissage plus difficile. Une bonne pratique répandue pour traiter de telles données consiste à effectuer une normalisation caractéristique : pour chaque caractéristique dans les données d'entrée (une colonne dans la matrice de données d'entrée), vous soustrayez la moyenne de la caractéristique et divisez par l'écart type, de sorte que la caractéristique est centré autour de 0 et a un écarttype unitaire. Cela se fait facilement dans R en utilisant la fonction **scale()**.

```
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
train_data <- scale(train_data, center = mean, scale = std)
test_data <-scale(test_data, center = mean, scale = std)
```

Notez que les quantités utilisées pour normaliser les données de test sont calculées à l'aide des données d'apprentissage. Vous ne devez jamais utiliser dans votre flux de travail une quantité calculée sur les données de test, même pour quelque chose d'aussi simple que la normalisation des données.

### 3.3.3. Construction du modèle

Étant donné que si peu d'échantillons sont disponibles, nous utiliserons un très petit réseau avec deux couches cachées, chacune avec 64 unités. En général, moins vous avez de données d'entraînement, pire sera le surapprentissage, et l'utilisation d'un petit réseau est un moyen d'atténuer le surapprentissage.

```
build_model <- function(){
  model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu",
      input_shape = dim(train_data)[[2]]) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1)

model %>% compile(
          optimizer = "rmsprop",
          loss = "mse",
          metrics = c("mae"))
}
```

- Le réseau se termine par une seule unité et aucune activation (ce sera une couche linéaire). Il s'agit d'une configuration typique pour la régression scalaire (une régression où vous essayez de prédire une seule valeur continue). L'application d'une fonction d'activation limiterait la plage que la sortie peut prendre ; par exemple, si vous appliquiez une fonction d'activation sigmoïde à la dernière couche, le réseau ne pourrait apprendre à prédire que des valeurs comprises entre 0 et 1. Ici, parce que la dernière couche est purement linéaire , le réseau est libre d'apprendre à prédire des valeurs dans n'importe quelle plage.
- Notez que nous compilons le réseau avec la fonction de perte **mse** (erreur quadratique moyenne), le carré de la différence entre les prédictions et les cibles. Il s'agit d'une fonction de perte largement utilisée pour les problèmes de régression.
- Nous surveillons également une nouvelle métrique pendant l'entraînement : l'erreur absolue moyenne (MAE). C'est la valeur absolue de la différence entre les prédictions et les cibles. Par exemple, un MAE de 0,5 sur ce problème signifierait que vos prédictions sont erronées de 500 $.

### 3.3.4. Validation du modèle à l'aide de la validation K-fold

Pour évaluer votre réseau tout en ajustant ses paramètres (tels que le nombre d'époques utilisées pour l'entraînement), vous pouvez diviser les données en un ensemble d'apprentissage et un ensemble de validation, comme nous l'avons fait dans les exemples précédents. Mais parce que nous avons si peu de points de données, l'ensemble de validation finirait par être très petit (par exemple, environ 100 exemples). Par conséquent, les scores de validation peuvent varier considérablement en fonction des points de données que vous avez choisi d'utiliser pour la validation et de ceux que vous avez choisis pour la formation. Cela vous empêcherait d'évaluer le modèle de manière fiable.

- La meilleure pratique dans de telles situations consiste à utiliser la validation croisée Kfold. Elle consiste à découper les données disponibles en K partitions (typiquement K = 4 ou 5), à instancier K modèles identiques, et à entraîner chacun sur K-1 partitions tout en évaluant sur la partition restante. Le score de validation du modèle utilisé est alors la moyenne des K scores de validation obtenus. En termes de code, c'est simple :

```
# Découper les données disponibles pour l'entrainement en k partitions
k <- 4
indices <- sample(1:nrow(train_data))
folds <- cut(indices, breaks = k, labels = FALSE)
num_epochs <- 100
all_scores <- c()

for (i in 1:k) {
  # Regrouper k-1 partitions pour l'entrainnement et évaluer le modèle sur la partitio
  cat("\n\n processing fold #", i, "\n---------------\n")
  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]
  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]
  # Construition du modèle Keras (déjà compilé)
  model <- build_model()
  # Formation du modèle sur les k-1 partitions
  model %>% fit(partial_train_data, partial_train_targets,
  epochs = num_epochs, batch_size = 1, verbose = 0)
  # Evaluation du modèle sur la partition i
  results <- model %>% evaluate(val_data, val_targets, verbose = 0)
  print(results)
  all_scores <- c(all_scores, results[2])
}
cat('moyenne mae',mean(all_scores))
```

Les différents runs montrent en effet des scores de validation assez différents, de 2,18 à 2,75. La moyenne (2,47) est une mesure beaucoup plus fiable que n'importe quel score unique :

c'est tout l'intérêt de la validation croisée Kfold. Dans ce cas, nous perdons en moyenne 2 470 $, ce qui est important compte tenu du fait que les prix varient de 10 000 $ à 50 000 $.

- Essayons d'entraîner le réseau un peu plus longtemps : 500 époques. Pour conserver une trace des performances du modèle à chaque époque, nous allons modifier la boucle d'entraînement pour enregistrer le journal des scores de validation par époque.

```
num_epochs <- 500
all_mae_histories <- NULL

for (i in 1:k) {
  # Regrouper k-1 partitions pour l'entrainnement et évaluer le modèle sur la partitio
  cat("\n\n processing fold #", i, "\n---------------\n")
  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]
  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]
  # Construition du modèle Keras (déjà compilé)
  model <- build_model()
  # Formation du modèle sur les k-1 partitions
  # en conservant les traces ses performances à chaque époques
  history <- model%>% fit(
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),
    epochs = num_epochs, batch_size = 1, verbose = 0
  )
  print(history)
  # Score de validation
  mae_history <- history$metrics$val_mae
  all_mae_histories <- rbind(all_mae_histories, mae_history)
}
```

- Nous pouvons ensuite calculer la moyenne des scores MAE par époque pour tous les plis afin de visualiser les performances du modèle en donction du nombre d'époques.

```
average_mae_history <- data.frame(
  epoch = seq(1:ncol(all_mae_histories)),
  validation_mae = apply(all_mae_histories, 2, mean)
  )
ggplot(average_mae_history, aes(x = epoch,
                          y = validation_mae)) +
                          geom_line()
```

Selon ce graphique, la validation MAE cesse de s'améliorer de manière significative après 52 époques. Passé ce point, le modèle commence à sur-ajuster.

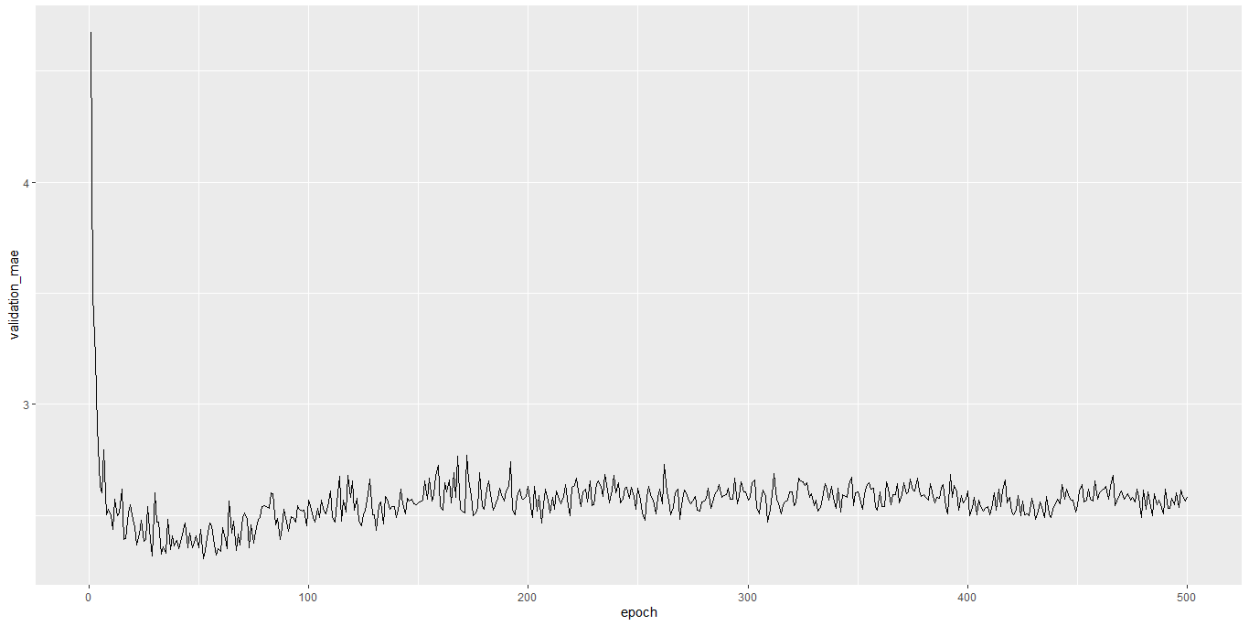


Figure 6: Plotting validation scores

### 3.3.5. Formation du modèle final

- Une fois que vous avez fini de régler d'autres paramètres du modèle (en plus du nombre d'époques, vous pouvez également ajuster la taille des couches cachées), vous pouvez former un modèle de production final sur toutes les données de formation, avec les meilleurs paramètres , puis examinez ses performances sur les données de test.

```
model <- build_model()
model %>% fit(train_data, train_targets,
   epochs = 52, batch_size = 16, verbose = 0)

result <-model%>% evaluate(test_data, test_targets)
```

Nous sommes toujours en retard d'environ 2 530 $.

### 3.3.6. Récapitulation et recommandations

Voici ce que vous devez retenir de cet exemple :

- **La régression est effectuée à l'aide de fonctions de perte différentes de celles de la classification. L'erreur quadratique moyenne (MSE) est une fonction de perte couramment utilisée pour la régression.**

- **De même, les mesures d'évaluation à utiliser pour la régression diffèrent de celles utilisées pour la classification; naturellement, le concept de précision ne s'applique pas à la régression. Une métrique de régression courante est l'erreur absolue moyenne (MAE).**
- **Lorsque les caractéristiques des données d'entrée ont des valeurs dans différentes plages, chaque entité doit être mise à l'échelle indépendamment en tant qu'étape de prétraitement.**
- **Lorsqu'il y a peu de données disponibles, l'utilisation de la validation Kfold est un excellent moyen d'évaluer de manière fiable un modèle.**
- **Lorsque peu de données d'entraînement sont disponibles, il est préférable d'utiliser un petit réseau avec peu de couches cachées (généralement une ou deux seulement), afin d'éviter un surajustement important.**

## 4. Conclusion

Vous pouvez désormais gérer les types de tâches de machine learning les plus courants sur des données vectorielles : classification binaire, classification multiclasse et régression scalaire. Les sections «Récapitulation» plus haut dans ce chapitre résument les points importants que vous avez appris concernant ces types de tâches.

Selon le type de problème d'apprentissage, le choix des paramètres de configuration du réseau peut se faire comme suit.

| Problem type Lastlayer | activation | Loss function |
|---|---|---|
| Binary classification | sigmoid | binary_crossentropy |
| Multiclass, singlelabel classification | softmax | categorical_crossentropy |
| Multiclass, multilabel classification | sigmoid | binary_crossentropy |
| Regression to arbitrary values | None | mse |
| Regression to values between 0 and 1 | sigmoid | mse or binary_crossentropy |

# III. Les réseaux de neurones convolutionnels (convnets)

Nous sommes sur le point de plonger dans la théorie de ce que sont les convnets et pourquoi ils ont si bien réussi dans les tâches de vision par ordinateur. Les réseaux de neurones convolutifs ont une méthodologie similaire à celle des méthodes traditionnelles d'apprentissage supervisé : ils reçoivent des images en entrée, détectent les features de chacune d'entre elles, puis entraînent un classifieur dessus.

Cependant, les features sont apprises automatiquement ! Les CNN (Convolutional Neural Network) réalisent eux-mêmes tout le boulot fastidieux d'extraction et description de features : lors de la phase d'entraînement, l'erreur de classification est minimisée afin d'optimiser les paramètres du classifieur ET les features ! De plus, l'architecture spécifique du réseau permet d'extraire des features de différentes complexités, des plus simples au plus sophistiquées. L'extraction et la hiérarchisation automatiques des features, qui s'adaptent au problème donné, constituent une des forces des réseaux de neurones convolutifs : plus besoin d'implémenter un algorithme d'extraction "à la main".

Contrairement aux techniques d'apprentissage supervisé, les réseaux de neurones convolutifs apprennent les features de chaque image. C'est là que réside leur force : les réseaux font tout le boulot d'extraction de features automatiquement, contrairement aux techniques d'apprentissage.

## 1. Différence entre un réseau de neurones et un réseau de neurones convolutif

Les réseaux de neurones convolutifs désignent une sous-catégorie de réseaux de neurones : ils présentent donc toutes les caractéristiques des réseaux de neurones vus aux chapitres 1 et 2. Cependant, les CNN sont spécialement conçus pour traiter des images en entrée. Leur architecture est alors plus spécifique : elle est composée de deux blocs principaux.

- Le premier bloc fait la particularité de ce type de réseaux de neurones, puisqu'il fonctionne comme un extracteur de features. Pour cela, il effectue du template matching en appliquant des opérations de filtrage par convolution. La première couche filtre l'image avec plusieurs noyaux de convolution, et renvoie des "feature maps", qui sont ensuite normalisées (avec une fonction d'activation) et/ou redimensionnées.

- Le second bloc n'est pas caractéristique d'un CNN : il se retrouve en fait à la fin de tous les réseaux de neurones utilisés pour la classification. Les valeurs du vecteur en entrée sont transformées (avec plusieurs combinaisons linéaires et fonctions d'activation) pour renvoyer un nouveau vecteur en sortie. Ce dernier vecteur contient autant d'éléments qu'il y a de classes : l'élément i représente la probabilité que l'image appartienne à la classe i. Ces probabilités sont calculées par la dernière couche de ce bloc (et donc du réseau), qui utilise une fonction logistique (classification binaire) ou une fonction softmax (classification multi-classe) comme fonction d'activation.

Comme pour les réseaux de neurones ordinaires, les paramètres des couches sont déterminés par rétropropagation du gradient : l'entropie croisée est minimisée lors de la phase d'entraînement.

## 2. Les couches d'un réseau de neurones convolutif

Il existe quatre types de couches pour un réseau de neurones convolutif : la couche de **convolution**, la couche de **pooling**, la couche de correction **ReLU** et la couche **fully-connected**.

- **La couche de convolution** : C'est la composante clé des réseaux de neurones convolutifs, et constitue toujours au moins leur première couche. Elle reçoit en entrée plusieurs images, et calcule la convolution de chacune d'entre elles avec chaque filtre. Les filtres correspondent exactement aux features que l'on souhaite retrouver dans les images. Par exemple, si la question est de distinguer les chats des chiens, les features automatiquement définies peuvent décrire la forme des oreilles ou des pattes.
- **La couche pooling** : Ce type de couche est souvent placé entre deux couches de convolution : elle reçoit en entrée plusieurs feature maps, et applique à chacune d'entre elles l'opération de pooling qui consiste à réduire la taille des images, tout en préservant leurs caractéristiques importantes. La couche de pooling permet de réduire le nombre de paramètres et de calculs dans le réseau. On améliore ainsi l'efficacité du réseau et on évite le sur-apprentissage.
- **La couche correction ReLu** : La couche de correction ReLU remplace donc toutes les valeurs négatives reçues en entrées par des zéros. Elle joue le rôle de fonction d'activation.
- **La couche fully-connected** : La couche fully-connected constitue toujours la dernière couche d'un réseau de neurones, convolutif ou non – elle n'est donc pas caractéristique d'un CNN. Cette dernière couche permet de classifier l'image en entrée du réseau : elle renvoie un vecteur de taille N, où N est le nombre de classes dans notre problème de classification d'images. Chaque élément du vecteur indique la probabilité pour l'image en entrée d'appartenir à une classe.

## 3. Utilisation d'un CNN pour classer les chiens des chats

### 3.1. Préparation des données

L'ensemble de données Dogs vs. Cats que nous utiliserons n'est pas fourni avec Keras. Il a été mis à disposition par Kaggle dans le cadre d'un concours de vision par ordinateur fin 2013, à l'époque où les convnets n'étaient pas courants. Vous pouvez télécharger l'ensemble de données original à partir de https://www.kaggle.com/datasets/tongpython/cat-and-dog/download.

Cet ensemble de données contient 10 000 images de chiens et de chats (5 000 de chaque classe dont 4000 échantillons d'entrainement et 1000 échantillons de test). Après l'avoir téléchargé et décompressé, vous allez créer un nouvel ensemble de données contenant trois sous-ensembles : un ensemble d'apprentissage avec 3 000 échantillons de chaque classe, un ensemble de validation avec 1 000 échantillons de chaque classe et un ensemble de test avec 1 000 échantillons de chaque classe.

```
## dossier des données d'entrainement initial
base_train_dir <- "training_set"
dir.create(base_train_dir)
## dossier des données de test initial
base_test_dir <- "test_set"
dir.create(base_test_dir)
## repertoir d'entrainement (contenant les données à utiliser pour
## l'entrainement avant la validation du modèle)
train_dir <-file.path("train")
dir.create(train_dir)
## dossier de validation
validation_dir <-file.path("validation")
dir.create(validation_dir)
## dossier de test
test_dir <-file.path("test")
dir.create(test_dir)
## sous dossier d'entrainement des données de chats
train_cats_dir <-file.path(train_dir, "cats")
dir.create(train_cats_dir)
##sous dossier d'entrainement des données de chiens
train_dogs_dir <-file.path(train_dir, "dogs")
dir.create(train_dogs_dir)
## sous dossier de validation des données de chats
validation_cats_dir <-file.path(validation_dir, "cats")
dir.create(validation_cats_dir)
##  sous dossier de validation des données de chiens
validation_dogs_dir <-file.path(validation_dir, "dogs")
dir.create(validation_dogs_dir)
##  sous dossier de test des données de chats
test_cats_dir <-file.path(test_dir, "cats")
dir.create(test_cats_dir)
## sous dossier de test des données de chiens
test_dogs_dir <-file.path(test_dir, "dogs")
dir.create(test_dogs_dir)

### Préparation des données des chiens
## Données d'entrainement
fnames <-paste0("training_set/dogs/dog.", 1:3000, ".jpg")
```

```
file.copy(file.path(base_train_dir, fnames),
          file.path(train_dogs_dir))

## Données de validation
fnames <-paste0("training_set/dogs/dog.", 3001:4000, ".jpg")
file.copy(file.path(base_train_dir, fnames),
          file.path(validation_dogs_dir))

## Données de test
fnames <-paste0("test_set/dogs/dog.", 4001:5000, ".jpg")
file.copy(file.path(base_test_dir, fnames),
          file.path(test_dogs_dir))

### Préparation des données des chats
## Données d'entrainement
fnames <-paste0("training_set/cats/cat.", 1:3000, ".jpg")
file.copy(file.path(base_train_dir, fnames),
          file.path(train_cats_dir))

## Données de validation
fnames <-paste0("training_set/cats/cat.", 3001:4000, ".jpg")
file.copy(file.path(base_train_dir, fnames),
          file.path(validation_cats_dir))

## Données de test
fnames <-paste0("test_set/cats/cat.", 4001:5000, ".jpg")
file.copy(file.path(base_test_dir, fnames),
          file.path(test_cats_dir))
```

### 3.2. Construire votre réseau

Nous construisons un convnet de une pile alternée d'une couche de convolution **layer_conv_2d** (avec activation relu) et d'une couche de pooling **layer_max_pooling_2d** étapes.

Mais parce que nous avons affaire à des images plus grandes et à un problème plus complexe, nous allons agrandir notre réseau en conséquence : il aura une étape supplémentaire **layer_conv_2d + layer_max_pooling_2d**. Cela sert à la fois à augmenter la capacité du réseau et à réduire davantage la taille des cartes d'entités afin qu'elles ne soient pas trop grandes lorsque nous atteignons **layer_flatten**. Ici, parce que nous partez d'entrées de taille $150 \times 150$ (un choix quelque peu arbitraire), vous vous retrouvez avec des cartes d'entités de taille $7 \times 7$ juste avant **layer_flatten**.

Parce que nous attaquons un problème de classification binaire, nous terminerons le réseau avec une seule unité (un **layer_dense** de taille 1) et une activation **sigmoïde**. Cette unité encodera la probabilité que le réseau regarde une classe ou l'autre.

```
library(keras)
model <- keras_model_sequential() %>%
layer_conv_2d(filters = 32, kernel_size = c(3, 3),
    activation = "relu", input_shape = c(150, 150, 3)) %>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_conv_2d(filters = 64, kernel_size = c(3, 3),
                    activation = "relu") %>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_conv_2d(filters = 128, kernel_size = c(3, 3),
                    activation = "relu") %>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_conv_2d(filters = 128, kernel_size = c(3, 3),
                    activation = "relu") %>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_flatten() %>%
layer_dense(units = 512, activation = "relu") %>%
layer_dense(units = 1, activation = "sigmoid")
```

La profondeur des cartes d'entités augmente progressivement dans le réseau (de 32 à 128), tandis que la taille des cartes d'entités diminue (de 148 × 148 à 7 × 7). C'est un modèle que vous verrez dans presque tous les convnets.

Pour l'étape de compilation, Nous irONS avec l'optimiseur RMSprop, comme d'habitude. Parce que nous terminons le réseau avec une seule unité sigmoïde, nous utiliserons l'entropie croisée binaire comme perte.

```
model %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("acc")
)
```

### 3.3. Prétraitement des données

Comme vous le savez maintenant, les données doivent être formatées dans des tenseurs à virgule flottante prétraités de manière appropriée avant d'être introduites dans le réseau. Actuellement, les données se trouvent sur un lecteur sous forme de fichiers JPEG, donc les étapes pour les intégrer au réseau sont à peu près les suivantes :

- Lisez les fichiers image.
- Décodez le contenu JPEG en grilles RVB de pixels.
- Convertissez-les en tenseurs à virgule flottante.

- Remettez à l'échelle les valeurs de pixel (entre 0 et 255) à l'intervalle [0, 1].

Cela peut sembler un peu intimidant, mais heureusement, Keras dispose d'utilitaires pour prendre en charge ces étapes automatiquement. Keras comprend un certain nombre d'outils d'aide au traitement d'images. En particulier, il inclut la fonction **image_data_generator()**, qui peut transformer automatiquement les fichiers image sur disque en lots de tenseurs prétraités. C'est ce que nous allons utiliser ici.

```
 # Redimensionne toutes les images au 1/255
train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)
#
train_generator <-flow_images_from_directory(
      train_dir, # Répertoire cible
      train_datagen, # Générateur de données d'entraînement
      target_size = c(150, 150), # Redimensionne toutes les images à 150 × 150
      batch_size = 20,
      class_mode = "binary" # Comme vous utilisez la perte binary_crossentropy, vous au
)
#
validation_generator <- flow_images_from_directory(
      validation_dir,
      validation_datagen,
      target_size = c(150, 150),
      batch_size = 20,
     class_mode = "binary"
)
```

### 3.4. Formation du modèle

Ajustons le modèle aux données à l'aide du générateur. Pour ce faire, utilisons la fonction **fit_generator**, l'équivalent de **fit** pour les générateurs de données comme celui-ci. Il attend comme premier argument un générateur qui produira indéfiniment des lots d'entrées et de cibles, comme celui-ci le fait. Étant donné que les données sont générées sans fin, le processus d'ajustement doit savoir combien d'échantillons prélever du générateur avant de déclarer une époque terminée. C'est le rôle de l'argument **steps_per_epoch**.

Lorsque vous utilisez **fit_generator**, vous pouvez passer un argument validation_data, comme avec la fonction **fit**. Il est important de noter que cet argument est autorisé à être un générateur de données, mais il peut également s'agir d'une liste de tableaux. Si vous transmettez un générateur en tant que validation_data, ce générateur est censé produire des lots de données de validation à l'infini ; vous devez donc également spécifier l'argument **validation_steps**, qui indique au processus le nombre de lots à tirer du générateur de validation pour l'évaluation.

```
history <- model %>% fit_generator(
   train_generator,
   steps_per_epoch = 100,
   epochs = 60,
   validation_data = validation_generator,
   validation_steps = 60
)
```

Il est toujours recommandé d'enrégistrer votre modèle après l'entrainement.

```
model %>% save_model_hdf5("cats_and_dogs_small_1.h5")
```

Nous pouvons maintenant visualiser les performances du modèle au cours de l'entrainement.

```
plot(history)
```

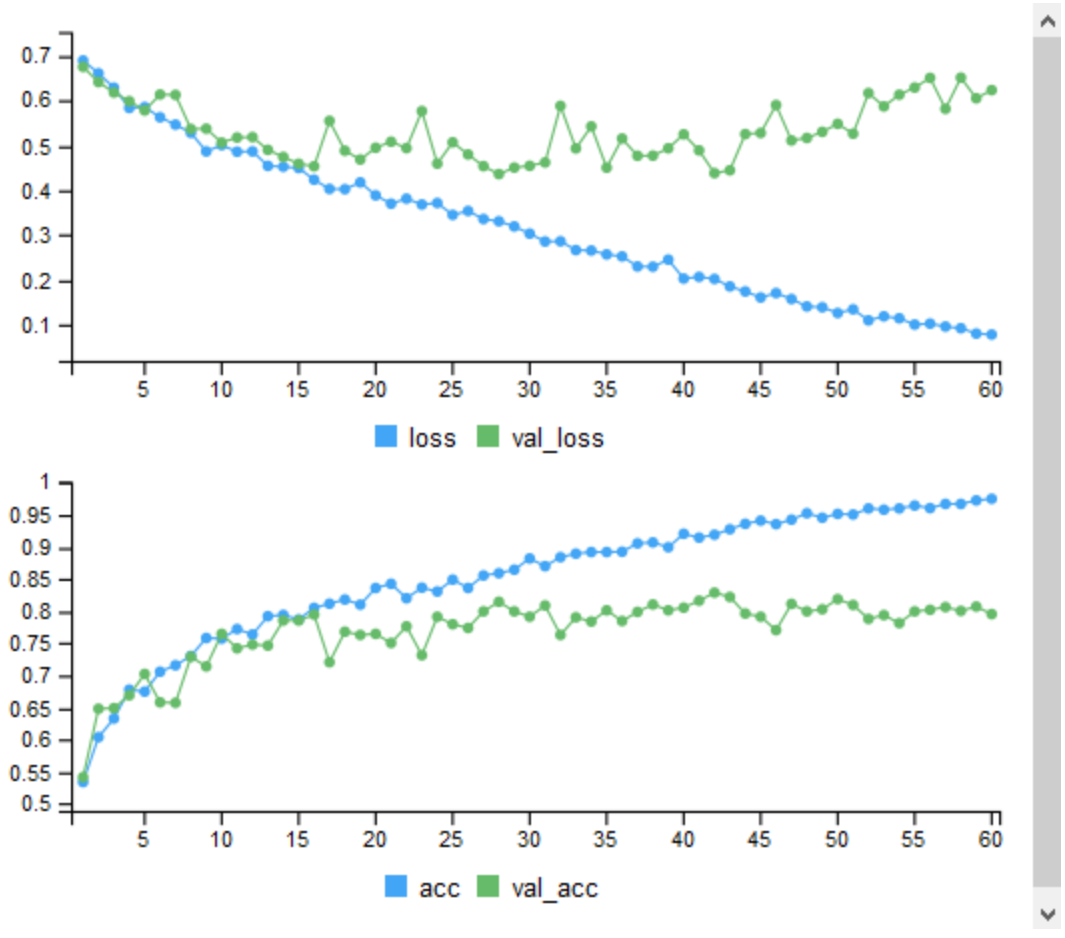


Figure 7: Performances du modèle par époque

On constate que, la perte d'entraînement ne cesse de diminuer linéairement à chaque époque et la précision de l'entraînement augmente à chaque époque. Mais ce n'est pas le cas pour la perte de validation et la précision. Elles semblent culminer à la seizième époque.

En termes précis, ce que vous voyez est un surajustement : après la seizième époque, vous suroptimisez les données d'entraînement et vous finissez par apprendre des représentations spécifiques aux données d'entraînement et ne vous généralisez pas aux données en dehors de l'entraînement.

La meilleur précision sur les données de validation se situe à la 42ème époque avec une précision de validation d'environ 83%.

### 3.4. Formation final du modèl

Entraînons maintenant le réseau que nous venons de former pendant 42 époques sur l'ensemble d'entrainement initial, puis évaluons-le sur l'ensemble de test.

```
## Ajouter les échantillons de valitation à l'ensemble d'entrainement
fnames <-paste0("dogs/dog.", 3001:4000, ".jpg")
file.copy(file.path(validation_dir, fnames),
          file.path(train_dogs_dir))
fnames <-paste0("cats/cat.", 3001:4000, ".jpg")
file.copy(file.path(validation_dir, fnames),
          file.path(train_cats_dir))

## Prétraitement des données
# Redimensionne toutes les images au 1/255
train_datagen <- image_data_generator(rescale = 1.0/255)
test_datagen <- image_data_generator(rescale = 1.0/255)
#
train_generator <-flow_images_from_directory(
  train_dir, # Répertoire cible
  train_datagen, # Générateur de données d'entraînement
  target_size = c(150, 150), # Redimensionne toutes les images à 150 × 150
  batch_size = 20,
  class_mode = "binary" # Comme vous utilisez la perte binary_crossentropy, vous avez b
)
#
test_generator <-flow_images_from_directory(
  test_dir, # Répertoire cible
  test_datagen, # Générateur de données d'entraînement
  target_size = c(150, 150), # Redimensionne toutes les images à 150 × 150
  batch_size = 20,
  class_mode = "binary" # Comme vous utilisez la perte binary_crossentropy, vous avez b
)
```

```
# Entrainement du modèle pré-formé
history <- model %>% fit_generator(
        train_generator,
        steps_per_epoch = 100,
        epochs = 42 )
# Evaluation du modèle
model %>% evaluate_generator(test_generator, steps = 2)
```

Nous arrivons à une précision de test de 92.5%. Ce qui montre un bon ajustement du modèle.

### 3.3. Remarques et récaputilations

Voici ce que vous devriez retenir de ce chapitre :

- **Les convnets sont le meilleur type de modèles d'apprentissage automatique pour les tâches de vision par ordinateur. Il est possible d'en former un à partir de zéro, même sur un très petit ensemble de données, avec des résultats décents.**
- **Les convnets fonctionnent en apprenant une hiérarchie de modèles et de concepts modulaires pour représenter le monde visuel.**
- **Sur un petit ensemble de données, le surajustement sera le principal problème. L'augmentation des données est un moyen puissant de lutter contre le surajustement lorsque vous travaillez avec des données d'image.**
- **Vous êtes maintenant capable de former votre propre convnet à partir de zéro pour résoudre un problème de classification d'images.**

# Conclusion

C'est la fin du Deep Learning avec R ! J'espère que vous avez appris une ou deux choses sur l'apprentissage automatique, l'apprentissage en profondeur, Keras. L'apprentissage est un voyage de toute une vie, en particulier dans le domaine de l'IA, où nous avons bien plus d'inconnues que de certitudes. Alors, s'il vous plaît, continuez à apprendre, à vous questionner et à faire des recherches. N'arrête jamais. Car même compte tenu des progrès réalisés jusqu'à présent, la plupart des questions fondamentales de l'IA restent sans réponse. Beaucoup n'ont même pas encore été correctement interrogés.

# Recommandations et références bibliographiques

Keras dispose d'un vaste écosystème de tutoriels, de guides et de projets open source associés :

- Votre principale référence pour travailler avec l'interface Keras R est la documentation en ligne sur https://keras.rstudio.com.
- Le blog Keras, https://blog.keras.io, propose des tutoriels Keras et d'autres articles liés à l'apprentissage en profondeur.
- Le blog TensorFlow pour R, https://tensorflow.rstudio.com/blog.html, propose des articles sur l'utilisation des interfaces R avec Keras et TensorFlow.
- Cet exposé s'inspire en majeure partie du livre Deep learning with R de François Chollet et J.J. Allaire disponible sur https://fr.b-ok.africa/book/3504490/502088.