CSC 8607 – Introduction au deep learning

Projets

GhostNet: More Features from Cheap Operations

-

Lien vers le Git du projet : MarinetHENEIN/DL-project

21/03/2025 Marinet HENEIN

Voici les réponses aux questions du projet de deep learning:

1. **Présentez l'objectif de l'article. Quel problème cherche-t-il à résoudre ?**

L'article **"GhostNet: More Features from Cheap Operations"** vise à résoudre le problème de la complexité et du coût d'exécution élevé des réseaux de neurones convolutifs (CNN) tout en maintenant leur performance.

Il propose une architecture appelée GhostNet, qui utilise des modules Ghost pour générer davantage de cartes de caractéristiques avec moins de paramètres, réduisant ainsi les coûts.

2. **Comment avez-vous chargé le dataset ? Quelles transformations avez-vous effectuées ? Pourquoi ?**

Le dataset CIFAR-10 a été chargé en utilisant `: from tensorflow.keras.datasets import cifar10`

Les images ont été normalisées en divisant leurs valeurs par 255.0 pour obtenir des valeurs entre 0 et 1. Cette normalisation est effectuée pour réduire l'amplitude des valeurs d'entrée, ce qui facilite l'apprentissage du modèle.

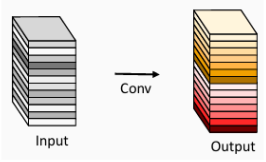
Le dataset CIFAR-10 a été choisi au lieu d'ImageNet pour plusieurs raisons :

- **Taille du dataset** : CIFAR-10 est beaucoup plus petit (60,000 images) que ImageNet (1,28 million d'images), ce qui réduit le temps d'entraînement et les ressources nécessaires.
- **Complexité du modèle :** Les modèles entraînés sur CIFAR-10 peuvent être moins complexes, ce qui facilite l'optimisation et la recherche d'hyperparamètres.
- **Rapidité des tests :** CIFAR-10 permet de tester rapidement différentes architectures et hyperparamètres sans attendre des heures pour chaque entraînement.

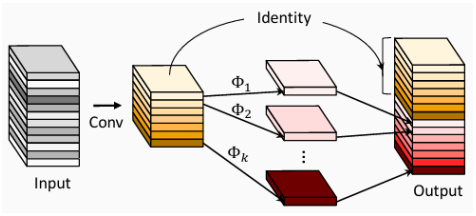
3. **Décrivez l'architecture proposée dans l'article.**

GhostNet est principalement composé d'une pile de Ghost bottlenecks, avec une couche de convolution standard initiale suivie de plusieurs Ghost bottlenecks à canaux progressivement augmentés. Les Ghost bottlenecks sont regroupés en différentes étapes selon la taille des cartes de caractéristiques d'entrée. À la fin, une moyenne globale et une couche de convolution sont utilisées pour transformer les

cartes de caractéristiques en un vecteur de caractéristiques à 1280 dimensions pour la classification finale. Le module Squeeze-and-Excitation (SE) est également appliqué dans certains Ghost bottlenecks.



(a) The convolutional layer.



(b) The Ghost module.

Figure 2. An illustration of the convolutional layer and the proposed Ghost module for outputting the same number of feature maps. Φ represents the cheap operation.

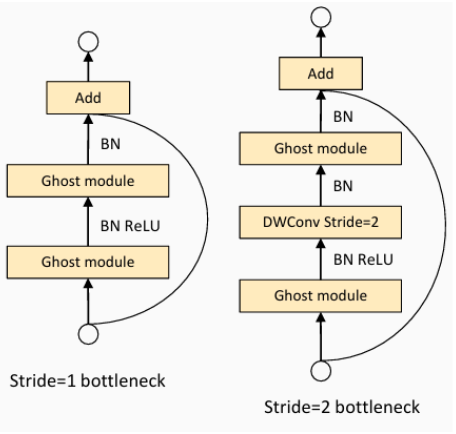


Figure 3. Ghost bottleneck. Left: Ghost bottleneck with stride=1; right: Ghost bottleneck with stride=2.

4. **Pour chaque couche du modèle, précisez :**
    - **Type de couche (Dense, Conv2D, LSTM, etc.)**
    - **Taille des entrées et sorties**
    - **Nombre de paramètres**

- Première couche Conv2D : Type Conv2D, Entrées (32, 32, 3), Sorties (32, 32, 32), Paramètres : $32\times3\times3\times3+32=896$

- GhostModule : Chaque GhostModule inclut une convolution standard suivie d'une convolution depthwise. Par exemple, pour le premier GhostModule, Entrées (32, 32, 32), Sorties (32, 32, 64), Paramètres :

  $32\times3\times3\times16+16\times1\times1\times32+2\times64=1\ 536+512+128=2\ 176$

- Depthwise Separable Convolution : Type DepthwiseConv2D, Entrées (32, 32, 512), Sorties (32, 32, 512), Paramètres : $512\times3\times3+512=4\ 608+512=5\ 120$

- Global Average Pooling : Type GlobalAveragePooling2D, Entrées (32, 32, 512), Sorties (512).
- Dense : Type Dense, Entrées (512), Sorties (512), Paramètres :

  $512 \times 512 + 512 = 262\,144 + 512 = 262\,656$

- Dense finale : Type Dense, Entrées (512), Sorties (10), Paramètres :

  $512 \times 10 + 10 = 5\,120 + 10 = 5\,130$

5. **Quel est le nombre total de paramètres du modèle ?**

Le nombre total de paramètres dépend des couches et de leur configuration. Pour une estimation approximative, on peut additionner les paramètres des couches et avoir un ordre de grandeur autour de 300 000 paramètres.

6. **Quelle fonction de perte a été utilisée ? Pourquoi ?**

La fonction de perte utilisée est la cross-entropie catégorielle (`sparse_categorical_crossentropy`). Elle est choisie pour des problèmes de classification multi-classe, ce qui convient à notre dataset CIFAR-10.

7. **Quel optimiseur a été choisi ? Justifiez ce choix.**

L'optimiseur Adam a été choisi pour son efficacité dans la convergence rapide et stable. Adam adapte les taux d'apprentissage individuellement pour chaque paramètre, ce qui est utile pour éviter les problèmes de vanishing ou exploding gradients.

8. **Expliquez la stratégie de train/validation/test mise en place.**

Le dataset CIFAR-10 est divisé en deux parties : 50 000 images pour l'entraînement et 10 000 images pour la validation et le test. Dans ce projet, les données d'entraînement sont utilisées pour ajuster les paramètres du modèle, et les données de validation sont utilisées pour évaluer les performances pendant l'entraînement.

9. **Détaillez les expériences (taille du dataset, paramètres fixes, temps d’entraînement, nombre d'époques, batch size, ...).**

Taille du dataset : 50 000 images pour l'entraînement et 10 000 images pour la validation et le test.

Paramètres fixes : Learning rate initial à 0.001, batch size à 32.

Temps d'entraînement : Moins d'1 minute par epoch sur un GPU (collab)

Nombre d'époques : 20.

Batch size : 32.

```
# Entraîner le modèle GhostNet sur GPU avec TensorBoard
best_lr = 0.001
best_bs = 32
ghostnet_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=best_lr),
                       loss='sparse_categorical_crossentropy',
                       metrics=['accuracy'])

ghostnet_history = ghostnet_model.fit(
    x_train, y_train,
    epochs=20,
    validation_data=(x_test, y_test),
    batch_size=best_bs,
    callbacks=[tensorboard_callback]  # Ajout de TensorBoard
)
```

```
Epoch 1/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 27s 12ms/step - accuracy: 0.8291 - loss: 0.4535 - val_accuracy: 0.6397 - val_loss: 1.3969
Epoch 2/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.8446 - loss: 0.4157 - val_accuracy: 0.6388 - val_loss: 1.4208
Epoch 3/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.8535 - loss: 0.3971 - val_accuracy: 0.6378 - val_loss: 1.4296
Epoch 4/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.8547 - loss: 0.3922 - val_accuracy: 0.6367 - val_loss: 1.4448
Epoch 5/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.8542 - loss: 0.3914 - val_accuracy: 0.6388 - val_loss: 1.4572
```

10. **Implémentez un mécanisme d'early stopping et une planification du learning rate (scheduling), même si ce n'était pas présent dans l'article. Quels sont les effets observés ?**

```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3)
```

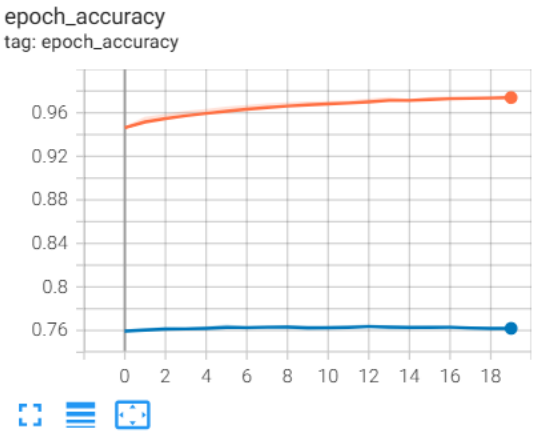
Ces callbacks permettent d'arrêter l'entraînement lorsque la perte de validation cesse d'améliorer pendant un certain nombre d'époques et de réduire le learning rate pour éviter l' overfitting.

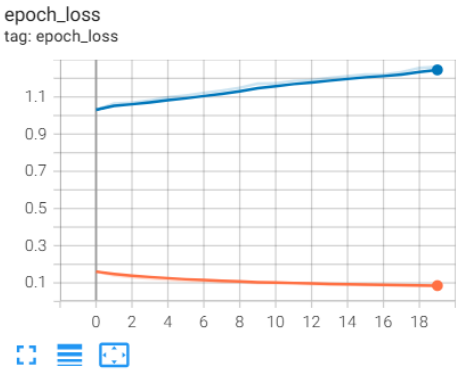
11. **Pour guider votre recherche d'hyperparamètres, suivez les étapes suivantes vues en cours:**
    1. **Vérifiez la perte initiale du modèle. Est-elle cohérente ?**
    2. **Overfit intentionnellement sur un petit échantillon. Le modèle parvient-il à apprendre ?**
    3. **Trouvez un learning rate qui fait diminuer la loss de manière stable.**
    4. **Effectuez une petite recherche en grille autour des paramètres trouvés.**
    5. **Pour les configurations prometteuses, poursuivez l'entraînement et observez les courbes d'apprentissage.**
    6. **Que remarquez-vous sur les courbes d'apprentissage (TensorBoard) ?**

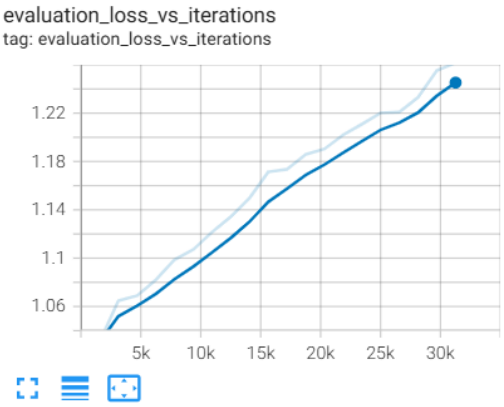
```
# Hyperparamètres à tester
learning_rates = [0.001, 0.01, 0.1]
batch_sizes = [16, 32, 64]

# Matrice pour stocker les résultats
results_grid = np.zeros((len(learning_rates), len(batch_sizes)))

# Boucle pour tester chaque combinaison d'hyperparamètres
for i, lr in enumerate(learning_rates):
    for j, bs in enumerate(batch_sizes):
        # Compiler le modèle avec les nouveaux hyperparamètres
        ghostnet_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
                               loss='sparse_categorical_crossentropy',
                               metrics=['accuracy'])

        # Entraîner le modèle
        history = ghostnet_model.fit(
            x_train, y_train,
            epochs=10,
            batch_size=bs,
            validation_data=(x_test, y_test),
            verbose=0
        )

        # Stocker la meilleure précision de validation
        results_grid[i, j] = max(history.history['val_accuracy'])

# Afficher les résultats
print(results_grid)
```

Résultat de la matrice :

```
Num GPUs Available:  1
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 32s 0us/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.0966 - loss: 2.3026
Perte initiale : 2.302651882171631, Précision initiale : 0.0934000015258789
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 360ms/step - accuracy: 0.1416 - loss: 3.9055
Perte après overfitting : 4.034611225128174, Précision : 0.12999999523162842
[[0.80400002 0.85070002 0.83810002]
 [0.77090001 0.83759999 0.84429997]
 [0.71929997 0.736      0.7421    ]]
```

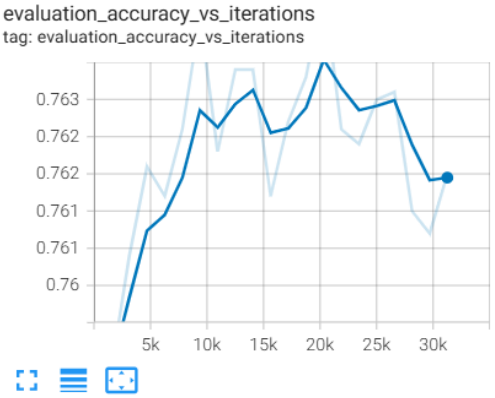






12. Quels hyperparamètres semblent optimaux ? Justifiez.

Les hyperparamètres optimaux sont un learning rate de 0.01 et un batch size de 32, car ils offrent la meilleure précision sur les données de validation . Cette combinaison montre une convergence stable sans trop de fluctuations dans la précision.

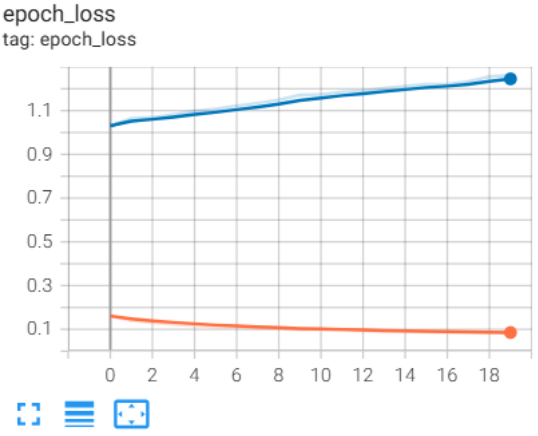


13. Avez-vous réussi à reproduire les résultats de l'article ? Si non, quelles pourraient être les raisons ?

Les résultats ne sont pas directement comparables à ceux de l'article original, car les expériences ont été menées avec des configurations différentes et un dataset différent. Mais CIFAR-10 permet quand même d'avoir une approche quasi similaire au fonctionnement de GhostNet à plus grande échelle.

14. Avez-vous observé des problèmes d'exploding ou vanishing gradient ? Si oui, comment les avez-vous identifiés et résolus ?

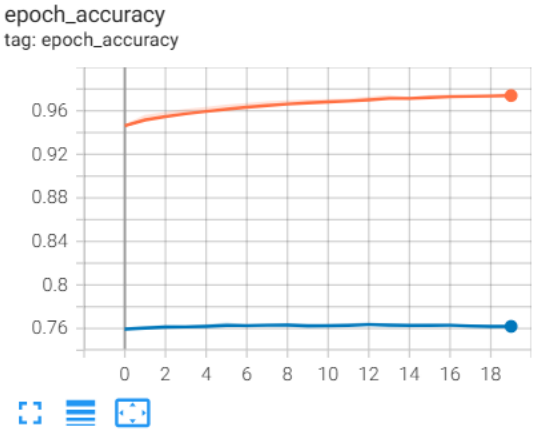
Pas de problème d'exploding ou vanishing gradient observé. Sur `epoch_loss`, la perte diminue régulièrement pour les données d'entraînement, ce qui indique que le modèle converge correctement grâce à l'optimiseur Adam.

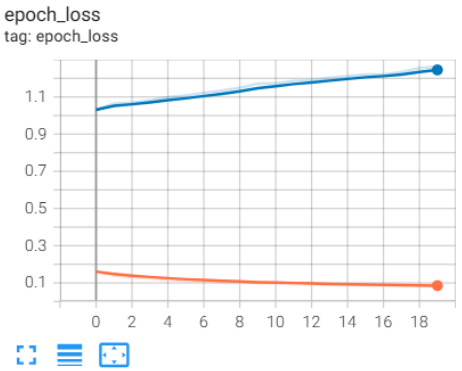


15. En utilisant des graphiques, montrez si votre modèle a surappris (overfitting) ou sous-appris (underfitting).

Le modèle montre un léger overfitting :

- Sur `epoch_accuracy`, on observe une divergence entre la précision d'entraînement et celle de validation vers la fin des epoch.
- Sur `epoch_loss`, la perte d'entraînement diminue fortement, tandis que celle de validation reste stable ou augmente légèrement.





16. Avez-vous utilisé des techniques de régularisation (dropout, weight decay) ? Étaient-elles efficaces ?

Non, cependant leur inclusion pourrait réduire l'overfitting observé

17. Qu'avez-vous appris de ce projet ?

Ce projet m'a permis de comprendre l'implémentation et l'entraînement d'un modèle GhostNet simplifié. Il m'a permis de manipuler les points suivant:

1. la recherche d'hyperparamètres pour optimiser les performances.
2. l'utilisation des courbes TensorBoard pour surveiller le comportement du modèle.
3. l'implémentation de techniques comme le scheduling du learning rate et l'early stopping pour éviter l'overfitting

18. Que feriez-vous différemment si vous deviez recommencer ?

Si je devais le refaire :

- Ajouter des techniques de régularisation comme le dropout ou le weight decay
- Étendre la recherche d'hyperparamètres avec plus de combinaisons

- Augmenter le nombre d'epoch (ça aurait pris vraiment beaucoup de temps)