Projet numérique : Réseau de neurones artificiels

Léo Reynaud et Théo Dubroca

Avril 2022

Contents

Réseau de neurones :
1ère couche : couche "sensorielle"
2e couche : couche cachée qui va pondérer le signal envoyé par la 1ère couche
3e couche : " " 2eme couche
.
.
.
Dernière couche : somme les signaux pondérés et analyse ceux-ci pour en tirer des conclusions

Un réseau de neurones peut être unidirectionnel mais il peut aussi boucler afin d'afiner le résultat.
Problèmes d'un réseau de neurones : suraprentissage, etc

1 Introduction

Les recherches scientifiques en biologie sur le fonctionnement du cerveau sont parmi les plus prometteuses afin de comprendre comment il est possible pour un être vivant d'interagir et d'apprendre de son environnement. Avec cette forme d'intelligence, qu'est la capacité à résoudre des problèmes qui s'offrent à nous dans la vie de tous les jours et notre capacité à répéter de telles opérations sur des situations différentes. On parle alors d'apprentissage. Ce phénomène s'applique tant à l'utilisation de capacités motrices pour exécuter une action qu'à la reconnaissance de formes, d'images que nous pouvons identifier, classer selon nos souvenirs et nos connaissances. Par exemple lorsqu'un nouveau né voit un cube en mousse celui-ci ne se dit pas "c'est un cube" mais pourtant il est capable de différencier ce cube d'une boule ou d'un objet d'une forme différente. Il est aussi capable au bout d'un moment de différencier ses parents et même de marcher à force de tentatives et après avoir regardé son entourage le faire, il a appris grâce à la répétition du phénomène et en le gardant en mémoire.
Avec la découverte des neurones et des signaux électriques échangés entre eux les scientifiques ont eu l'idée de les reproduire numériquement par analogie avec les neurones biologiques.
Ce biomimétisme a ouvert dans les années 40-50 un nouveau champ de recherche qui est le développement des réseaux de neurones artificiels. A la frontière entre la biologie et l'algorithmie ces réseaux sont aujourd'hui au coeur d'objet du quotidien et leur histoire est une succession d'éclatantes réussites et de désillusions tout aussi importantes. Le premier type de réseau neuronal artificiel fut développé dans un article publié en 1959 par Lettvin J.Y., Maturana H.R., McCulloch W.S., et Pitts W.H. dans lequel ils démontrèrent qu'un neurone biologique pourrait tout à fait être numériquement implémenté (de manière simplifié) en un réseau formel permettant de réaliser des opérations arithmétiques, logiques et symboliques complexes.
C'est dans les années 80 que naîtront les premières formes modernes de l'apprentissage automatique avec le perceptron multicouche ou, et c'est ce que nous allons étudier ici, le réseau neuronal convolutif.

2 Réseau de neurones artificiels

2.1 Le neurone formel : l'unité élémentaire des réseaux de neurones

Un neurone formel, par analogie au neurone biologique, est une fonction qui va recevoir des données x_i par une ou plusieurs entrées (<=> dendrites). L'équivalent excitateur et inhibiteur des synapses est l'association de poids w_i aux entrées. A cela nous devons ajouter l'équivalent du seuil d'activation en-deçà duquel le neurone ne sera pas activé par l'intermédiaire d'un biais. Enfin le neurone formel va associer une fonction d'activation, par analogie au potentiel d'activation, à la somme pondérée biaisée. Voici la représentation mathématique et schématique d'un neurone formel :

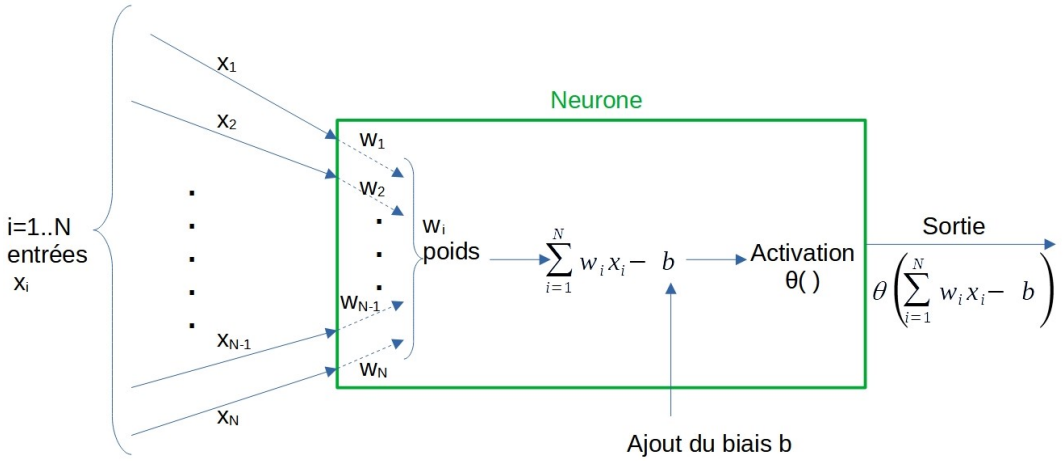


Figure 1: Représentation schématique d'un neurone formel avec son écriture mathématique en sortie

Dans un soucis d'efficacité il sera nécessaire de choisir des fonctions d'activations particulières à compter du moment ou nous mettrons plusieurs neurones formels en connection les uns avec les autres (i.e. création d'un réseau de neurones), et il faut aussi prendre en compte le type de données que nous mettons en entrée d'un réseau neuronal. C'est pourquoi le choix des fonctions d'activations est un choix primordial pour l'établissement d'un réseau de neurones.

2.2 Réseaux de neurones et fonctions d'activation

Historiquement le perceptron fut le premier réseau de neurones artificiels capable d'apprendre par expérience mais nous n'en parlerons que très peu ici car il fut très vite mis en échec par sa non capacité à résoudre des problèmes non-linéaires. C'est son successeur le perceptron multicouche qui permit de telles avancées. Développé dans les années 80 il permet de résoudre des problèmes non-linéaires, ce qui fut très pratique notamment en physique pour laquelle beaucoup de phénomènes sont non-linéaires et étaient donc difficilement étudiables, par la méthode de la rétropropagation du gradient.

2.2.1 Perceptron multicouche

Le perceptron multicouche est organisé de la manière suivante : la première couche de neurones est la couche d'entrée dans laquelle les données, comme par exemple la valeur en nuance de gris de pixel d'une image, sont traitées; vient ensuite plusieurs couches dites "cachées" qui vont attribuer des poids, des fonctions d'activations et des biais aux valeurs d'entrées, ces couches sont généralement au nombre de deux ; ensuite viennent les couches de sortie qui vont soit classer des images, soit donner les valeurs des paramètres que nous cherchons à évaluer pour un modèle physique selon ce que nous voulons que le réseau fasse. Il existe bien entendu d'autres applications à de tels réseaux mais ils restent limité notamment par leur lourds besoins en mémoire et puissance de calcul. Comme nous pouvons le voir sur le schéma ci-dessous, les neurones sont tous connectés entre eux, c'est pourquoi l'on appelle ce type de réseau de neurones des réseaux complètement connectés (fully connected network). De plus, ce type de réseau est unidirectionnel, plus précisément il est dit "feed forward" ce qui implique que les données ne vont que dans un seul sens : des couches d'entrée vers les couches

de sortie. Intéressons-nous maintenant aux couches cachées de ce réseau qui sont à la base de la réussite des perceptrons multicouches.

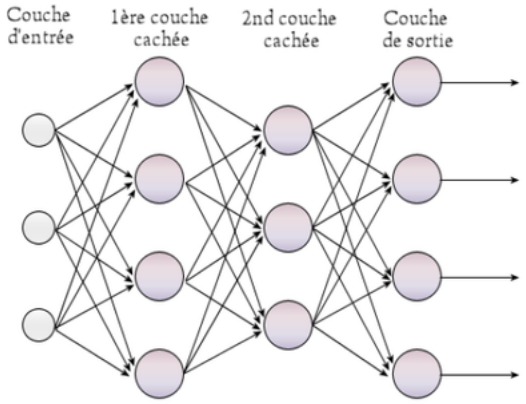


Figure 2: Représentation schématique d'un perceptron 4-couches

Les couches cachées comptent plusieurs neurones par couches qui chacun reçoivent le même nombre d'entrées. La première couche recevra les données d'entrée tandis que la deuxième couche cachée recevra comme données d'entrée les sorties de la première couche. Chaque neurone reçoit et ressort donc une combinaison linéaire d'inputs (<=> entrée) et de poids associés à ces inputs. Chaque neurone dispose de son propre biais et de sa fonction d'activation. Mathématiquement l'output d'un neurone d'une couche cachée s'écrit :

$$Y = \phi\left(\sum_{i=1}^{N} w_i X_i - b\right) \tag{1}$$

où Y est la valeur de sortie, b est le biais, X_i sont les données, w_i sont les poids associés aux données et ϕ est la fonction d'activation du neurone.

Comme chaque couche est composée de plusieurs neurones et qu'il y a plusieurs couches nous allons devoir complexifier l'écriture en ajoutant des indices mais cela nous permettra de réécrire les sommes sous forme matricielle pour chaque couche : un indice en haut indiquera à quelle couche nous nous trouvons à commencer par (0) pour la couche d'entrée tandis qu'un indice en bas nous indiquera sur quel neurone nous sommes, ainsi pour la première couche cachée nous avons :

$$a^{(1)} = \phi\left(\begin{bmatrix} w_{0,0} & \dots & w_{0,N} \\ w_{1,0} & \dots & w_{1,N} \\ \cdot & \dots & \cdot \\ \cdot & \dots & \cdot \\ \cdot & \dots & \cdot \\ w_{k,0} & \dots & w_{k,N} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \cdot \\ \cdot \\ \cdot \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \cdot \\ \cdot \\ \cdot \\ b_N \end{bmatrix}\right) \tag{2}$$

Comme nous l'avons dit plus tôt, la résolution des problèmes non-linéaires fut permise grâce à l'implémentation de la méthode de la rétropropagation du gradient. En effet, cette méthode vise à réduire les erreurs en cherchant à diminuer la fonction de coût ce qui est efficace car les problèmes non-linéaires ont tendance, si on cherche à les résoudre sans cette méthode, à accroître les erreurs et donc à grandement réduire la fiabilité des résultats obtenus. Mais comment fonctionne cette méthode ? Et que veut-elle dire concrètement ? La réponse à cette question dans la section suivante pour le cas du perceptron multicouche.

2.2.2 Rétropropagation du gradient

Commençons par définir la fonction de coût C : dans un réseau de neurones, la fonction de coût est une fonction qui va nous servir de critère afin de minimiser les erreurs faites par l'algorithme en corrigeant plus ou moins les biais et les poids selon l'importance de l'erreur qu'ils apportent. Pour ce faire, ayant des résultats attendus et des résultats obtenus car nous sommes en apprentissage

supervisé, nous allons calculer et différencier les carrés des résultats obtenus aux carrés des résultats attendus. Cette différence est ce que l'on appelle la fonction de coût que nous chercherons donc à minimiser car plus l'écart "obtenus-attendus" est faible, plus l'erreur l'est aussi. Minimiser cette différence revient à déterminer le minimum de la fonction $C(w^{(0)},..,w^{(L)},b^{(0)},..,b^{(L)})$par la descente pas à pas du gradient <=> nous forcerons les poids et biais à diminuer la fonction C en s'opposant au gradient de C.
Cette méthode s'implémente en calculant le gradient ∇C et en mettant à jour les poids et biais représentés par un vecteur $\vec{V} = \begin{pmatrix} \Delta w_{0,0}^{(0)} \\ . \\ . \\ . \\ \Delta b_N^{(L)} \end{pmatrix}$ tels que :

$$\Delta\vec{V} = -\eta\vec{\nabla}C \text{ Pour la descente du gradient} \tag{3}$$

$$\vec{V} \to \vec{V} - \eta\vec{\nabla}C \text{ Pour la rétropropgation} \tag{4}$$

où η est la vitesse d'apprentissage définie par l'utilisateur.
Voici un schéma de ce qu'est la descente du gradient pour deux paramètres :

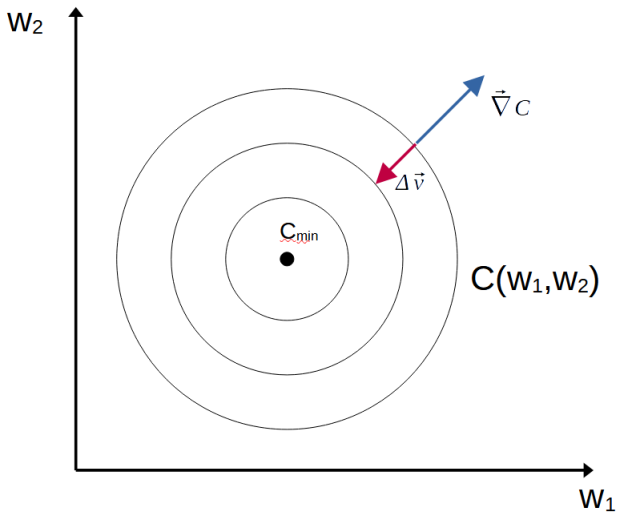


Figure 3: Représentation schématique de la descente du gradient pour deux poids w_1 et w_2

Un problème souvent rencontré avec cette méthode est la longueur du processus de calcul du gradient si le nombre de données est très grand car alors il y aura beaucoup de paramètres. Par exemple, si nous cherchons à faire de la reconnaissance d'image à partir de la banque d'images MNIST qui présente plusieurs milliers d'images de 28x28 pixels, et que nous prenons : en entrée les valeurs en nuance de gris de ces pixels nous avons 28x28=784 entrées ; en première couche cachée les bords, soit une couche qui détermine à quels endroits de l'image nous avons des pixels sombres (<=> écriture) et des pixels blancs (<=>pas d'écriture) ; en 2e couche cachée les motifs, par exemple un 9 est composé d'un rond en haut, d'une barre verticale à droite et d'une barre horizontale en bas ; et en sortie les neurones déterminent les chiffres correspondants, alors C sera dépendante de 13 000 paramètres (poids et biais) environ. Pour pallier à ce problème nous allons utiliser la méthode la descente stochastique du gradient qui consiste à mélanger les données d'entraînements et à ensuite les séparer en des mini-lots de 100 données pour lesquels nous calculerons l'erreur ∇C sans mettre à jour les poids et biais des mini-lots, erreurs que nous additionnerons pour ensuite effectuer la rétropropagation.

La rétropropagation nous est utile pour calculer $\vec{\nabla}C = \begin{pmatrix} \frac{\partial C}{\partial w_{0,0}^{(0)}} \\ . \\ . \\ . \\ \frac{\partial C}{b_N^{(L)}} \end{pmatrix}$. Pour un neurone $a_i^{(k)}$ nous avons $z_i^{(k)} = \sum_j w_{ij}^{(k)} a_j^{(k-1)} + b_j^{(k)}$. La rétropropagation est représentée par la formule suivante:

$\delta_i^{(k)} = \sum_j \delta_j^{(k+1)} w_{ji}^{(k+1)} \phi'(z_i^{(k)}$ car l'on voit que l'on passe de la couche (k+1) à la couche (k). Cela nous permet d'écrire les équations de $\vec{\nabla}C$:

$$\frac{\partial C}{\partial b_j^{(k)}} = \delta_j^{(k)} \tag{5}$$

$$\frac{\partial C}{\partial w_{ij}^{(k)}} = \sum_m \frac{\partial C}{\partial z_m^{(k)}} \frac{\partial z_m^{(k)}}{\partial w_{ij}^{(k)}} = \delta_i^{(k)} a_j^{(k-1)} \tag{6}$$

.

3 Réseau de neurones convolutif ou CNN pour Convolutional Neural Network

3.1 CNN pour Cifar10

Au contraire de son prédécesseur le CNN est un réseau de deep learning principalement utilisé pour la reconnaissance d'image. Ici nous utiliserons un apprentissage supervisé avec la banque d'image Cifar10 composée de 60 000 images de 32x32 pixels en couleurs RGB divisées en dix classes de 6000 images chacune : avion, voiture, oiseau, chat, cerf, chien, grenouille, cheval, bateau et camion. Parmi ces classes, 1000 images ont été aléatoirement sélectionnées pour former un lot de tests et les images restantes sont dans des lots de 5000 images aléatoirement sélectionnées aussi, ce sont les lots d'entraînement.
L'architecture générale des CNN est la suivante :

1. L'image est vue comme 3 couches de pixels correspondantes aux couleurs RGB, ou comme une seule couche si l'image est en niveaux de gris

2. Une couche de convolution composée de plusieurs filtres

3. Une couche de MaxPooling qui vient réduire la taille des couches de convolutions obtenues

4. Une couche de convolution elle aussi composée de plusieurs filtres

5. Une couche de MaxPooling, et ainsi de suite jusqu'à ce que la taille de l'image la rende inutile à convoluer

6. Vient ensuite une couche Flatten qui va prendre toutes les couches de convolution après les MaxPooling afin de les mettre sur une seule et même couche

7. Enfin cette image Flatten passe dans un réseau de neurones complètement connectés, couche Dense, comme un perceptron multicouche pour afficher en sortie des probabilités d'appartenance à telle ou telle classe. C'est dans cette couche notamment que se fait l'apprentissage par rétropropagation.

Comme nous pouvons le voir dans la fonction "creationmodele" du code ligne 27, l'architecture du modèle est celle décrite ci-dessus mais nous avons intercalé des couches Dropout qui vont éteindre

aléatoirement un certain pourcentage de neurones afin d'améliorer l'apprentissage du réseau en évitant le sur-apprentissage. Ce problème fréquemment rencontré dans les algorithmes d'apprentissage automatique consiste en l'apprentissage "par coeur" de paramètres : la classification des données est trop parfaite et donc le modèle perd en pouvoir de prédiction puisqu'il généralisera mal face à l'apport de données complètement nouvelles. L'utilisation des couches Dropout est une méthode dite de régularisation mais il existe d'autres méthodes possibles comme la validation croisée par exemple. De plus, on peut voir que les couches dépendent de paramètres, ou d'hyper-paramètres que nous allons définir :

1. Conv2D(a,(b,c),padding='same'...) où (a) est le nombre de filtres qui influe beaucoup sur le nombre de paramètres totaux à entraîner et (b,c) la taille du noyau qui va convoluer le long de l'image, le padding='same' stipule que les couches de convolution obtenues après la convolution doivent être de la même taille que l'image convoluée en ajoutant si nécessaire des pixels de valeurs nulles sur les bords.

2. MaxPooling2D((a,b)) avec (a,b) la taille de l'image après passage dans cette couche, cela permet de réduire la quantité de paramètres et donc d'optimiser la puissance de calcul nécessaire. De plus cela permet de réduire le sur-apprentissage puisqu'il y a moins de paramètres.

3. Dense(a) où a est le nombre de neurones complètement connectés

Dans notre code chaque couche se voit associée à une fonction d'activation 'elu'=Exponential Linear Unit qui est une forme dérivée de la 'ReLU'= Rectified Linear Unit. Généralement, un CNN utilise la 'ReLU' mais cela est de moins en moins vrai car la 'ReLU' peut entraîner la mort de neurones rendant son utilisation impossible et donc diminuant l'efficacité du réseau au contraire de la 'elu'. De plus cette dernière est considérée comme plus performante. La seule couche à ne pas utiliser cette fonction est la dernière dont le nombre de neurones correspond au nombre de classes. Cela est dû au fait que c'est la couche de sortie et qui sort donc des probabilités que l'on souhaite entre 0 et 1 c'est pourquoi on utilise la 'softmax'.
A partir de la ligne 61 nous précisons au modèle ses paramètres. On a la fonction de perte qu'il doit minimiser : ici c'est la fonction 'categorical crossentropy' utilisée pour des modèles avec plusieurs classes afin d'associer à chaque nom de classe une catégorie avec un encodage 1 parmis N. L'optimiseur 'Adam' est choisi car il permet d'avoir une précision plus importante en effectuant la rétropropagation et la descente du gradient plus efficacement : il est plus économique en puissance de calcul, de mémoire, utilise la descente stochastique et est donc adapté à notre base de données très importante. Enfin, nous choisissons le nombre d'époque, i.e. le nombre de fois que le modèle s'effectuera sur les données afin d'apprendre et d'améliorer sa précision.
Afin de chercher à améliorer le modèle nous avons fait varier le taux de Dropout, le nombre de convolution effectuées en ajoutant une couche de convolution et en faisant varier le nombre d'époques. Voici les résultats de l'étude du taux d'apprentissage et de perte:

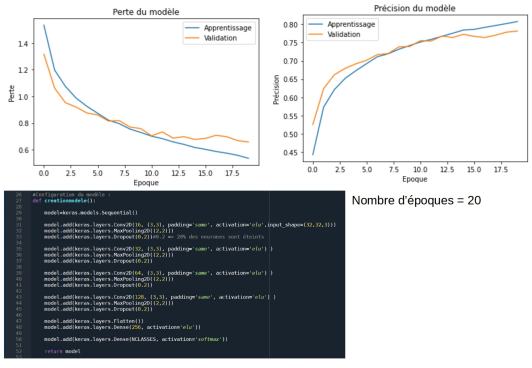


Figure 4: Courbe d'évolution de la perte et de la précision avec comparaison entre les données d'apprentissage et les données de validation avec une couche de convolution par séquence et 20 époques => environ 230 000 paramètres

Sur les courbes pour un nombre de 40 époques, des couches de convolution doublées et un dropout à 0.2 on observe que la courbe d'apprentissage finit par dépasser très fortement aux alentours de 10

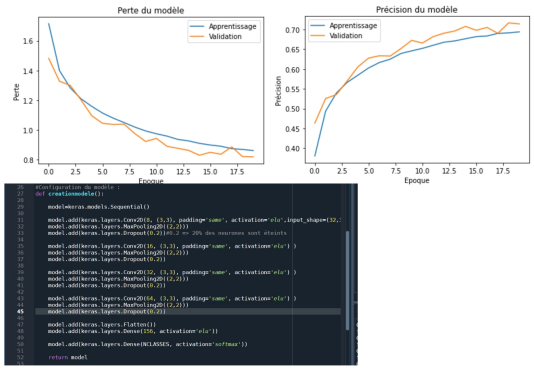


Figure 5: Courbe d'évolution de la perte et de la précision avec comparaison entre les données d'apprentissage et les données de validations avec une couche de convolution par séquence, nombres d'hyper-paramètres divisés par 2 et 20 époques => environ 66 000 paramètres

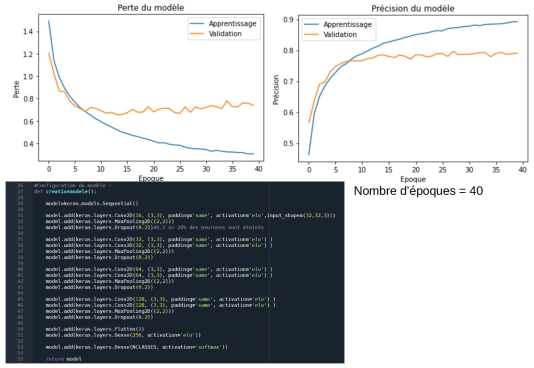


Figure 6: Courbe d'évolution de la perte et de la précision avec comparaison entre les données d'apprentissage et les données de validations avec deux couches de convolution par séquence et 40 époques => environ 400 000 paramètres

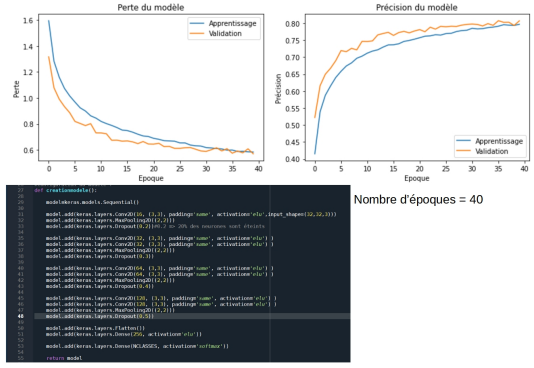


Figure 7: Courbe d'évolution de la perte et de la précision avec comparaison entre les données d'apprentissage et les données de validations avec deux couches de convolution par séquence, Dropout croissant et 40 époques => environ 400 000 paramètres

époques les courbes de validations ce qui signifie que notre modèle est en sur-apprentissage ; on peut ajouter que sa sa précision avant cela est trop faible car inférieure à 0.8, le modèle est donc à rejeter . Afin de régler ce problème, on augmente le dropout de 0.1 à chaque séquence ce qui nous permet d'obtenir aucun sur-apprentissage au bout des 40 époques et la précision est de 0.8 si ce n'est un peu plus en augmentant le nombre d'époques ce qui est correct, on peut donc garder ce modèle. De plus, ce modèle est meilleur que celui avec 20 époques, un dropout de 0.2 et une couche de convolution

par séquence car celui-ci sur-apprend aussi au bout de 12 époques environ alors que le nombre de paramètres à entraîner est divisé par deux par rapport au modèle précédent, donc on doit rejeter le modèle à 20 époques (qui a aussi une précision trop faible avant le sur-apprentissage). Augmenter le drop-out résoudrait ce problème. Si nous réduisons maintenant le nombre de paramètres en divisant par 2 le nombre de filtres de chaque convolutions et aussi par 2 le nombre de neurones dans la couche dense nous avons environ 66 000 paramètres au lieu des 230 000, alors le modèle ne sur-apprend pas mais la précision n'est plus convenable puisque d'à peine 0.7 au bout des 20 époques, on doit donc rejeter ce modèle.

On peut en conclure qu'augmenter le nombre d'époques ne permet pas de régler un problème de sur-apprentissage mais l'augmentation du dropout semble le régler sans avoir à changer le nombre de paramètres. A cela s'ajoute le fait que si nous changeons le nombre de paramètres en le divisant par 2 en gardant un dropout de 0.2, le problème de sur-apprentissage survient très tôt et ne peut donc être régler par le nombre d'époques ce qui souligne l'importance du dropout pour contrer le sur-apprentissage d'un modèle. Enfin un autre moyen de pallier au sur-apprentissage est de diminuer le nombre de paramètres à entraîner mais cela diminue la précision et donc l'on doit augmenter le nombre d'époques pour l'améliorer. Que ce soit pour un doublement des couches de convolution avec augmentation du dropout ou pour une couche de convolution par séquence avec deux fois moins de paramètres, il faut un nombre d'époque d'au moins 40 pour obtenir une précision convenable de 0.8. Donc le meilleur modèle que nous avons est celui de 40 époques avec deux couches de convolution par séquence et un taux de dropout croissant.
Avec celui-ci voici ses prédictions pour 9 images de la base cifar10 :

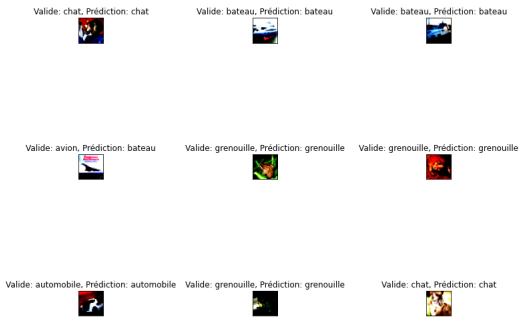


Figure 8: Résultats des prédictions de 9 images du meilleur modèle obtenu

3.2 Code pour la base de données personnelles

Nous avons donc dans un second temps essayé de créer une base de données en récoltant des images sur internet puis de créer un réseau de neurones basé sur le même schéma que celui utilisé précédemment mais en traitant les images que nous voulions. Pour cela nous avons fait plusieurs tests et nous parlerons ici du plus simple, celui basé sur des images de la lune et de saturne. Nous avons rassemblé en tout 200 images que nous avons uploadé dans un dossier sur la plate-forme github et créé une petite fonction pour pouvoir aller le chercher de n'importe où.

```
data_dir = tf.keras.utils.get_file("base.zip",
                                   "https://github.com/leoauguste/aaa/blob/main/base.zip?raw=true")

import zipfile
with zipfile.ZipFile(data_dir, 'r') as zip_ref:
    zip_ref.extractall('/content/datasets')

data_dir = pathlib.Path('/content/datasets/base')
```

Figure 9: Récupération de nos images

Il nous a fallu ensuite écrire une fonction pour mettre toutes les images dans le même format et pour les diviser en deux sous-parties, la partie entraînement et la partie validation.

Nous avons utiliser le réseau CNN utilisé avant, en y faisant quelques modifications dans l'espoir de le rendre plus performant mais sans réel succès comme nous le verrons plus loin.

```
batch_size = 3
img_height = 200
img_width = 200

train_data = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=42,
  image_size=(img_height, img_width),
  batch_size=batch_size,
  )

val_data = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=42,
  image_size=(img_height, img_width),
  batch_size=batch_size)

class_names = val_data.class_names
print(class_names)
```

Figure 10: Traitement des images

```
from tensorflow.keras import layers


num_classes = 2

model = tf.keras.Sequential([
    layers.experimental.preprocessing.Rescaling(1./255),
    layers.Conv2D(128,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(16,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(64,activation='relu'),
    layers.Dense(num_classes, activation='softmax')])

model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])

logdir="logs"

tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir,histogram_freq=0.9, write_images=logdir,
                                                   embeddings_data=train_data)

model.fit( train_data,validation_data=val_data,epochs=20,callbacks=[tensorboard_callback])
```

Figure 11: Réseau CNN modifié

La dernière partie du code nous sert juste à le tester. Nous prenons un autre dossier dans lequel se trouve des images de saturne et de la lune mélangées et nous les faisons passer par le CNN pour qu'il puisse nous dire à quelle catégorie ils appartiennent.

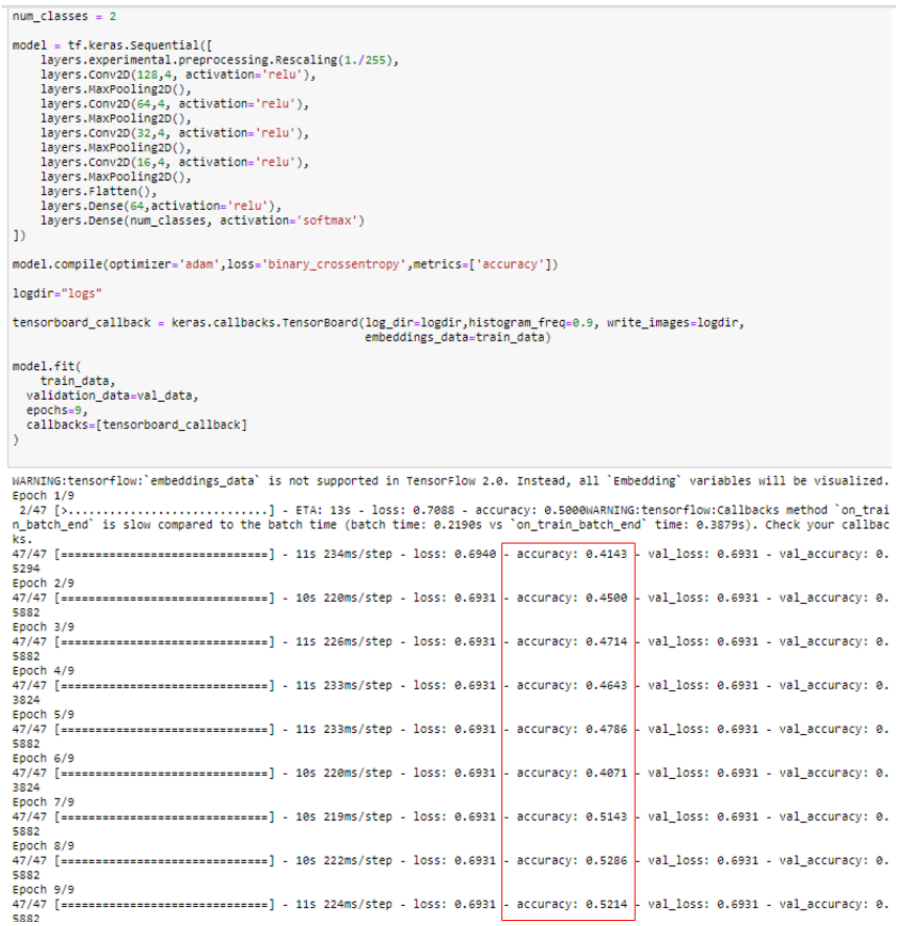
```
predict = tf.keras.utils.get_file("test.zip","https://github.com/leoauguste/aaa/blob/main/test.zip?raw=


with zipfile.ZipFile(predict, 'r') as zip_ref:
    zip_ref.extractall('/content/datasets')

predict = pathlib.Path('/content/datasets/test')

for file_ in predict:
    image_to_predict = cv2.imread(file_,cv2.IMREAD_COLOR)
    plt.imshow(cv2.cvtColor(image_to_predict, cv2.COLOR_BGR2RGB))
    plt.show()
    img_to_predict = np.expand_dims(cv2.resize(image_to_predict,(200,200)), axis=0)
    res = model.predict_classes(img_to_predict)
    print(model.predict_classes(img_to_predict))
    print(model.predict(img_to_predict))
    if res == 1:
        print(lune!)
    elif res == 0 :
        print(saturne!)
```

Figure 12: Programme nous permettant de tester notre CNN

Quand est-il du succès de notre expérience?

Nous ne somme pas très satisfaits du résultat final de ce deuxième test. En effet lorsque nous prenons des image assez similaire, par exemple saturne et la lune, nous n'arrivons pas à créer un CNN avec un taux de succès dépassant celui du hasard, c'est à dire avec une précision dépassant les 0,5 et ce même avec une 50ène d'époque. Nous avons eu beau essayer de faire varier les différents paramètres en jeu, ou de prendre de nouvelle fonction d'activation ou de perte, nous n'avons pas pus avoir de résultat décent. Pour obtenir des résultats un peu meilleur il nous a fallu prendre des images plus distinctes, comme par exemple des images d'arbres et de lune. Mais là encore nous n'étions pas capables de dépasser les 0.7 de précision.
Le nombre trop faible d'images pouvant aussi être une cause du mauvais fonctionnement de notre réseau, nous avons cependant pu constater lors de nos recherches que certains réseaux pouvaient

tout de même assez bien fonctionner avec autant d'images. Nous nous sommes donc demandés, en dernier lieu, si le problème ne venait pas du formatage des images lui même, puisque lorsque nous utilisons une banque de données construite au préalable nous arrivons à de bien meilleurs résultats. Cependant les bases de données d'images sont de vraies boites noires et il nous a été impossible de reproduire le type de formatage qu'elles subissent.



```
num_classes = 2

model = tf.keras.Sequential([
    layers.experimental.preprocessing.Rescaling(1./255),
    layers.Conv2D(128,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(16,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(64,activation='relu'),
    layers.Dense(num_classes, activation='softmax')
])

model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])

logdir="logs"

tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir,histogram_freq=0.9, write_images=logdir,
                                                   embeddings_data=train_data)

model.fit(
    train_data,
  validation_data=val_data,
  epochs=9,
  callbacks=[tensorboard_callback]
)
```

```
WARNING:tensorflow:`embeddings_data` is not supported in TensorFlow 2.0. Instead, all `Embedding` variables will be visualized.
Epoch 1/9
 2/47 [>.............................] - ETA: 13s - loss: 0.7088 - accuracy: 0.5000WARNING:tensorflow:Callbacks method `on_trai
n_batch_end` is slow compared to the batch time (batch time: 0.2190s vs `on_train_batch_end` time: 0.3879s). Check your callbac
ks.
47/47 [==============================] - 11s 234ms/step - loss: 0.6940 - accuracy: 0.4143 - val_loss: 0.6931 - val_accuracy: 0.
5294
Epoch 2/9
47/47 [==============================] - 10s 220ms/step - loss: 0.6931 - accuracy: 0.4500 - val_loss: 0.6931 - val_accuracy: 0.
5882
Epoch 3/9
47/47 [==============================] - 11s 226ms/step - loss: 0.6931 - accuracy: 0.4714 - val_loss: 0.6931 - val_accuracy: 0.
5882
Epoch 4/9
47/47 [==============================] - 11s 233ms/step - loss: 0.6931 - accuracy: 0.4643 - val_loss: 0.6931 - val_accuracy: 0.
3824
Epoch 5/9
47/47 [==============================] - 11s 233ms/step - loss: 0.6931 - accuracy: 0.4786 - val_loss: 0.6931 - val_accuracy: 0.
5882
Epoch 6/9
47/47 [==============================] - 10s 220ms/step - loss: 0.6931 - accuracy: 0.4071 - val_loss: 0.6931 - val_accuracy: 0.
3824
Epoch 7/9
47/47 [==============================] - 10s 219ms/step - loss: 0.6931 - accuracy: 0.5143 - val_loss: 0.6931 - val_accuracy: 0.
5882
Epoch 8/9
47/47 [==============================] - 10s 222ms/step - loss: 0.6931 - accuracy: 0.5286 - val_loss: 0.6931 - val_accuracy: 0.
5882
Epoch 9/9
47/47 [==============================] - 11s 224ms/step - loss: 0.6931 - accuracy: 0.5214 - val_loss: 0.6931 - val_accuracy: 0.
5882
```

Figure 13: Test du réseau CNN pour 9 époques avec des images de lune et de saturne. La précision de chaque époque est encadré en rouge

4 Annexe

4.1 Code pour la base de donnée personnel

```
import cv2
import numpy as np
import requests
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import sys
import datetime
from tensorflow import keras
from tensorflow.keras.models import Model
import tensorflow as tf
import pathlib
import os
```

```
data_dir = tf.keras.utils.get_file("base.zip",
                                   "https://github.com/leoauguste/banque-image/raw/main

import zipfile
with zipfile.ZipFile(data_dir, 'r') as zip_ref:
    zip_ref.extractall('/content/datasets')

data_dir = pathlib.Path('/content/datasets/base')




batch_size = 3
img_height = 200
img_width = 200

train_data = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=42,
  image_size=(img_height, img_width),
  batch_size=batch_size,
  )

val_data = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=42,
  image_size=(img_height, img_width),
  batch_size=batch_size)

class_names = val_data.class_names
print(class_names)




from tensorflow.keras import layers


num_classes = 2

model = tf.keras.Sequential([
    layers.experimental.preprocessing.Rescaling(1./255),
    layers.Conv2D(128,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(16,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(64,activation='relu'),
```

```
    layers.Dense(num_classes, activation='softmax')])

model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])

logdir="logs"

tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir,histogram_freq=0.9, w
                                                   embeddings_data=train_data)

model.fit( train_data,validation_data=val_data,epochs=20,callbacks=[tensorboard_callbac



predict = tf.keras.utils.get_file("test.zip","https://github.com/leoauguste/banque-imag


with zipfile.ZipFile(predict, 'r') as zip_ref:
    zip_ref.extractall('/content/datasets')

predict = pathlib.Path('/content/datasets/test')

for file_ in predict:
    image_to_predict = cv2.imread(file_,cv2.IMREAD_COLOR)
    plt.imshow(cv2.cvtColor(image_to_predict, cv2.COLOR_BGR2RGB))
    plt.show()
    img_to_predict = np.expand_dims(cv2.resize(image_to_predict,(200,200)), axis=0)
    res = model.predict_classes(img_to_predict)
    print(model.predict_classes(img_to_predict))
    print(model.predict(img_to_predict))
    if res == 1:
        print(lune!)
    elif res == 0 :
        print(saturne!)
```

4.2 Code pour la base de données Cifar10

Python 3.8.8

```
import numpy as np
import matplotlib.pyplot as plt
from keras import backend as K
from tensorflow import keras

#Import des donn es de cifar10
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

#Transformation des valeurs des pixels sur les [0,255] valeurs possibles de RGB d'entie
x_train=x_train.astype('float32')
x_test=x_test.astype('float32')

# Normalisation des donn es
    #Calcul des valeurs moyennes et l' cart -type
```

```
mean = np.mean(x_train, axis=(0,1,2,3))
sigma = np.std(x_train, axis=(0,1,2,3))
    #Normalisation (le 1e-7 vient du fait que nous voulons  viter  la division par 0)
x_train=(x_train-mean)/(sigma+1e-7)
x_test=(x_test-mean)/(sigma+1e-7)

#Hot encoding
NCLASSES=10 #le nombre de classe (dans cifar10 on a 'avion', 'bateau',...)
y_train=keras.utils.to_categorical(y_train,NCLASSES)
y_test=keras.utils.to_categorical(y_test,NCLASSES)

#Configuration du mod le :
def creationmodele():

    model=keras.models.Sequential()

    model.add(keras.layers.Conv2D(16, (3,3), padding='same', activation='elu',input_shap
    model.add(keras.layers.MaxPooling2D((2,2)))
    model.add(keras.layers.Dropout(0.2))

    model.add(keras.layers.Conv2D(32, (3,3), padding='same', activation='elu') )
    model.add(keras.layers.MaxPooling2D((2,2)))
    model.add(keras.layers.Dropout(0.2))

    model.add(keras.layers.Conv2D(64, (3,3), padding='same', activation='elu') )
    model.add(keras.layers.MaxPooling2D((2,2)))
    model.add(keras.layers.Dropout(0.2))

    model.add(keras.layers.Conv2D(128, (3,3), padding='same', activation='elu') )
    model.add(keras.layers.MaxPooling2D((2,2)))
    model.add(keras.layers.Dropout(0.2))

    model.add(keras.layers.Flatten())
    model.add(keras.layers.Dense(256, activation='elu'))

    model.add(keras.layers.Dense(NCLASSES, activation='softmax'))

    return model

K.clear_session()
model=creationmodele()

#Param tres utilis  par le mod le et son r sum  pour voir ce qu'il contient
model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
model.summary()

#Apprentissage du mod le
    #D finition du nombre d' poque  et de la taille des batch
batch_size=200
epochs=20
    #Apprentissage
history = model.fit(x_train, y_train,batch_size=batch_size,epochs=epochs,verbose= 0,vali

#Graphe de la perte ('loss') et de la pr cision ('accuracy') du mod le
    #Pr cision
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Pr cision du mod le')
plt.ylabel('Pr cision')
```

```
plt.xlabel('Epoque')
plt.legend(['Apprentissage', 'Validation'])
plt.show()
    #Perte
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Perte du mod le')
plt.ylabel('Perte')
plt.xlabel('Epoque')
plt.legend(['Apprentissage', 'Validation'])
plt.show()

#Affichages des images ainsi que les pr dictions du mod les associ es    leur classen
prediction=model.predict(x_test[0:9])
y_classe_valide = np.argmax(y_test[0:9], axis=1)
y_classe_predite = np.argmax(prediction, axis=1)
classe_labels =["avion", "automobile","oiseau","chat","cerf","chien","grenouille","chev

fig, axes = plt.subplots(3, 3, figsize=(10,10))
fig.subplots_adjust(hspace=0.5, wspace=6)

for i, ax in enumerate(axes.flat):
  ax.imshow(x_test[i])
  xtitle = "Valide: {0}, Pr diction: {1}".format(classe_labels[y_classe_valide[i]], cla
  ax.set_title(xtitle)
  ax.set_xticks([])
  ax.set_yticks([])

plt.show()
```