

Filière MP - ENS de Paris-Saclay, Lyon, Rennes et Paris - Session 2021

Page de garde du rapport de TIPE

NOM :	QUEMENER	Prénoms :	Maena
Classe :	MP*		
Lycée :	Lycée Hoche	Numéro de candidat :	30496
Ville :	Versailles		

Concours auxquels vous êtes admissible, dans la banque MP inter-ENS (les indiquer par une croix) :

ENS Cachan	MP - Option MP		MP - Option MPI	
	Informatique	X		
ENS Lyon	MP - Option MP		MP - Option MPI	
	Informatique - Option M	X	Informatique - Option P	
ENS Rennes	MP - Option MP		MP - Option MPI	
	Informatique			
ENS Paris	MP - Option MP		MP - Option MPI	
	Informatique			

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

Informatique	X	Mathématiques		Physique	
--------------	---	---------------	--	----------	--

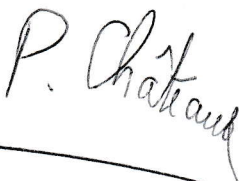
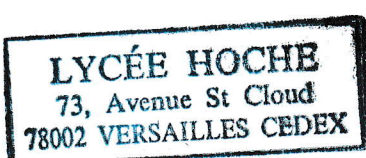
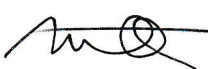
Titre du TIPE : Réseaux de neurones et traitement d'images appliqués à la détection d'incendie

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	7	Illustration	3	Bibliographie	1
-------	---	--------------	---	---------------	---

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :

Dans le cadre de la détection d'incendie, je me suis penchée sur le problème de classification d'images avec les catégories feu/non feu. Pour cela j'ai développé un réseau de neurones convolutif reposant sur la structure du modèle VGG16. Je l'ai entraîné sur une base de données adaptée à l'aide de différentes méthodes telles *que le* transfer-learning ou fine-tuning. J'ai comparé les résultats obtenus avec la méthode d'entraînement classique. Finalement, j'ai exploré quelques pistes d'améliorations potentielles de notre modèle.

À	Versailles	Signature du professeur responsable de la classe préparatoire dans la discipline	Cachet de l'établissement
Le	08/06/21		
Signature du (de la) candidat(e)			
La signature du professeur responsable et le tampon de l'établissement ne sont pas indispensables pour les candidats libres (hors CPGE).			

Réseaux de neurones et traitement d'images appliqués à la détection d'incendie

Maena QUEMENER

2021

J'ai travaillé, dans le cadre de la détection d'incendie, sur le développement d'un réseau de neurones convolutifs (CNN) pour le problème de classification d'images.

1 Introduction

Aujourd'hui, les principaux outils utilisés dans la détection d'incendie sont des capteurs physiques, adaptés en intérieurs mais pas pour de grandes étendues comme des forêts. Ainsi, dans le cadre d'un TIPE sur la détection d'incendie, il était intéressant de réfléchir à d'autres outils adaptés pour ce genre d'espace. L'imagerie est particulièrement efficace pour la surveillance de larges zones. J'ai donc choisi de traiter le problème de classification d'images dans le cadre de la détection d'incendie. J'ai développé pour cela un réseau de neurones convolutifs que j'ai entraîné sur une base de données d'images de feu/non feu.

Problématique :

- Trouver un modèle pertinent pour le problème de classification d'images.
- Développer un réseau de neurones capable d'atteindre une bonne précision dans le cadre de la détection d'incendie.

2 Réseaux de neurones

Un problème de classification peut être modélisé par une fonction mathématique souvent trop compliquée pour être explicitée et calculée. On peut cependant tenter de l'approximer à l'aide de modèles comme les réseaux de neurones. Les réseaux de neurones consistent en un empilement de « neurones » totalement ou partiellement reliés entre eux.

2.1 Perceptron

Pour bien prendre en main le concept de neurones et réseaux de neurones, j'ai dans un premier temps implémenté un perceptron en Python.

Le perceptron est le type de réseau de neurones le plus élémentaire : il est composé d'un unique neurone (description d'un neurone en annexe). L'intérêt du modèle du perceptron est la phase d'apprentissage, durant cette phase, les poids synaptiques vont être mis à jour à l'aide de la méthode de gradient.

Principe :

On définit une fonction d'erreur (aussi dit de coût) qui va permettre de mesurer l'écart entre le résultat renvoyé par le réseau et le résultat attendu. Le but est de minimiser cette fonction, afin d'avoir des résultats de plus en plus correct.

La fonction choisie ici pour évaluer l'erreur est l'entropie croisée définie de cette manière :

$$E : (x, y) \mapsto -y \log(f(x)) - (1 - y) \log(1 - f(x))$$

avec :

- $x = (x_1, \dots, x_n)$ l'entrée.
- y la sortie attendue.
- f la fonction du neurone défini précédemment.

Pour minimiser cette fonction, on utilise la méthode du descente de gradient. L'idée est que l'on va approcher « par petit pas » un minimum de la fonction d'erreur. Pour cela, on va corriger chaque poids synaptique en calculant le gradient selon la formule suivante :

$$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i}(f(x), y) = w_i - \eta(y - f(x))x_i$$

avec :

- w_i le poids qu'on actualise.
- f la fonction d'erreur.
- η est le taux d'apprentissage. Cette variable permet que les modifications apportées à chaque actualisation ne soit pas trop importante, afin de se stabiliser autour d'un minimum.

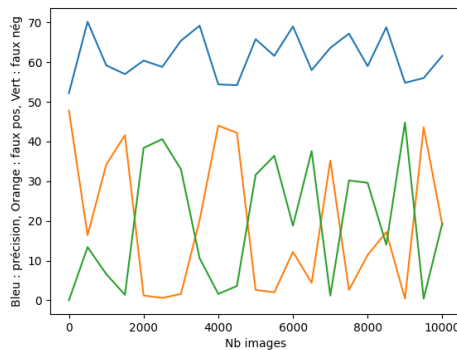
Ainsi, la phase d'entraînement va consister à répéter ce procédé pour toutes les images de notre base de données.

Limitations :

Le perceptron est en pratique un modèle très limité, car c'est un classifieur linéaire. En particulier, les classifieurs linéaires ne peuvent pas distinguer les formes, mais uniquement les couleurs. Un perceptron serait donc incapable par exemple de différencier une forêt en automne d'un feu de forêt.

Résultats :

J'ai donc entraîné un perceptron implémenté en Python (codes en annexe), avec le choix de la fonction sigmoïde pour la fonction d'activation et de l'entropie croisée introduite plus haut pour la fonction d'erreur. Le graphe suivant montre l'évolution de ses résultats, avec comme paramètre $\eta = 0.001$.



- La courbe bleu représente le pourcentage de précision (nombre d'images correctement classifiés par rapport à l'ensemble des images d'évaluation).
- La courbe orange représente la proportion de faux-positifs (images classées *feu* alors qu'elles sont *non feu*).
- La courbe verte représente la proportion de faux-négatifs.

On constate que les résultats du modèle ne convergent effectivement pas vers un résultat intéressant (le modèle ne dépasse jamais les 70% de précision) et qu'il est assez instable.

Je suis donc assez rapidement passée à l'étude d'un réseau de neurones convolutif pour avoir des résultats plus pertinents.

2.2 Réseau de neurones convolutif

J'ai ensuite implémenté un réseau de neurones convolutif (abrégié CNN pour Convolutional Neural Network) pour résoudre ce problème. Le principal avantage d'un CNN par rapport à un perceptron est que ce n'est pas un modèle linéaire, il peut donc résoudre des problèmes plus complexes.

Son fonctionnement de base est inspiré par le cortex visuel des animaux. Lors de l'analyse d'une image, le cerveau des animaux va découper le champ visuel en petites zones qui vont chacune être analysées individuellement par les neurones, puis les données récupérées vont être réassemblées par la suite. C'est cette notion d'analyse *locale* de l'image, réalisée lors de l'étape de « convolution », qui caractérise les CNN.

Un réseau de neurones est composé de « couches ». Une couche est un assemblage de neurones qui vont traiter une entrée appelée *volume d'entrée* et renvoyer un *volume de sortie* qui sera traité par les couches suivantes. On parle de volume pour souligner qu'on ne travaille généralement pas sur une unique matrice de réels, mais sur un « empilement de matrices », appelé tenseur (par exemple, j'ai travaillé avec des images rgb de taille 100×100 , donc le volume d'entrée est de taille $100 \times 100 \times 3$ car on a une matrice pour chaque couleur R/G/B).

Un CNN est composé de trois types de couches caractéristiques :

- Les couches de convolutions.
- Les couches de pooling.
- Les couches totalement connectées (abrégées couches FC pour fully-connected).

Ces couches sont alternées de la manière suivante :

Entrée \rightarrow Convolutions \rightarrow Pooling \rightarrow Convolutions \rightarrow Pooling $\rightarrow \dots \rightarrow$ Couches FC \rightarrow Sortie

Le fonctionnement de ces couches est expliqué en détail en annexe.

Méthode d'apprentissage

La méthode d'apprentissage repose également sur la mise à jour des poids par méthode du gradient. On parle cependant généralement de méthode de rétropropagation du gradient.

En effet, pour diminuer le nombre de calculs, on calcule le gradient des poids en partant de la sortie du réseau jusqu'à l'entrée du réseau. Ainsi grâce à la règle de la chaîne pour les fonctions composées, on va pouvoir réutiliser le calcul des gradients d'une couche pour le calcul des gradients de la couche qui précède dans le réseau.

Règle de la chaîne

$$\frac{\partial E}{\partial w_{i,j}} = \sum_{k=1}^n \frac{\partial E}{\partial w_{k,j+1}} \frac{\partial w_{k,j+1}}{\partial w_{i,j}}$$

avec :

- $w_{i,j}$ le i -ème poids de la j -ème couche.

Dans la pratique, je n'ai pas utilisé une simple méthode de descente de gradient qui est assez peu efficace, mais des méthodes d'optimisation (optimiseur Adam (1)) basées sur le momentum. Le momentum consiste à garder en mémoire les valeurs des différents gradients calculés à des dates antérieures pour un poids donné avec une *moyenne mobile exponentielle*. Cela va permettre de suivre une direction plus globale et de ne pas aller juste où la pente est la plus raide. Cette méthode accélère la convergence de la fonction d'erreur vers un minimum intéressant et permet généralement d'éviter les minimums locaux.

3 Implémentation et résultats

3.1 Base de données

La qualité d'une base de données est aussi importante que la qualité d'un réseau de neurones. Ainsi pour la mise en pratique de toutes ces notions, la première étape a été de construire une base de données fournies et de bonne qualité. Cette étape a pris autant de temps que l'implémentation du réseau. Cela souligne l'un des obstacles principaux des réseaux de neurones : la nécessité d'avoir accès à un grand nombre de données sur le problème considéré. Je voulais initialement travailler avec des images infrarouges de feu qui auraient permis une détection encore plus précoce des départs de feu. Mais par manque de ressources, j'ai finalement opté pour des images de feux classiques.

Pour construire ma base de données, j'ai donc récupéré un grand nombre de bases de données déjà formées sur internet que j'ai fusionnées. Puis j'ai codé des scripts en Bash afin de pouvoir convertir l'ensemble des images au format .png et les redimensionner à la taille 300×200 , et ainsi réduire les temps de calculs lors de l'entraînement des réseaux.

J'ai ainsi récupéré et prétraité environ 40000 images, divisées en un set d'entraînement d'environ 30000 images (18000 de feu et 13000 de non feu) et en un set de vérification de 10000 images.

3.2 Implémentation

Pour l'implémentation de mes CNN, j'ai utilisé la librairie Tensorflow développée pour le machine learning en Python, et plus spécifiquement les modules Keras. Ceux-ci permettent d'utiliser des réseaux de neurones sans avoir à tout réimplémenter.

Mon but étant de développer un modèle efficace, il ne me paraissait pas pertinent de tenter de reconstruire une structure de CNN de zéro. J'ai donc réutilisé la structure du réseau VGG16. VGG16 est un réseau qui atteint 92.5 % de précision sur la base de données ImageNet (plusieurs millions d'images). Il a été développé en 2014, mais reste aujourd'hui encore un modèle très performant.

Deux éléments sont important pour l'entraînement d'un réseau :

- La puissance de calcul/le temps d'entraînement.
- La qualité de la base de données.

Je disposais d'une base de données d'environ 30000 images réservées à l'entraînement et d'un ordinateur peu puissant pour les calculs. Ainsi un entraînement de zéro était possible, mais pas forcément très efficace.

La base de données ImageNet est une base de données de référence et le réseau VGG16 a été entraîné dessus durant plusieurs semaines. Les poids de l'entraînement sont publics, il était donc intéressant de trouver un moyen d'exploiter ces poids pour adapter le réseau à un nouveau problème de classification.

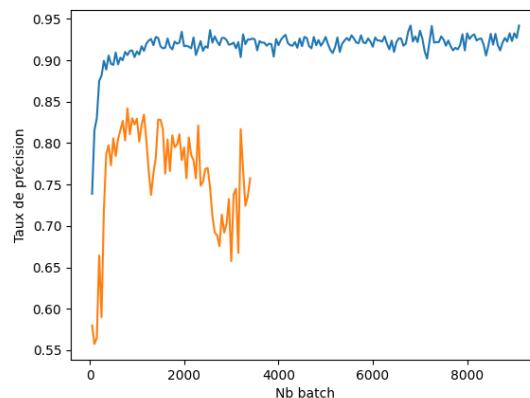
3.2.1 Transfer-learning

Le transfer learning est justement un problème de recherche qui consiste à utiliser les connaissances acquises par un réseau sur un problème donné pour résoudre un problème différent mais similaire (en l'occurrence ici, la classification d'images, mais pour des catégories différentes). D'après (2), pour un entraînement utilisant le transfer-learning, les résultats pour la précision sont meilleurs initialement, le modèle converge plus vite vers de bons résultats et l'asymptote est meilleure que pour un entraînement depuis zéro.

Dans le cadre des CNN, pour mettre en oeuvre un transfer-learning, l'idée est que l'on va geler les couches du modèle entraîné et que l'on va rajouter un coucou au début et changer le classifieur pour qu'il corresponde à notre problème (j'ai donc choisi un classifieur binaire, car je n'avais que deux catégories).

Lors de l'entraînement, uniquement la couche du dessus va être modifiée, cette couche permet la détection de caractéristiques basiques comme les formes ou les couleurs. Les couches du milieu ont été entraînées à reconnaître des caractéristiques précises lors de son entraînement initial, et nous les gelons pour conserver cette aptitude.

J'ai donc implémenté deux réseaux reposants sur la structure VGG16 : un réseau entraîné par méthode de transfer-learning et un réseau qui entraîne de manière classique, avec des poids initialisés de manière aléatoire, et je les ai entraînés sur ma base de données feu/non feu.



En orange, ce sont les résultats de la méthode classique, et en bleu de la méthode par transfer-learning.

Les entraînements pour les deux réseaux ont été lancés en même temps, sur la même base de données, avec les mêmes paramètres. Ainsi cela met en évidence deux avantages principaux de la méthode de transfer-learning :

- L'entraînement d'un réseau par transfer-learning est beaucoup plus rapide que par la méthode classique.
- Les résultats obtenus sont bien meilleurs.

De plus, le nombre de poids à entraîner étant moins élevé, cette méthode est également adaptée pour des plus petites bases de données.

J'ai ensuite évalué ces deux réseaux sur une base de données de 10000 images indépendantes de la base de données d'entraînement. Les résultats obtenus au terme de l'évolution sont les suivants :

	Transfer learning	De zéro
Taux de précision	91,0 %	80,9 %

3.2.2 Fine-tuning







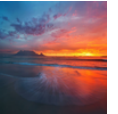
Une fois le réseau entraîné par transfer-learning, on peut encore essayer de l'améliorer un peu, par une méthode appelée fine-tuning. En effet, les différentes couches du réseau ont été entraînées indépendamment. Le fine-tuning va consister à dégeler toutes les couches, et à entraîner une dernière fois le réseau avec un **très faible taux d'apprentissage** afin d'homogénéiser l'ensemble du réseau sans détruire tout l'entraînement acquis avec des modifications trop brusques.

Les résultats obtenus au terme du fine-tuning ont vraiment été concluant, ils ont permis de monter à une précision de 94 %.

	De zéro	T-L	F-T
Taux de précision	80,9 %	91,0 %	94,3 %

Une variante possible du fine-tuning est de dégeler uniquement les dernières couches du réseau. En effet, dans la pratique, les premières couches (i.e couches les plus près de l'entrée) sont les couches les plus générales. Elles peuvent reconnaître des caractéristiques utiles à un grand nombre de problèmes (comme des formes : courbes, pointes, etc...), tandis que les dernières couches identifient des caractéristiques plus abstraites et spécifiques au problème. De plus les dernières couches sont proches de la sortie, donc ont une influence plus importante sur le résultat final. C'est pourquoi qu'il n'est pas forcément intéressant de toucher aux premières couches, mais seulement aux dernières pour adapter un réseau à un nouveau problème.

3.3 Exemples de résultats

Images							
De zéro	0.85	-0.28	0.17	-0,67	1.16	-0.30	-0.15
T-L	-0.74	1.78	-9.19	-0.65	0.40	-0.51	13.27
F-T	-3.06	-2.84	-13.04	1.30	3.74	-0.45	8.94

Un score négatif correspond à une image *feu* et positif *non feu*. Plus le score est important, plus le réseau détecte l'image comme étant d'une catégorie ou d'une autre. J'ai pu observer sur un échantillon testé à la main (et comme l'illustre le tableau ci-dessus) que le réseau détecte beaucoup mieux la présence de fumée que de feu. Une des explications les plus plausibles est un déséquilibre dans la base de données que j'ai constitué entre les images comportant principalement de la fumée et celle uniquement des flammes. Cependant, dans la pratique, de tels résultats sont préférables, car dans le cadre de la détection d'incendie, il est plus intéressant d'être capable de détecter de la fumée, plutôt que des flammes.

Plusieurs pistes peuvent être explorées pour améliorer le modèle, comme rajouter de couches de normalisation, ou encore essayer de faire varier les hyperparamètres pour trouver les plus optimaux et essayer différentes méthodes d'entraînement (comme la deuxième méthode de fine-tuning expliquée par exemple). Cependant je pense que le principal obstacle afin de pouvoir améliorer encore davantage les résultats est de réussir à former une base de données fournies, avec une grande variété d'images.

4 Annexe

4.1 Neurone formel

Le neurone est la brique élémentaire des réseaux de neurones. Ils vont généralement être assemblés sous forme de couches reliées totalement ou partiellement entre elles.

Mathématiquement, un neurone est une fonction de plusieurs variables dans \mathbb{R} :

$$f : (x_1, \dots, x_n) \in \mathbb{R}^n \mapsto \varphi \left(\sum_{i=1}^n w_i x_i + w_0 \right)$$

avec $\varphi : x \mapsto \frac{1}{1+e^{-x}}$.

- (x_1, \dots, x_n) sont les entrées. Ici, cela peut par exemple être la valeur des pixels de l'image d'entrée.
- (w_1, \dots, w_n) sont les poids synaptiques. Ces poids vont être mis à jour lors de la phase d'apprentissage d'un réseau de neurones.
- w_0 est appelé *biais*.
- φ est la fonction d'activation. Elle permet d'obtenir un modèle non-linéaire.

4.2 CNN

Convolution

Les couches de convolution sont l'élément clef des CNN. Ces couches agissent comme des filtres qui extraient des caractéristiques de l'image (une forme en pointe pour des flammes par exemple, ou des zones grises pour de la fumée).

Une couche de convolution est un empilement de plusieurs filtres appelés *noyaux de convolutions*. Un noyau est un assemblage de neurones et extrait de l'image une caractéristique donnée bien précise. Tous les neurones d'un même noyau ont le même poids et chaque neurone traite uniquement une petite zone de l'image. C'est ce traitement local qu'on appelle convolution. Il y a invariance du traitement de l'image par translation, ce qui est intéressant car dans notre cas, le feu peut aussi bien se trouver en haut à gauche, que au milieu ou en bas à droite de l'image par exemple.

Chaque noyau va renvoyer une matrice de réels qui caractérisent les éléments identifiés et ces matrices vont être assemblées pour créer le volume d'entrée des couches suivantes.

Pooling

Les couches de pooling s'intercalent entre les couches de convolution pour éliminer le bruit.

Le volume d'entrée est découpée en petite zone et l'étape de pooling va consister à ne conserver de chaque petite zone que l'information la plus importante. Cela va avoir pour effet de diminuer la taille du volume d'entrée et d'enlever le « bruit ».

Deux grandes opérations de pooling existent :

- le max-pooling, on ne va garder que la plus grande valeur d'une zone.

1	10	7	3
3	8	9	0
18	0	0	0
4	1	3	2

 →

10	9
18	3

- le average-pooling, on va faire la moyenne des valeurs d'une zone.

1	10	7	3
3	8	9	0
18	0	0	0
4	1	3	2

 →

5.5	4.75
5.75	1.25

Dans tous les modèles que j'ai implémenté, c'est le max-pooling qui est utilisé.

Couches totalement connectées, dit couches FC

Après l’alternance des couches de convolution et de pooling, viennent les couches FC (fully-connected). Le rôle de ces couches est de faire une analyse globale des caractéristiques extraites lors des convolutions. Pour cela, chaque neurone d’une couche est relié à tous les neurones de la couche suivante, et les poids de chaque neurone est indépendant de ceux des autres neurones de la couche (contrairement aux couches de convolution).

A l’issu de ces couches, les résultats vont être envoyés à un classifieur qui va renvoyer un réel indiquant la catégorie de l’image.

Bibliographies

- [1] Diederik P KINGMA et Jimmy LEI BA : Adam : A method for stochastic optimization. *Conference paper at ICLR 2015*.
- [2] Emilio Soria OLIVAS, Jose David Martin GUERRERO, Marcelino Martinez SOBER, Jose Rafael Magdalena BENEDITO et Antonio Jose Serrano LOPEZ : *Handbook Of Research On Machine Learning Applications and Trends : Algorithms, Methods and Techniques*. 2009.

Liste des codes

1	Fonctions principales du perceptron	8
2	Exemple d’initialisation d’un réseau pour le transfer-learning	9
3	Prétraitement des images de la base de données	9
4	Entraînement d’un réseau	9


```

def poids_init():
    """Fonction pour créer une nouvelle grille de poids."""

    print("Initialisation des poids")

    poids_act = [[[random.random(),random.random(),random.random()] for _ in range(300)] for _ in range(300)]
    l = pickle.load(open('poids','rb'))
    l.append(poids_act)
    pickle.dump(l,open('poids','wb'))

    s = pickle.load(open('stat','rb'))
    s.append([verif(poids_act)])
    pickle.dump(s,open('stat','wb'))

def calcul_res(img,poids):
    """Calcul le poids d'une image."""
    sum = 1
    for i in range(200):
        for j in range(300):
            for c in range(3):
                sum += (img[i,j,c]/255)*poids[i][j][c]
    return sum

def weight_update(poids,img,erreur):
    """Met à jour la grille de poids en cours d'entraînement en fonction de l'erreur de l'image."""
    for i in range(200):
        for j in range(300):
            for c in range(3):
                poids[i][j][c] = poids[i][j][c] + erreur * img[i,j,c] * 0.001
    return poids

def entraînement(poids):
    """Entraîne une grille de poids à l'aide de toutes les fonctions définies plus tôt, pour une image."""
    type = random.randint(0,1)
    if type == 1:
        num = random.randint(0,18577)
        lien = 'dataset/TRAIN/fire/'+str(num)+'.png'
    else:
        num = random.randint(0,13757)
        lien = 'dataset/TRAIN/non_fire/'+str(num)+'.png'

    img = cv2.imread(lien)

    erreur = 1

    while abs(erreur) > 0.1:
        res = calcul_res(img,poids)
        val = fc_sigmoide(res)
        erreur = type - val

        poids = weight_update(poids,img,erreur)

        erreur = type - val

    return poids

```

Listing 1: Fonctions principales du perceptron

```

base_model = keras.applications.VGG16(
    include_top=False, #On ne garde pas le classifieur de ImageNet (1000 catégories)
    weights='imagenet', # On récupère les poids entraînés sur ImageNet.
    input_shape=(100, 100, 3)
)

base_model.trainable = False #On gèle les couches entraînées.

inputs = keras.Input(shape=(100,100,3))

x = base_model(inputs, training=False)

x = keras.layers.GlobalAveragePooling2D()(x)
#On choisit un classifieur binaire.

outputs = keras.layers.Dense(1)(x)
#La sortie sera une valeur unique.

model = keras.Model(inputs, outputs)

model.compile(optimizer=keras.optimizers.Adam(),
              loss=keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=[keras.metrics.BinaryAccuracy()])

```

Listing 2: Exemple d'initialisation d'un réseau pour le transfer-learning

```

train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    "dataset/TRAIN",
    labels='inferred',
    label_mode='binary',
    class_names=None,
    color_mode='rgb',
    batch_size=32,
    image_size=(100,
100),
    shuffle=True,
    seed=None,
    interpolation='bilinear'
)

```

Listing 3: Prétraitement des images de la base de données

```

checkpoint_filepath = 'checkpoint/model_1'

cp_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=False,
    monitor='binary_accuracy',
    mode='max',
    save_best_only=True,
    save_freq = 'epoch')

model = keras.models.load_model('checkpoint/model_1')

history = model.fit(train_ds, epochs=10, steps_per_epoch=50, callbacks=[cp_callback])

```

Listing 4: Entraînement d'un réseau