インテル® Arc™ GPU 上の TensorFlow* による転移学習

インテルのコンシューマー向け GPU と Windows* Subsystem for Linux* 2 を使用した高速かつ簡単なトレーニングと推論

Christopher Lishka インテル コーポレーション AI フレームワーク・エンジニア
AG Ramesh インテル コーポレーション 主席エンジニア
Geetanjali Krishna インテル コーポレーション AI ソフトウェア・エンジニアリング・マネージャー

転移学習は、新しいデータセットで事前トレーニング済みモデルの使用を可能にするディープラーニング（DL）手法です。モデルの重みで学習した特徴が適切に適用されるように、新しいデータはオリジナルのデータに十分に類似している必要があります。この場合、モデルを完全にトレーニングする場合と比較して、新しいデータセットを使用してトレーニングする時間を大幅に短縮できます。これは、レイヤーの大部分で事前トレーニング済みの静的重みを残し、最後のモデルレイヤーを新しいデータセットでトレーニングされた新しい Dense レイヤーに置き換える、「ヘッドレスモデル」を使用しているためです。

インテル® Arc™ A シリーズ・ディスクリート GPU は、PC 上で DL ワークロードを迅速に実行する簡単な方法を提供し、TensorFlow* モデルと PyTorch* モデルの両方で動作します。この記事では、インテル® Arc™ GPU 上で TensorFlow 向けインテル® エクステンション（ITEX）（英語）を実行し、Windows* 上で事前に構築された ITEX Docker* イメージを使用してセットアップを簡素化します。この例では、ImageNet データセットで事前トレーニング済みの、TensorFlow* Hub の EfficientNetB0 モデルを使用します。新しいデータセットは、TensorFlow* Datasets にある「Stanford Dogs」です。これには ImageNet で提供されているものより正確な犬種のラベルが付けられた犬の画像が含まれています。Stanford Dogs 向けに DL ネットワークを調整するため、インテル® Arc™ A770 GPU を使用して迅速にトレーニングされる新しい Dense レイヤーを追加します。次に、完全にトレーニングした EfficientNetB0 と比較して、転移学習がどのくらい高速化されたかを確認します。

セットアップ

以前の記事、「インテル® Arc™ GPU で TensorFlow* Stable Diffusion を実行する」では、Windows* Subsystem for Linux* 2（WSL2）で実行する Ubuntu* コンテナをセットアップして Windows* ホスト上のインテル® Arc™ GPU にアクセスする方法を説明しました。この例では、WSL2 と Docker*、および事前に構築された ITEX Docker* イメージを使用して、ITEX プラグインのインストールを簡素化する方法を紹介します。この記事では、16GB のインテル® Arc™ A770 ディスクリート GPU カードを装着した第 13 世代インテル® Core™ i9 プロセッサー・ベースの PC を使用しました。Windows* 11 上の Docker* コンテナで実行する ITEX を使用します。

準備

まず、Windows* 11 に WSL2 と Ubuntu* 22.04 コンテナがインストールされている必要があります。インストール方法は、こちらを参照してください。Docker* Desktop on Windows* は、Docker* イメージをダウンロードし、WSL2 サブシステムを使用して Docker* コンテナを実行する簡単な方法を提供します。ITEX が設定済みのイメージを使用しますが、最初に Docker* を Windows* にインストールする必要があります。Docker* Desktop for Windows* を使用するか(インストール手順(英語))、WSL2 内で実行している Linux* に Docker* をインストールします(インストール例（英語))。

Docker* コンテナで Jupyter Notebook* を実行する

この例では、Ubuntu* (WSL2 内で実行) から Docker* を実行します。最初に、Ubuntu* WSL2 コンテナを起動します。Unix* シェルプロンプトから、ITEX 向けに事前に構築された Docker* イメージをプルします。

```
$ docker pull intel/intel-extension-for-tensorflow:xpu
```

ITEX Docker* イメージがダウンロードされていることを確認します。

```
$ docker images
REPOSITORY                                TAG       IMAGE ID       CREATED         SIZE
intel/intel-extension-for-tensorflow      xpu       2fc4b6a6fad7   8 days ago      7.42GB
```

次に、`docker run` コマンドを使用して、`intel/intel-extension-for-tensorflow:xpu` イメージでコンテナを開始します。

```
$ docker run  -ti   -p 9999:9999  --device /dev/dxg --mount type=bind,src=/usr/lib/wsl,dst=/usr/
lib/wsl -e LD_LIBRARY_PATH=/usr/lib/wsl/lib  intel/intel-extension-for-tensorflow:xpu
```

`docker run` コマンドのオプションは次のとおりです。

- `-p 9999:9999` は、Jupyter Notebook* を Windows* のブラウザーで使用できるように、Docker* コンテナ経由で標準ポートを渡します。
- `--device /dev/dxg` は、インテル® Arc™ A770 GPU にアクセスできるように、DirectX* ドライバーを Docker* コンテナに渡します。
- `--mount type=bind,src=/usr/lib/wsl,dst=/usr/lib/wsl -e LD_LIBRARY_PATH=/usr/lib/wsl/lib` は、WSL2 の共有ライブラリー・ディレクトリーを Docker* コンテナに渡します。

Docker* Desktop for Windows* を使用している場合は、PowerShell* またはコマンドプロンプトで上記の Docker* コマンドを直接実行できます。

ITEX Docker* コンテナを実行したら、Jupyter Notebook* を含むいくつかのパッケージを `pip` でインストールします。

```
root:/# pip install jupyter 'ipywidgets>=7.6.5' 'matplotlib>=3.3.4' scipy 'tensorflow_hub>=0.12.0'
'tensorflow-datasets>=4.4.0'
```

Jupyter Notebook* を開始します。

```
root:/# jupyter notebook --allow-root --ip 0.0.0.0 --port 9999
```

上記の Jupyter Notebook* コマンドを実行した後、Microsoft Edge* ブラウザーを使用して Jupyter Notebook * サーバーに接続します。Microsoft Edge* ブラウザーで、Jupyter* の出力でリストされた、次のような URL を開きます。

```
http://127.0.0.1:8888/?token=...
```

インテル® Arc™ A770 GPU を使用した転移学習

この例を Jupyter Notebook* で実行すると、トレーニング前、トレーニング中、トレーニング後に犬の画像がどのように分類されているか簡単に視覚化できます。各コードセクションをセルとして実行し、効果を確認します。各自の Notebook に従ってかまいません。

インポートする主なパッケージは、TensorFlow*、TensorFlow* Hub（標準の事前トレーニング済みモデルへのアクセスを提供）、および TensorFlow* Datasets（さまざまな用途に適した標準のトレーニングおよびテストセットを提供）です。また、ヘルパー関数で画像と分類の視覚化に使用するため、Matplotlib と NumPy* もインポートします。

```
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import numpy as np
```

```
[…]
2023-08-03 16:44:35.519530: I itex/core/devices/gpu/itex_gpu_runtime.cc:129] Selected platform:
Intel(R) Level-Zero
2023-08-03 16:44:35.519589: I itex/core/devices/gpu/itex_gpu_runtime.cc:154] number of sub-devices is
zero, expose root device.
```

ITEX プラグインは、TensorFlow* パッケージのインポート中に自動的に検出されます。上記の出力メッセージで示されているように、インテルの GPU が存在する場合は自動的にデフォルトのデバイス（インテル® oneAPI レベルゼロを使用）に設定されます。

次に、環境に設定できるパラメーターをいくつか示します。

```
batch_size = 32
dataset_directory = '/tmp/efficienetB0_datasets'    # ダウンロードしたデータセットの場所
output_directory = '/tmp/efficienetB0_saved_model'  # モデルを保存する場所
```

この実行では、TensorFlow* Hub の EfficientNetB0 を使用します。最後のレイヤーを削除した重み付きの事前トレーニング済みレイヤーであるヘッドレスモデル（「特徴ベクトルモデル」とも呼ばれます）のみを使用します。各画像の解像度も指定する必要があります。ここでは、EfficientNetB0 を 224 x 224 ピクセルの画像でトレーニングします。これは、データセットの画像のサイズを変更するために使用されます。

TensorFlow* Hub の EfficientNet モデルのドキュメントはこちら（英語）から入手できます。分類モデルと特徴ベクトルモデル（ヘッドレスモデル）の両方へのリンクが含まれています。

```
# TensorFlow Hub の "efficientnet_b0" モデル
model_handle = "https://tfhub.dev/google/efficientnet/b0/classification/1"
feature_vector_handle = "https://tfhub.dev/google/efficientnet/b0/feature-vector/1"
image_size = 224
```

次のヘルパー関数、`show_predictions()` は、予測されたラベルと実際のラベル、およびバッチ内の最初の 10 個の画像を視覚的に表示します。正しくラベル付けされた画像は緑で表示されます。誤ってラベル付けされた画像は赤で表示され、括弧内に正しいラベルが表示されます。転移学習のトレーニング中に進行状況を確認するため、いくつかのポイントで `show_predictions()` を使用します。

では、データセットを設定しましょう。ここでは、TensorFlow* Datasets を使用して、ImageNet で提供されている

```
# ヘルパールーチンで正しい予測にビジュアル・インジケーターを追加してバッチの 10 の画像を表示
def show_predictions(image_batch, label_batch, class_names):

    predicted_batch = model.predict(image_batch)  # サンプルバッチを使用して予測
    predicted_id = np.argmax(predicted_batch, axis=-1)
    predicteds = [class_names[id] for id in predicted_id]

    actuals = [class_names[int(id)] for id in label_batch]

    print("Correct predictions are shown in green")
    print("Incorrect predictions are shown in red with the actual label in parenthesis")

    plt.figure(figsize=(10,9))  # 結果を表示
    plt.subplots_adjust(hspace=0.5)
    for n in range(0, 10):  # (batch_size - 2) を使用するとバッチ全体を出力
        plt.subplot(6,5,n+1)
        plt.imshow(image_batch[n])
        correct_prediction = actuals[n] == predicteds[n]
        color = "darkgreen" if correct_prediction else "crimson"
        title = predicteds[n].title() \
            if correct_prediction \
            else "{}\\n({})".format(predicteds[n], actuals[n])
        plt.title(title, fontsize=9, color=color)
        plt.axis('off')
    _ = plt.suptitle("Model predictions")
    plt.show()
```

ものより正確な犬種のラベルが付けられた犬の画像を含む「Stanford Dogs」データセットをロードします。このコードを初めて実行すると、上記で指定したディレクトリーにデータセットがダウンロードされます。完了するまで 5 ~ 10 分かかります。ダウンロードの状況は、プログレスバーで確認できます。同じ Docker* コンテナ内で続けて実行する場合は、キャッシュされたデータセットが使用されるため、高速に実行されます。

データセットが利用可能になると、トレーニング (75%) とテスト (25%) に分割されます。次に、それぞれキャッシュされ、バッチに分割され、プリフェッチに設定されます。トレーニング・データセットもシャッフルされるため、Dense レイヤーのトレーニング速度にランダム性が生じます。最後に、後で使用できるようにデータセットのクラス名が保存されます。

Stanford Dogs データセットはこちら（英語）から入手できます。このデータセットが選択された理由は、転移学習で 90% までトレーニングするのに数エポックかかる大きさでありながら、妥当な時間でダウンロードできるためです。TF-Datasets には、小規模なものから大規模なものまで、使用できるデータセットがほかにもあります。

```
[train_ds, test_ds], info = tfds.load("stanford_dogs",  # TensorFlow Datasets からデータセットをロード
                        data_dir=dataset_directory,
                        split=["train[:75%]", "train[:25%]"],
                        as_supervised=True,
                        shuffle_files=True,
                        with_info=True)

def preprocess_image(image, label):  # 画像を float32 に変換してモデルと一致するようにサイズ変更
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize_with_pad(image, image_size, image_size)
    return (image, label)

train_ds = train_ds.map(preprocess_image)
test_ds = test_ds.map(preprocess_image)

# ランダムになるようにトレーニング・データをシャッフル
train_ds = train_ds.cache()
train_ds = train_ds.shuffle(info.splits['train'].num_examples)
train_ds = train_ds.batch(batch_size)
train_ds = train_ds.prefetch(tf.data.AUTOTUNE)

# テストデータはシャッフル不要、キャッシュはバッチ後に完了
test_ds = test_ds.batch(batch_size)
test_ds = test_ds.cache()
test_ds = test_ds.prefetch(tf.data.AUTOTUNE)

class_names = info.features["label"].names  # データセットのクラス名を取得
```

次に、TensorFlow* Hub からヘッドレス EfficientNetB0 モデルをロードします。このモデルに含まれる、(ImageNet データセットでのトレーニングに基づく) 事前トレーニング済みの重みは変更しません。完全にトレーニングされる Dense レイヤーも追加します。このレイヤーのサイズは、データセットのクラスの数に合わせて設定されます。次にモデルがコンパイルされ、形状が出力されます。ヘッドレスモデルは、事前トレーニング済みの重みを含む多くのレイヤーで構成されているにもかかわらず、単一の KerasLayer として表示されることに注意してください。完全なモデルには 4,203,284 のパラメーターがありますが、そのうち 153,720 のパラメーターのみトレーニングされ、残りは変更されません。

```
# 特徴ベクトルレイヤーはトレーニングしない （すでにトレーニング済み）
feature_extractor_layer = hub.KerasLayer(feature_vector_handle,
                                         input_shape=(image_size, image_size, 3),
                                         trainable=False)
model = tf.keras.Sequential([
    feature_extractor_layer,
    tf.keras.layers.Dense(len(class_names))  # トレーニングする Dense レイヤーを追加
])

model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['acc'])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)    (None, 1280)              4049564

 dense (Dense)               (None, 120)               153720

=================================================================
Total params: 4,203,284
Trainable params: 153,720
Non-trainable params: 4,049,564
_________________________________________________________________
```

トレーニングを開始する前に、トレーニングされていない Dense レイヤーを使用して、モデルが適切な犬種ラベルをどの程度予測できるか確認してみましょう。まず、テスト・データセットで推論を実行します。

```
with tf.device('/xpu:0'):  model.evaluate(test_ds, batch_size=batch_size)
```

```
94/94 [==============================] - 29s 158ms/step - loss: 2.4268 - acc: 0.0087
```

次に、ヘルパー関数を使用して、最初のバッチの最初の 10 枚の画像を表示します。

```
batch = next(iter(test_ds))  # テストに使用するデータセットのバッチを取得
image_batch, label_batch = batch
show_predictions(image_batch, label_batch, class_names)
```

```
1/1 [==============================] - 1s 834ms/step
      Correct predictions are shown in green
      Incorrect predictions are shown in red with the actual label in parenthesis
```

Dense レイヤーをトレーニングしていない場合、予想どおり、すべての予測は不正確で、赤で表示されています。上のラベルは誤った分類で、下のラベル（括弧内）は正しい予測です。

では、トレーニングを行ってみましょう。Dense レイヤーの重みのみ調整します。まず、ラベル予測に対する完全でないトレーニングの効果を後から確認できるように、1 エポックのみ実行します。使用されているデバイスは `/xpu:0`（つまり、ホスト PC にインストールされているインテル® Arc™ A770 GPU）であることに注意してください。

```
with tf.device('/xpu:0'):  model.fit(train_ds, epochs=1, shuffle=True, verbose=1)
```

```
282/282 [==============================] - 28s 85ms/step - loss: 0.8545 - acc: 0.6581
```

1 エポック後、転移学習を使用して Dense レイヤーのみトレーニングしただけで、精度は 65.81% になりました。Stanford Dogs を使用して EfficientNetB0 を最初から完全にトレーニングした場合、最初のエポック後の精度は約 1.99% にすぎません。この最初のエポックには約 28 秒かかることにも注意してください。これは、最初のエポックでは GPU をウォームアップする（データを GPU メモリーに転送して、後の実行のために保存する）必要があり、後のエポックよりも遅くなるためです。

では、最初のバッチの予測を見てみましょう。まず、テスト・データセットに推論を実行します。

```
with tf.device('/xpu:0'):  model.evaluate(test_ds, batch_size=batch_size)
```

```
94/94 [==============================] - 4s 41ms/step - loss: 0.3159 - acc: 0.8697
```

次に、ヘルパー関数を使用して予測を表示します。

```
show_predictions(image_batch, label_batch, class_names)
```

```
1/1 [==============================] - 0s 65ms/step
Correct predictions are shown in green
Incorrect predictions are shown in red with the actual label in parentheses
```

バッチの最初の 10 枚の画像のうち 9 枚が正しく予測されました。これは、テスト・データセットの推論の精度が 86.97% に向上したためです。トレーニングの精度を向上させると、より多くの画像が正しく予測されるようになります。実行では、トレーニング・データセットがシャッフルされ、トレーニングの精度にある程度のランダム性が生じるため、多少の予測ミスが発生する可能性があります。

では、さらに 2 つのトレーニング・エポックを実行してみましょう。

```
with tf.device('/xpu:0'):  model.fit(train_ds, epochs=2, shuffle=True, verbose=1)
```

```
Epoch 1/2
282/282 [==============================] - 12s 41ms/step - loss: 0.2812 - acc: 0.8581
Epoch 2/2
282/282 [==============================] - 12s 40ms/step - loss: 0.1967 - acc: 0.9029
```

わずか 3 エポックのトレーニングの後、転移学習の精度は 90.29% まで向上しました。インテル® Arc™ A770 GPU で Stanford Dogs を使用して EfficientNetB0 を最初から完全にトレーニングした場合、90% の精度に達するには約 30 エポックかかります。つまり、転移学習を使用すると、より速く高い精度に達することができます。

最初のエポックでデータが GPU メモリーにコピーされていたため、以降の 2 つのエポックはそれぞれインテル® Arc™ A770 GPU でわずか 12 秒で完了しました。Dense レイヤーのみトレーニングすればよいことも転移学習でエポックの完了が速くなる理由の 1 つです。インテル® Arc™ A770 GPU で EfficientNetB0 のすべてのレイヤーを完全にトレーニングすると、(最初のエポックの後) エポックごとに約 61 秒かかります。転移学習は、トレーニング時間の短縮 (収束までのエポック数の減少) とエポック数の短縮 (トレーニングするパラメーター数の減少) の両方を実現します。

インテル® Arc™ A770 GPU は、インテル® Core™ i9 CPU に比べて大幅なスピードアップを達成します。結果を比較したところ、GPU での転移学習のトレーニングは CPU よりも 10 倍以上高速でした。

では、わずか 3 エポックのトレーニング後に予測がどのようになるか見てみましょう。

```
with tf.device('/xpu:0'):  model.evaluate(test_ds, batch_size=batch_size)
```

```
94/94 [==============================] - 4s 42ms/step - loss: 0.1411 - acc: 0.9410
```

テスト・データセットの精度はすでに 94.10% に達しています。

```
show_predictions(image_batch, label_batch, class_names)
```

```
1/1 [==============================] - 0s 56ms/step
Correct predictions are shown in green
Incorrect predictions are shown in red with the actual label in parentheses
```

モデルの予測

94.10% の精度で、テスト・データセットの最初の 10 枚の画像がすべて正しく予測されました。インテル® Arc™ A770 GPU で Stanford Dogs を使用して EfficientNetB0 を最初から完全にトレーニングすると、90% の精度に達するまで 30 エポックで約 31 分かかります。転移学習を使用すると、わずか数分で同じ結果を得ることができます。