Université Mohammed V
Faculté des Sciences
Rabat

Techniques avancées de conception et de manipulation des réseaux de neurones profonds

Mehdi Bouskri

mehdi_bouskri@um5.ac.ma

Introduction

- Le modèle séquentiel suppose que le réseau possède exactement une seule entrée et une seule sortie, et qu'il est constitué d'une pile linéaire de couches.

- Certains réseaux nécessitent plusieurs entrées indépendantes, d'autres plusieurs sorties, et certains réseaux ont des branches internes entre les couches

Introduction

- Certaines tâches fusionnent des données provenant de différentes sources d'entrée, en traitant chaque type de données à l'aide de différents types de couches neuronales.

- De même, certaines tâches nécessitent de prédire plusieurs attributs cibles de certaines données d'entrée.

- De nombreuses architectures de neurones nécessitent une topologie de réseau non linéaire.

Introduction

- Ces trois cas d'utilisation importants - modèles à entrées multiples, modèles à sorties multiples et modèles en forme de graphe - ne sont pas possibles si l'on utilise uniquement la classe de modèles séquentiels de Keras.

- L'API fonctionnelle est une autre façon, beaucoup plus générale et flexible, d'utiliser Keras.

l'API fonctionnelle de Keras

Dans l'API fonctionnelle, on manipule directement les tenseurs et on utilise les couches comme des fonctions qui prennent les tenseurs et les renvoient.

```
from keras import Input, layers

input_tensor = Input(shape=(64,))
dense = layers.Dense(64, activation='relu')
outpu_tensor = dense(input_tensor)
```

l'API fonctionnelle de Keras

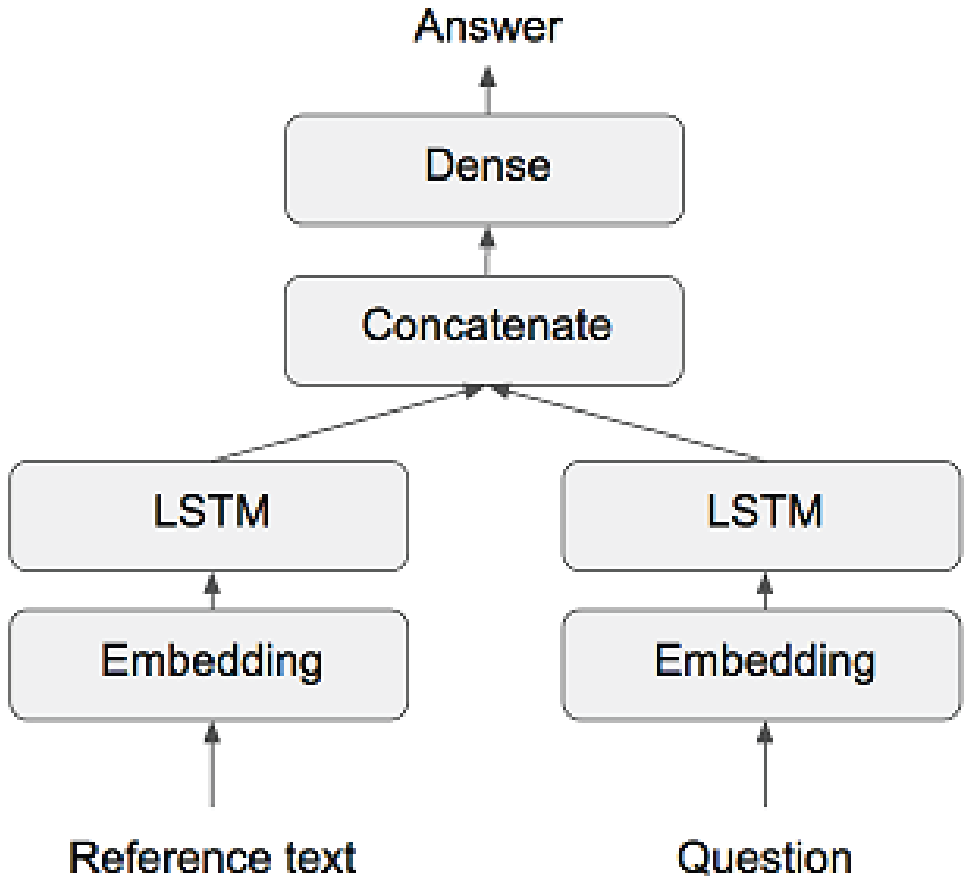
```
from keras.models import Sequential, Model

seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
api_model = Model(input_tensor, output_tensor)
```

```
api_model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
api_model.fit(x_train, y_train, epochs=10, batch_size=128)
```

Modèles à entrées multiples

L'API fonctionnelle peut être utilisée pour construire des modèles avec plusieurs entrées, ces modèles seront combinés via une opération de concaténation.



Modèles à entrées multiples

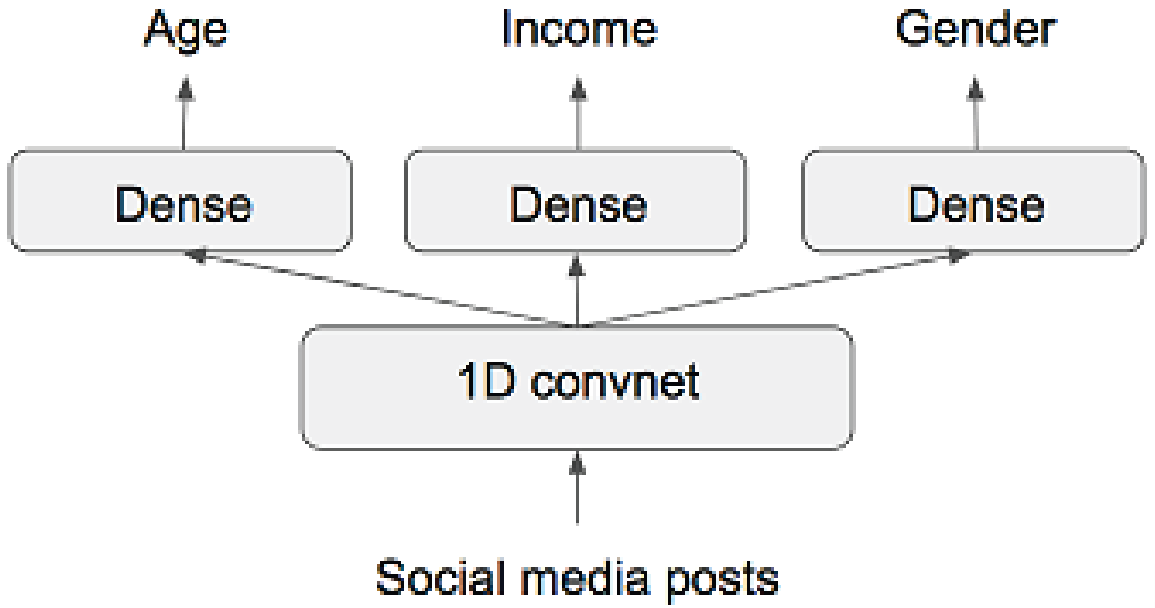
```
text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int32', name='text')
embedded_text = layers.Embedding(64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shapee=(None,), dtype='int32', name='question')
embedded_question = layers.Embedding(32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question], axis=-1)

answer = layers.Dense(answer_vocabulary_size, activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['acc'])
```

Modèles à entrées multiples

```
import numpy as np

text = np.random.randint(1, text_vocabulary_size, size=(num_samples, max_length))
question = np.random.randint(1, question_vocabulary_size, size=(num_samples, max_length))

answers = np.random.randint(0, 1, size=(num_samples, answer_vocabulary_size))

model.fit([text, question], answers, epochs=10, batch_size=128)

model.fit(('text', text, 'question', question), answer, epochs=10, batch_size=128)
```

Modèles à sorties multiples

De la même manière, l'API fonctionnelle peut être utilisée pour construire des modèles avec des sorties multiples.



Modèles à sorties multiples

```
vocabulary_size = 50000
num_income_groups = 10

posts_input = Input(shape=(None,), dtype='int32', name='posts')
embedded_posts = layers.Embedding(256, vocabulary_size)(posts_input)
x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation='relu')(x)

age_prediction = layers.Dense(1, name='age')(x)
income_prediction = layers.Dense(num_income_groups, activation='softmax', name='income')(x)
gender_prediction = layers.Dense(1, activation='sigmoid', name='gender')(x)

model = Model(input_posts, [age_prediction, income_prediction, gender_prediction])
```

Modèles à sorties multiples

L'apprentissage d'un tel modèle nécessite la spécification de différentes fonctions de perte pour chaque sortie du réseau.

```
model.compile(optimizer='rmsprop', loss=['mse', 'categorical_crossentropy',
                                         'binary_crossentropy'])

model.compile(optimizer='rmsprop',loss={'age': 'mse', 'income': 'categorical_crossentropy',
                                        'gender': 'binary_crossentropy'})
```

Modèles à sorties multiples

il est également possible d'attribuer des poids différents aux valeurs de perte dans leur contribution à la perte finale.

```
model.compile(optimizer='rmsprop', loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'],
              loss_weights=[0.25, 1., 10.])

model.compile(optimizer='rmsprop', loss={'age': 'mse','income': 'categorical_crossentropy',
                                         'gender': 'binary_crossentropy'},
              loss_weights={'age': 0.25,'income': 1.,'gender': 10.})
```

Modèles à sorties multiples

Comme dans le cas des modèles à entrées multiples, il est possible de transmettre des données Numpy au modèle pour l'entraînement, soit via une liste de tableaux, soit via un dictionnaire de tableaux.

```
model.fit(posts, [age_targets, income_targets, gender_targets],
 epochs=10, batch_size=64)

model.fit(posts, {'age': age_targets,'income': income_targets,'gender': gender_targets},
          epochs=10, batch_size=64)
```

Graphes acycliques dirigés de couches

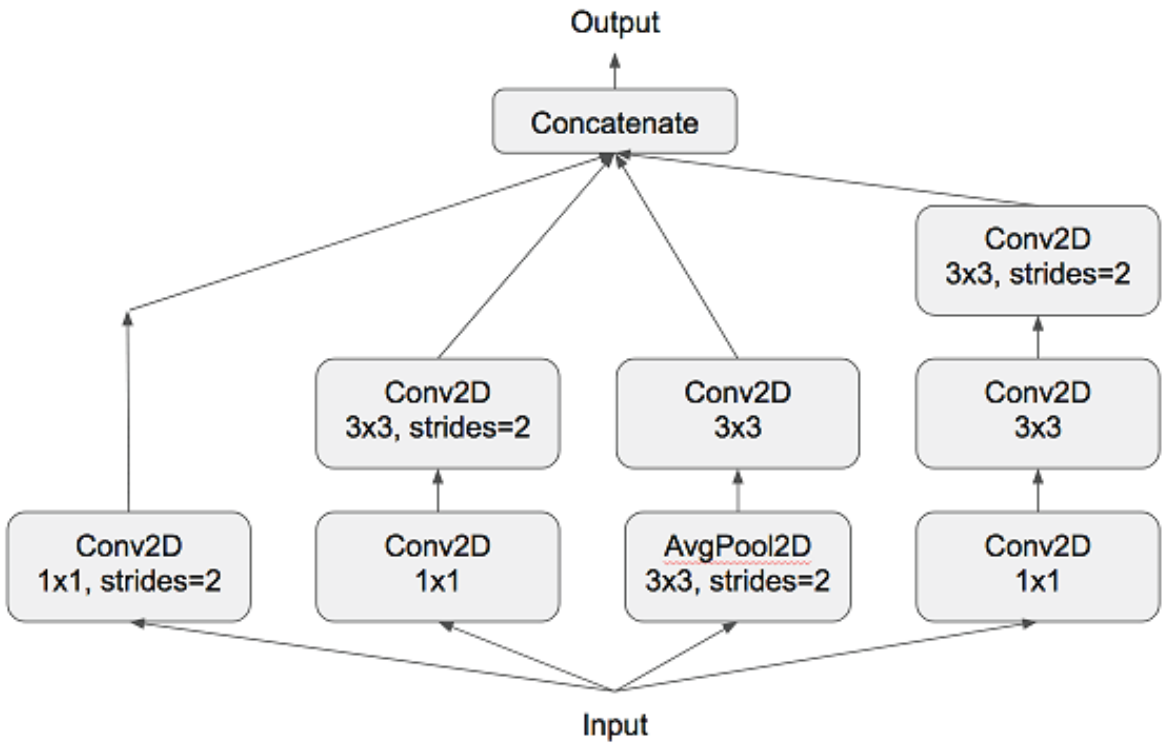
Avec l'API fonctionnelle, on peut construire des modèles avec plusieurs entrées et plusieurs sorties, mais aussi des réseaux avec une topologie interne complexe.

Plusieurs composants fréquents de réseaux de neurones sont implémentés sous forme de graphes. Les modules d'inception et les connexions résiduelles sont deux exemples importants.

Inception modules

Ils sont constitués d'un assemblage de modules qui eux-mêmes ressemblent à de petits réseaux indépendants, divisés en plusieurs branches parallèles.

Cette configuration permet au réseau d'apprendre séparément les caractéristiques spatiales et les caractéristiques par canal, ce qui est plus efficace que de les apprendre conjointement.



Inception modules

```
branch_a = layers.Conv2D(128, 1, activation='relu', strides=2)(x)

branch_b = layers.Conv2D(128, 1, activation='relu')(x)
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_b)

branch_c = layers.AveragePooling2D(3, strides=2, activation='relu')(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)

branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3, activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_d)

output = layers.concatenate([branch_a, branch_b, branch_c, branch_d], axis=-1)
```

Inception modules

Notez que l'architecture complète d'Inception V3 est disponible dans Keras sous le nom de keras.applications.inception_v3.InceptionV3, y compris les poids préentraînés sur les données ImageNet.

Un autre modèle très proche, disponible dans le module Keras est "Xception", qui signifie "extreme inception".

Cette méthode permet de séparer l'apprentissage des caractéristiques par canal et par espace.

Connexions résiduelles

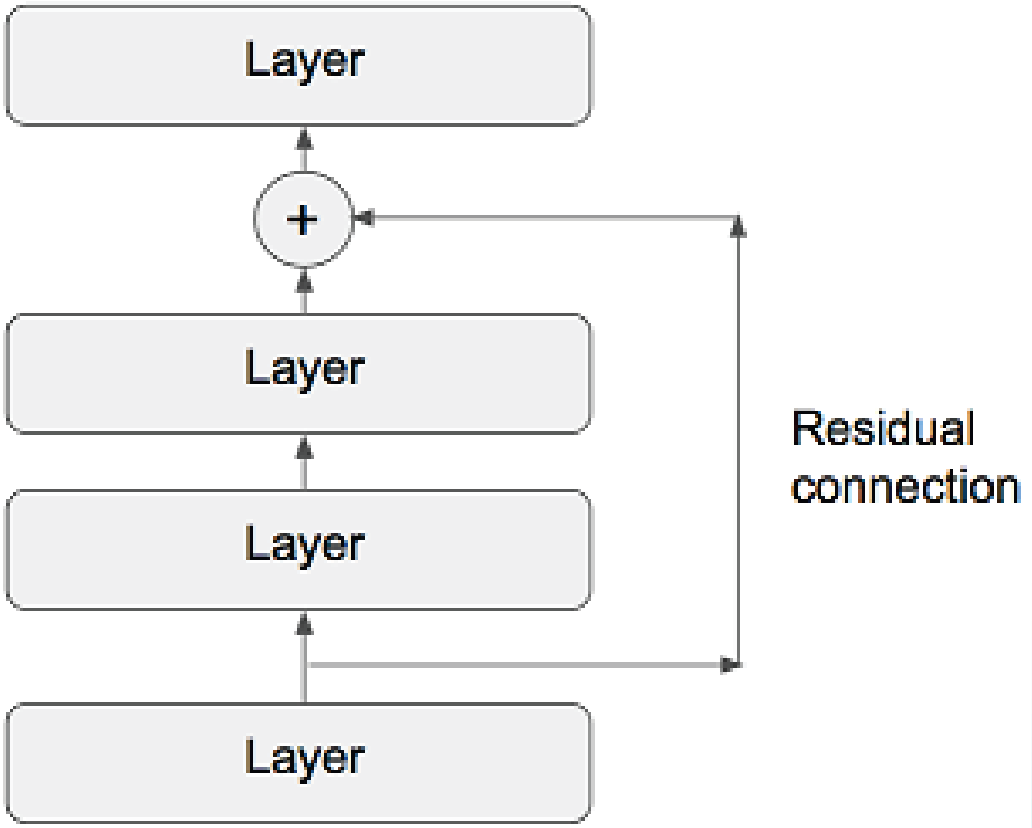
Les connexions résiduelles sont un composant de réseau de forme de graphe présent dans de nombreuses architectures, notamment Xception.

Elles traitent deux problèmes communs à tous les modèles d'apprentissage profond à grande échelle: les gradients disparaissant, et les restrictions de représentation.

Connexions résiduelles

Une connexion résiduelle consiste simplement à rendre la sortie d'une couche antérieure disponible comme entrée à une couche ultérieure, créant ainsi un raccourci dans un réseau séquentiel.

Au lieu d'être concaténée à l'activation ultérieure, la sortie antérieure est additionnée à l'activation ultérieure, ce qui suppose que les deux activations ont la même taille.



Connexions résiduelles

```
y = layers.Conv2D(128, 3, activation='relu')(x)
y = layers.Conv2D(128, 3, activation='relu')(y)
y = layers.Conv2D(128, 3, activation='relu')(y)

y = layers.add([y, x])
```

```
y = layers.Conv2D(128, 3, activation='relu')(x)
y = layers.Conv2D(128, 3, activation='relu')(y)
y = layers.MaxPooling2D(2, strides=2)(y)

residual = layers.Conv2D(1, strides=2)(x)

y = layers.add([y, residual])
```

Gradient disparaissant dans l'apprentissage profond

Si le signal de feedback doit être propagé à travers un ensemble profond de couches, le signal peut devenir très faible ou même être entièrement perdu, ce qui rend impossible d'entraîner le réseau.

Ce problème se produit à la fois avec des réseaux profonds et avec des réseaux récurrents sur de très longues séquences.

Obstacles représentationnels dans l'apprentissage profond

Dans un modèle séquentiel, chaque couche de représentation successive est construite sur la précédente, ce qui signifie qu'elle n'a accès qu'aux informations contenues dans l'activation de la couche précédente. Si une couche est trop petite, le modèle sera limité par la quantité d'informations qu'il est possible de faire passer dans les activations de cette couche.

Les connexions résiduelles, en réinjectant des informations antérieures dans le flux d'apprentissage, résolvent partiellement ce problème pour les modèles d'apprentissage profond.

Batch normalization

La normalisation est une grande catégorie de méthodes qui visent à rendre les différents éléments vus par un modèle d'apprentissage automatique plus semblables les uns aux autres, ce qui aide le modèle à apprendre et à généraliser correctement sur de nouvelles données.

Batch normalization est un type de couche dans Keras , capable de normaliser les données de manière adaptative, même si leur moyenne et leur variance changent au cours de l'apprentissage.

Batch normalization

Le principal effet de la normalisation par lots est qu'elle aide à la propagation du gradient, tout comme les connexions résiduelles, et permet donc la création des réseaux plus profonds.

La couche BatchNormalization est généralement utilisée après une couche convolutive ou dense.

Batch normalization

La couche BatchNormalization prend un argument 'axis', qui spécifie l'axe des caractéristiques qui doit être normalisé. Cet argument prend par défaut la valeur -1, le dernier axe dans le tenseur d'entrée.

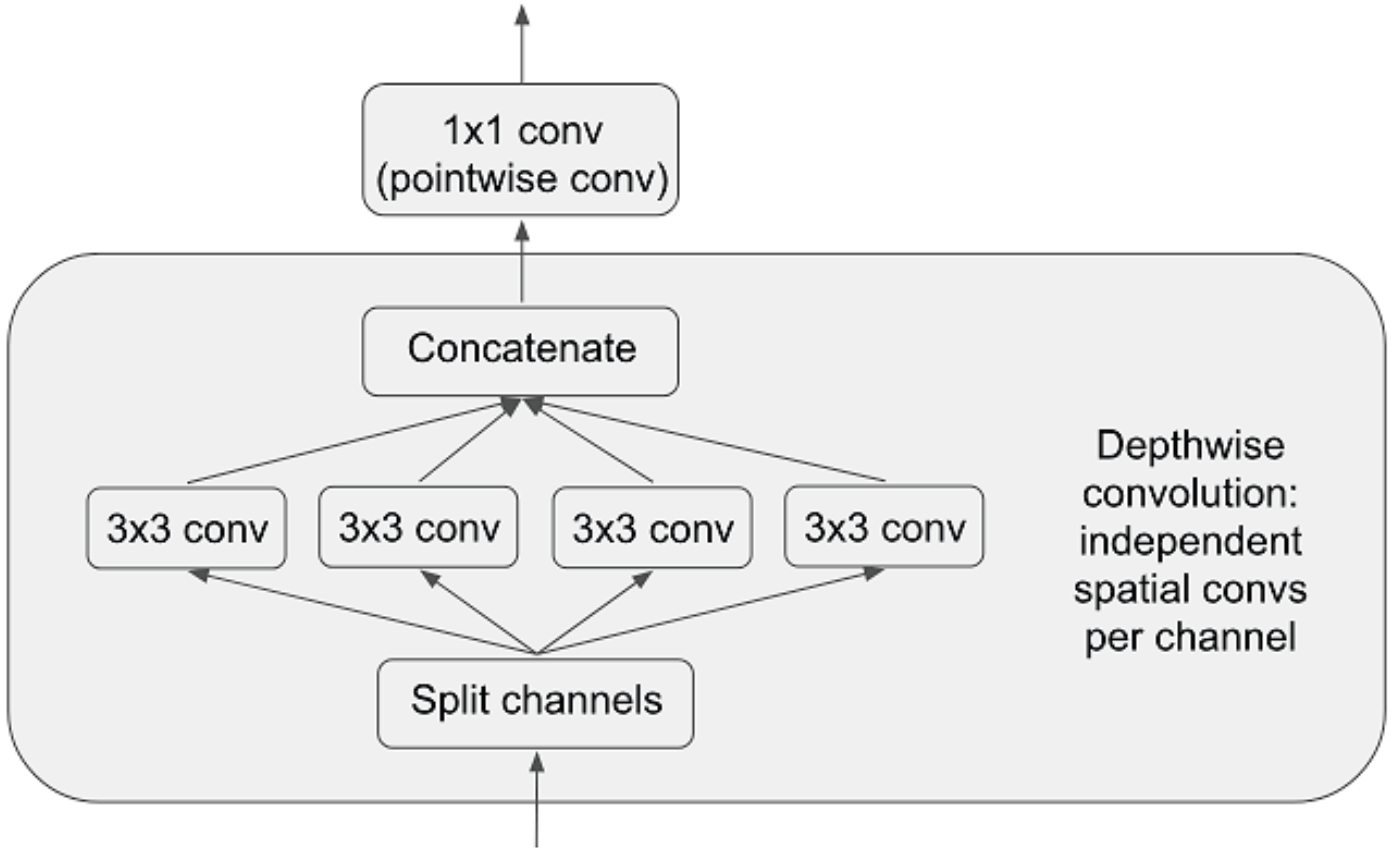
```
conv_model.add(layers.Conv2D(32, 3, activation='relu'))
conv_model.add(layers.BatchNormalization())

dense_model.add(layers.Dense(32, activation='relu'))
dense_model.add(layers.BatchNormalization())
```

Convolution séparable en profondeur

Une convolution séparable en profondeur effectue une convolution spatiale sur chaque canal de son entrée, indépendamment, avant de combiner les canaux de sortie via une convolution ponctuelle.

Elle nécessite beaucoup moins de paramètres et implique moins de calculs, ce qui permet d'obtenir des modèles plus petits et plus rapides.

Convolution séparable en profondeur



Convolution séparable en profondeur

```
model = Sequential()

model.add(SeparableConv2D(32,3, activation='relu', input_shape=(150, 150, 3)))
model.add(SeparableConv2D(64,3, activation='relu'))
model.add(MaxPooling2D(2))

model.add(SeparableConv2D(64,3, activation='relu'))
model.add(SeparableConv2D(128,3, activation='relu'))
model.add(MaxPooling2D(2))

model.add(SeparableConv2D(64,3, activation='relu'))
model.add(SeparableConv2D(128,3, activation='relu'))
model.add(GlobalAveragePooling2D())

model.add(Dense(32, activation='relu'))
model.add(Dense(7, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

Partage du poids des couches

Une autre caractéristique importante de l'API fonctionnelle est la possibilité de réutiliser une instance de couche plusieurs fois.

Cela nous permet de construire des modèles comportant des branches partagées, c'est-à-dire plusieurs branches qui partagent toutes les mêmes connaissances et effectuent les mêmes opérations.

Partage du poids des couches

On peut prendre l'exemple d'un modèle qui tente d'évaluer la similarité sémantique entre deux phrases. Le modèle aura deux entrées (les deux phrases à comparer) et produira un score compris entre 0 et 1.

```
lstm = layers.LSTM(32)

left_input = Input(shape=(None, 128))
left_output = lstm(left_input)

right_input = Input(shape=(None, 128))
right_output = lstm(right_input)

merged = layers.concatenate([left_output, right_output], axis=-1)
predictions = layers.Dense(1, activation='sigmoid')(merged)

model = Model([left_input, right_input], predictions)
model.fit([left_data, right_data], targets)
```

Utilisation des modèles en tant que couches

les modèles peuvent être utilisés comme on le fait pour les couches, on peut en effet considérer un modèle comme une " couche plus grande "

```
y = model(x)
y1, y2 = model([x1, x2])
```

Lorsque nous appelons une instance de modèle, nous réutilisons les poids du modèle, exactement comme ce qui se passe lorsque nous appelons une instance de couche

Utilisation des modèles en tant que couches

Un exemple simple de ce que nous pouvons construire en réutilisant des modèles serait un modèle de vision qui utilise une double caméra comme entrée.

```
from keras import applications

xception_base = applications.Xception(weights=None, include_top=False)

left_input = Input(shape=(250, 250, 3))
right_input = Input(shape=(250, 250, 3))

left_features = xception_base(left_input)
right_input = xception_base(right_input)

merged_features = layers.concatenate([left_features, right_input], axis=-1)
```

L’inspection et la surveillance des modèles d'apprentissage profond

Après avoir lancé un cycle d’apprentissage sur un grand ensemble de données, nous n'avons aucun contrôle sur la direction de l'apprentissage.
Si nous voulons éviter les mauvais résultats et le gaspillage de temps d’apprentissage, il est préférable d'utiliser une méthode qui renvoie des informations sur la progression de l’entraînement.

Utilisation de callbacks

Un callback est un objet qui est transmis au modèle dans l'appel à la fonction fit, et qui est appelé par le modèle à différents moments de l'apprentissage.

Il a accès à toutes les données disponibles sur l'état du modèle et ses performances, et il est capable d'agir, en interrompant l'entraînement, en sauvegardant un modèle, en chargeant un ensemble de poids différent ou en modifiant l'état du modèle.

Utilisation de callbacks

Les callbacks peuvent être utilisés pour :

- Pointage du modèle : sauvegarde des poids actuels du modèle à différents moments de l'apprentissage.
- Arrêt anticipé : interrompre l'apprentissage lorsque la perte de validation a cessé de s'améliorer.
- Ajustement dynamique des valeurs de certains paramètres pendant l'apprentissage, tels que le taux d'apprentissage.
- Enregistrement des mesures des entraînements et de la validation pendant l'entraînement, ou visualisation des représentations apprises par le modèle au fur et à mesure de leur mise à jour.

Les callbacks "modelcheckpoint" et "earlystopping"

On peut utiliser le callback EarlyStopping pour interrompre l'apprentissage lorsqu'une métrique cible surveillée a cessé de s'améliorer pendant un nombre fixe d'époques.

Ce callback est généralement utilisé en combinaison avec ModelCheckpoint, qui permet de sauvegarder régulièrement le modèle pendant l'apprentissage.

Les callbacks "modelcheckpoint" et "earlystopping"

```
callbacks_list = [keras.callbacks.EarlyStopping(monitor='acc',patience=1),
                  keras.callbacks.ModelCheckpoint(filepath='my_model.h5', monitor='val_loss',
                                                  save_best_only=True)]

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

model.fit(x, y,epochs=10,batch_size=32,
          callbacks=callbacks_list,validation_data=(x_val, y_val))
```

Le callback "ReduceLROnPlateau"

On peut utiliser ce callback pour réduire le taux d'apprentissage lorsque la perte de validation a cessé de s'améliorer.

La réduction du taux d'apprentissage en cas de "plateau de perte" est une stratégie efficace pour sortir des minima locaux pendant l'apprentissage.

```
callbacks_list = [keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
                                                    factor=0.1,patience=10)]
```

Callbacks personnalisés

Si on a besoin d'effectuer une action spécifique pendant l'apprentissage qui n'est pas couverte par l'un des callbacks intégrés, il faut écrire notre propre callback.

Les callbacks sont implémentés en sous-classant la classe keras.callbacks.Callback. Nous pouvons ensuite implémenter un nombre quelconque de ces méthodes, qui sont appelées à différents moments de l'apprentissage:

on_epoch_begin, on_epoch_end, on_batch_begin, on_batch_end, on_train_begin, on_train_end.

Callbacks personnalisés

Voici un exemple simple de callback personnalisé, où nous sauvegardons les activations de chaque couche du modèle à la fin de chaque époque, calculées sur le premier échantillon de l'ensemble de validation.

```
class ActivationLogger(keras.callbacks.Callback):
    def set_model(self, model):
        self.model = model
        layer_outputs = [layer.output for layer in model.layers]
        self.activations_model = keras.models.Model(model.input, layer_outputs)

    def on_epoch_end(self, epoch, logs=None):
        if self.validation_data is None:
            raise RuntimeError('Requires validation_data.')
            validation_sample = self.validation_data[0][0:1]
            activations = self.activations_model.predict(validation_sample)
            f = open('activations_at_epoch_' + str(epoch) + '.npz', 'w')
            np.savez(f, activations)
            f.close()
```

TensorBoard

TensorBoard est un outil de visualisation sur navigateur, qui est inclus dans la bibliothèque TensorFlow.

L'objectif principal de TensorBoard est de nous aider à surveiller visuellement tout ce qui se passe à l'intérieur de notre modèle pendant son apprentissage.

Il donne accès à plusieurs fonctionnalités intéressantes:

- Suivi visuel des métriques pendant l'apprentissage.
- Visualisation de l'architecture du modèle.
- Visualisation des histogrammes d'activations et de gradients.
- Exploration des embeddings en 3D.

Visualisation de la topologie du modèle

Keras permet également de représenter vos modèles de manière plus claire, sous forme de graphes de couches plutôt que de graphes de Tensorboard.

```
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='model1.png')
plot_model(model, show_shapes=True, to_file='model2.png')
```

Optimisation des hyperparamètres

Les paramètres au niveau de l'architecture sont appelés "hyperparamètres", pour les distinguer des paramètres d'un modèle.

Le processus d'optimisation des hyperparamètres se résume généralement à ceci :

- Choisissez un ensemble d'hyperparamètres (automatiquement).
- Construisez le modèle correspondant.
- Faire l'apprentissage sur vos données d'entraînement et mesurez la performance finale sur les données de validation.
- Choisissez le prochain ensemble d'hyperparamètres à essayer (automatiquement).
- Répétez l'opération.
- Finalement, mesurez la performance sur vos données de test.

Optimisation des hyperparamètres

La clé de ce processus est l'algorithme qui utilise l'historique des performances de validation pour différents ensembles d'hyperparamètres afin de choisir le prochain ensemble d'hyperparamètres à évaluer.

De nombreuses techniques sont possibles : l'optimisation bayésienne, les algorithmes génétiques, ou la recherche aléatoire simple.

Optimisation des hyperparamètres

L'apprentissage des poids d'un modèle est relativement facile, mais la mise à jour des hyperparamètres est extrêmement difficile.

- Le calcul du feedback peut être extrêmement coûteux, il nécessite la création et l'entraînement d'un nouveau modèle depuis le début sur votre ensemble de données.
- L'espace des hyperparamètres est typiquement constitué de choix discrets, et n'est donc pas différentiable. Par conséquent, on ne peut pas effectuer une descente de gradient. Au lieu de cela, on doit utiliser des techniques d'optimisation sans gradient, qui sont beaucoup moins efficaces.

Optimisation des hyperparamètres

Keras Tuner est un outil d'optimisation des hyperparamètres facile à utiliser qui résout les problèmes liés à la recherche d'hyperparamètres.

Il est équipé des algorithmes d'optimisation bayésienne, d'hyperbande et de recherche aléatoire.

```
import kerastuner as kt

tuner = kt.Hyperband( build_model, objective='val_accuracy',
                      max_epochs=30,hyperband_iterations=2)

tuner.search(train_data,validation_data=val_data, epochs=30)

best_model = tuner.get_best_models(1)[0]

best_hyperparameters = tuner.get_best_hyperparameters(1)[0]
```

https://github.com/keras-team/keras-tuner

Assemblage des modèles

L'assemblage consiste à mettre en commun les prédictions d'un ensemble de modèles différents, afin de produire de meilleures prédictions.

L'assemblage repose sur l'hypothèse que différents bons modèles entraînés indépendamment sont susceptibles d'être efficients puisque chaque modèle examine des aspects légèrement différents des données pour faire ses prédictions.

Assemblage des modèles

- Une façon d'assembler des modèles est de faire une moyenne pondérée.
- En général, les meilleurs modèles auront un poids plus élevé, et les moins bons auront un poids plus faible.
- Vous devez assembler des modèles qui sont aussi bons que possible tout en étant différents au maximum. Cela signifie généralement l'utilisation d'architectures très différentes ou même de méthodes d'apprentissage automatique de différentes catégories.

Conclusion

- Pour construire des convnets profonds très performants, vous devrez exploiter les connexions résiduelles, la normalisation des lots et les convolutions séparées en profondeur.

- La construction de réseaux profonds exige de faire de nombreux petits choix d'hyperparamètres et d'architecture, qui définissent ensemble la qualité du modèle. Au lieu de baser ces choix sur l'intuition ou le hasard, il est préférable de rechercher systématiquement l'espace des hyperparamètres pour trouver les choix optimaux.

- L'obtention des meilleurs résultats possible sur une tâche peut, dans certains cas, être effectuée avec de grands ensembles de modèles. L'assemblage par une moyenne pondérée est généralement suffisant.