TP de Tensor Processing Unit

Auteur: Jiangbo WANG Date: 6/1/2023

1 Tensors

Question 1 :définir une constante

```
a = tf.constant(8)
b = tf.constant(5)
print(a)
print(b)
```

Les résultats obtenus sont les suivants : il est apparent que nous obtenons un tensor. Comme il est défini comme une constante, la forme du tensor est (), et nous pouvons également obtenir le type de données du tensor, qui est int32.

```
tf.Tensor(8, shape=(), dtype=int32)
tf.Tensor(5, shape=(), dtype=int32)
```

Question 2 : définir vecteurs

Définir deux vecteurs X et Y:

```
X = tf.constant([1.0, 2.0, 3.0])
Y = tf.constant([4.0, 5.0, 6.0])
print(X)
print(Y)
```

```
tf.Tensor([1. 2. 3.], shape=(3,), dtype=float32)
tf.Tensor([4. 5. 6.], shape=(3,), dtype=float32)
```

Définir deux matrices A et B:

```
A = tf.constant([[1.0, 2.0],[4.0, 5.0]])
B = tf.constant([[3.0, 4.0],[6.0, 7.0]])
print(A)
print(B)
```

```
tf.Tensor(
[[1. 2.]
 [4. 5.]], shape=(2, 2), dtype=float32)
tf.Tensor(
[[3. 4.]
 [6. 7.]], shape=(2, 2), dtype=float32)
```

Définir un tenseur tridimensionnel de forme 3x2x5.

```
three_dim = tf.ones([3, 2, 5])
```

```
print(three_dim_tensor)
print("dimension {}, ndim {}".format(three_dim.shape,three_dim.ndim))
```

Les résultats obtenus sont les suivants, on peut voir que la dimension du tenseur obtenu est la même que celle que nous avons définie.

```
tf.Tensor(
[[[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]], shape=(3, 2, 5), dtype=float32)
dimension (3, 2, 5), ndim 3
```

Définir un tenseur quadridimensionnel de forme 4x3x6x3.

```
four_dimensional_tensor = tf.zeros([4, 3, 6, 3])
```

Question 3:Basic mathematical operations

```
print("X: ",X)
print("Y: ",Y)
addXY = tf.add(X,Y)
subtractXY = tf.subtract(X,Y)
multiplyXY = tf.multiply(X,Y)
divideXY = tf.divide(X,Y)
print("X+Y = ",addXY)
print("X-Y = ",subtractXY)
print("X*Y = ",multiplyXY)
print("X/Y = ",divideXY)
```

Les résultats obtenus sont les suivants, il est clair que l'utilisation des fonctions add, subtract, multiply, divide correspond à des opérations sur les éléments respectifs du tenseur, et non à des opérations de matrice.

```
X:  tf.Tensor([1. 2. 3.], shape=(3,), dtype=float32)
Y:  tf.Tensor([4. 5. 6.], shape=(3,), dtype=float32)
X+Y =  tf.Tensor([5. 7. 9.], shape=(3,), dtype=float32)
X-Y =  tf.Tensor([-3. -3. -3.], shape=(3,), dtype=float32)
X*Y =  tf.Tensor([ 4. 10. 18.], shape=(3,), dtype=float32)
X/Y =  tf.Tensor([0.25 0.4  0.5 ], shape=(3,), dtype=float32)
```

Question 4 : Reshapping

```
print("My three_dim shape: ",three_dim)
three_dim_reshape10_3 = tf.reshape(three_dim,[10,3])
print("My three_dimensional_tensor_reshape10_3: ",three_dim_reshape10_3)
```

```
three_dim_reshape2_15 = tf.reshape(three_dim,[2,15])
print("My three_dimensional_tensor_reshape2_15: ",three_dim_reshape2_15)
three_dim_reshape6_1 = tf.reshape(three_dim,[6,-1])
print("My three_dimensional_tensor_reshape6__1: ",three_dim_reshape6_1)
```

D'après les résultats ci-dessous, nous avons réussi à changer les dimensions de notre tenseur défini à l'aide de la fonction reshape. Cependant, le nombre d'éléments dans le tenseur doit rester constant, 3*2*5=10*3=6*5 ; en utilisant -1, TensorFlow calcule automatiquement la taille des dimensions restantes.

```
My three_dim shape:  tf.Tensor(
[[[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]], shape=(3, 2, 5), dtype=float32)
My three_dim_reshape10_3:  tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]], shape=(10, 3), dtype=float32)
My three_dim_reshape2_15:  tf.Tensor(
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]], shape=(2, 15), dtype=float32)
My three_dim_reshape6_1:  tf.Tensor(
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]], shape=(6, 5), dtype=float32)
```

Question 5 : Variables

```
bool_variable = tf.Variable([False, False, False, True])
complex_variable = tf.Variable([5 + 4j, 6 + 1j])
print("bool_variable",bool_variable)
print("complex_variable",complex_variable)
```

D'après les résultats ci-dessous, nous avons réussi à définir une variable.

```
bool_variable <tf.Variable 'Variable:0' shape=(4,) dtype=bool, numpy=array([False,
False, False,  True])>
complex_variable <tf.Variable 'Variable:0' shape=(2,) dtype=complex128,
numpy=array([5.+4.j, 6.+1.j])>
```

Question 6 :Your own function

```
@tf.function
def my_function(a,b):
  c = tf.add(a,b)
  return c

print("add two constants: ",my_function(a,b))
print("add two constants: ",my_function(X,Y))
```

D'après les résultats ci-dessous, nous avons réussi à définir une variable

```
add two constants:  tf.Tensor(13, shape=(), dtype=int32)
add two constants:  tf.Tensor([5. 7. 9.], shape=(3,), dtype=float32)
```

2 TPU Performance

Question 1 : available resource

```
# Detect hardware
try:
  tpu_resolver = tf.distribute.cluster_resolver.TPUClusterResolver() # TPU
detection
except ValueError:
  tpu_resolver = None
  gpus = tf.config.experimental.list_logical_devices("GPU")

# Select appropriate distribution strategy
if tpu_resolver:
  tf.config.experimental_connect_to_cluster(tpu_resolver)
  tf.tpu.experimental.initialize_tpu_system(tpu_resolver)
  strategy = tf.distribute.experimental.TPUStrategy(tpu_resolver)
  print('Running on TPU ', tpu_resolver.cluster_spec().as_dict()['worker'])
elif len(gpus) > 1:
  strategy = tf.distribute.MirroredStrategy([gpu.name for gpu in gpus])
  print('Running on multiple GPUs ', [gpu.name for gpu in gpus])
elif len(gpus) == 1:
  strategy = tf.distribute.get_strategy() # default strategy that works on CPU
and single GPU
  print('Running on single GPU ', gpus[0].name)
else:
```

```
  strategy = tf.distribute.get_strategy() # default strategy that works on CPU and single GPU
  print('Running on CPU')
print("Number of accelerators: ", strategy.num_replicas_in_sync)
```

Le code ci-dessus est utilisé pour détecter le matériel disponible. Les résultats obtenus sont les suivants : lorsque le périphérique disponible est un CPU ou un GPU, le nombre d'accélérateurs est de 1. Lorsque le périphérique disponible est un TPU, le nombre d'accélérateurs est de 8.

```
Running on CPU
Number of accelerators:  1
```

```
Running on single GPU  /device:GPU:0
Number of accelerators:  1
```

```
Running on TPU  ['10.81.220.178:8470']
Number of accelerators:  8
```

Question 2 : parameters

```
BATCH_SIZE = 64 * strategy.num_replicas_in_sync
LEARNING_RATE = 0.01
LEARNING_RATE_EXP_DECAY = 0.6 if strategy.num_replicas_in_sync == 1 else 0.7
EPOCHS = 10
steps_per_epoch = 60000//BATCH_SIZE
```

BATCH_SIZE : C'est la taille du lot global (global batch size), c'est-à-dire le nombre d'échantillons de données traités simultanément à chaque itération d'entraînement. Cette taille est ajustée en fonction du nombre de réplicas synchronisés dans la stratégie (strategy.num_replicas_in_sync). Ici, nous utilisons 64 multipliés par le nombre de réplicas. Dans l'entraînement distribué, la taille du lot est généralement augmentée en fonction du nombre de cœurs de processeur disponibles, afin d'améliorer l'efficacité et l'utilisation des ressources. Sur Google Colab, un TPU a 8 cœurs, donc avec un seul TPU, num_replicas_in_sync est 8, ce qui donne BATCH_SIZE = 512. Les GPU et les CPU utilisent un seul cœur, donc BATCH_SIZE = 64.

LEARNING RATE : Le taux d'apprentissage est un paramètre utilisé pour mettre à jour les poids du réseau pendant l'entraînement. Il détermine l'ampleur des ajustements de poids. S'il est trop élevé, cela peut rendre l'entraînement instable ; s'il est trop bas, l'entraînement peut être trop lent.

LEARNING RATE EXP DECAY : Il s'agit du facteur de décroissance exponentielle du taux d'apprentissage. Avec la progression de l'entraînement, le taux d'apprentissage est généralement réduit progressivement, ce qui aide le modèle à ajuster les poids de manière plus fine dans les dernières phases de l'entraînement. Si num_replicas_in_sync est égal à 1, c'est-à-dire sans entraînement distribué, un facteur de décroissance de 0,6 est utilisé ; s'il y a plusieurs réplicas, un facteur de 0,7 est utilisé. Une décroissance de 0,7 est plus lente que 0,6, ce qui signifie que le taux d'apprentissage diminue moins rapidement.

EPOCHS : Cela représente le nombre de fois que le jeu de données est parcouru en entier. Un epoch signifie que chaque point de données est utilisé une fois dans le processus d'entraînement. Plus il y a d'epochs, plus le modèle a de chances d'apprendre les données.
steps_per_epoch : Ce paramètre définit le nombre d'étapes d'entraînement dans chaque epoch. Il est calculé en divisant la taille totale du jeu de données par la taille du lot global. Comme le jeu de données contient 60 000 points de données, pour les CPU et GPU, steps_per_epoch = 60000/64. Pour les TPU, steps_per_epoch = 60000/512 = 117.

Question 3 : model structure

Les résultats de notre modèle sont présentés ci-dessous. Il contient au total 354 538 paramètres, dont 354 002 sont des paramètres entraînables et 536 sont des paramètres non entraînables.

Layer (type)	Output Shape	Param #
image (Reshape)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 28, 28, 12)	108
batch_normalization (BatchNormalization)	(None, 28, 28, 12)	36
activation (Activation)	(None, 28, 28, 12)	0
conv2d_1 (Conv2D)	(None, 14, 14, 24)	10368
batch_normalization_1 (BatchNormalization)	(None, 14, 14, 24)	72
activation_1 (Activation)	(None, 14, 14, 24)	0
conv2d_2 (Conv2D)	(None, 7, 7, 32)	27648
batch_normalization_2 (BatchNormalization)	(None, 7, 7, 32)	96
activation_2 (Activation)	(None, 7, 7, 32)	0
flatten (Flatten)	(None, 1568)	0
dense (Dense)	(None, 200)	313600
batch_normalization_3 (BatchNormalization)	(None, 200)	600
activation_3 (Activation)	(None, 200)	0
dropout (Dropout)	(None, 200)	0
dense_1 (Dense)	(None, 10)	2010

Total params: 354538 (1.35 MB)
Trainable params: 354002 (1.35 MB)
Non-trainable params: 536 (2.09 KB)

Question 4 : model training and evaluation

Le code relatif à l'entraînement et à l'évaluation du modèle est présenté ci-dessous.

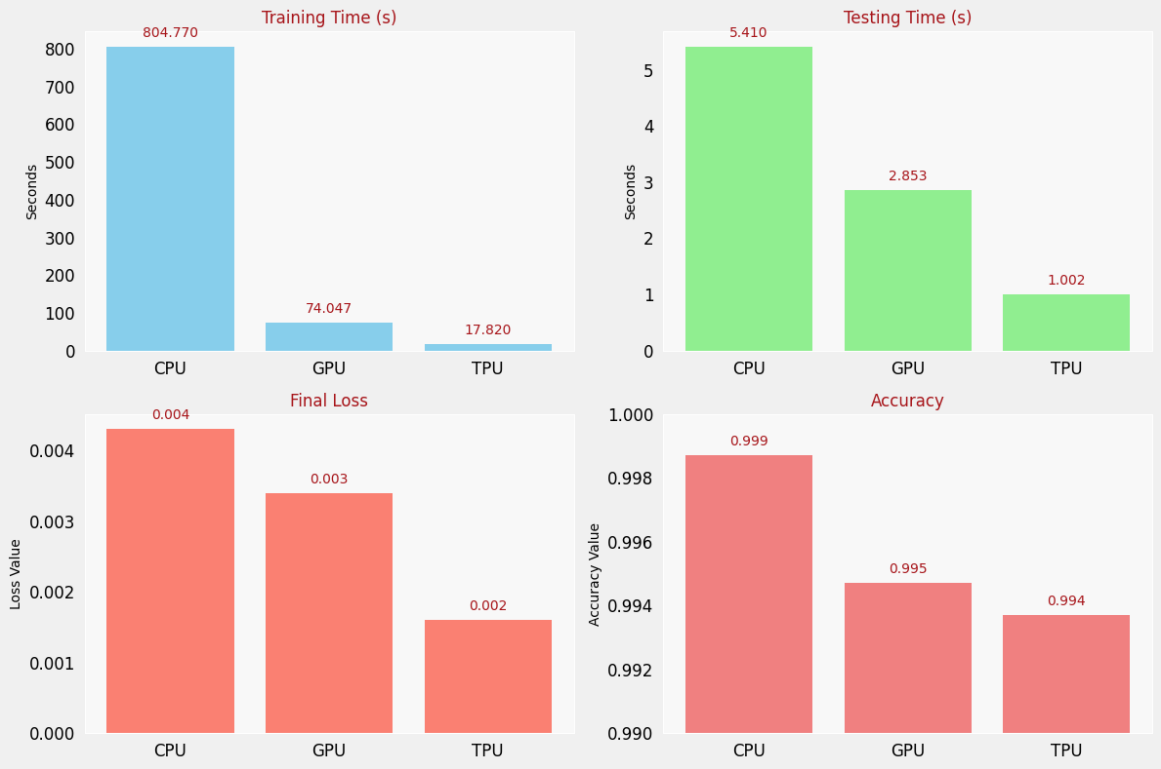
```
import time
EPOCHS = 10
steps_per_epoch = 60000//BATCH_SIZE  # 60,000 items in this dataset
print("Steps per epoch: ", steps_per_epoch)

start_time = time.time()
history = model.fit(training_dataset,
                    steps_per_epoch=steps_per_epoch, epochs=EPOCHS,
                    callbacks=[lr_decay])
end_time = time.time()
print("Training time:{} s".format(end_time-start_time))
start_time = time.time()
final_stats = model.evaluate(validation_dataset, steps=1)
end_time = time.time()
print("Testing time:{} s".format(end_time-start_time))
print("Validation accuracy: ", final_stats[1])
```

Question 5 : Results comparison

En exécutant le code d'entraînement du modèle présenté dans la question 4, les résultats obtenus sont les suivants :

	CPU	GPU	TPU
Training time	804.77 s	74.047 s	17.82 s
Testing time	5.41 s	2.853 s	1.002 s
Final loss	0.0043	0.0034	0.0016
Accuracy	0.9987	0.9947	0.9937
Step each echo	937	937	117



On peut voir que que ce soit en termes de temps d'entraînement, de valeur de perte ou de précision des tests, le TPU est supérieur au GPU, qui est lui-même supérieur au CPU(TPU>GPU>CPU). Cependant, bien que la différence en termes de précision ne soit pas très grande, il y a une grande différence dans le temps d'entraînement.

Question 6 : change the BATCH_SIZE

BATCH_SIZE	Training time(s)	Testing times()	Testing loss	Testing Accuracy
32 * 8	39.44	2.25	0.0165	0.9941
64 * 8	21.17	2.29	0.0188	0.9936
128 * 8	13.424	2.22	0.0290	0.9909
256 * 8	12.27	2.23	1.0061	0.6695
512* 8	11.74	2.27	1.4375	0.5500

On peut observer qu'avec l'augmentation de BATCH_SIZE, le temps d'entraînement s'allonge, la perte lors des tests (testing loss) augmente et la précision des tests (testing accuracy) diminue. Le tableau montre que lorsque BATCH_SIZE est inférieur à 128*8, la précision des tests peut se maintenir au-dessus de 0.99. Cependant, lorsque BATCH_SIZE augmente à 256*8, la précision des tests chute soudainement à 0.67. Si BATCH_SIZE continue

d'augmenter, la précision des tests diminue encore considérablement. En même temps, on peut voir que lorsque BATCH_SIZE est de 32*8, le temps d'entraînement est de 39.44 secondes, soit le double du temps d'entraînement pour un BATCH_SIZE de 64*8, mais la précision des tests des deux est similaire. Ainsi, la conclusion est que BATCH_SIZE ne doit pas être réglé trop bas ni trop haut.