Section3: 軽量化・高速化技術

1. 要点まとめ

1.1 高速化

深層学習のモデルは年々複雑化／肥大化しているが、モデル生成を行う計算資源（CPU,GPU等）の進化はそれに追いついていない。したがって、複数の計算資源を使って高速にモデル生成を行う分散深層学習の技術が求められている。

複数の計算資源を使ってモデル学習を高速化する手法としては主に以下がある。

- データ並列化
- モデル並列化
- GPUによる高速化

データ並列化は、データセットを分割して複数のワーカー（PC、CPU、GPU等）に配置し、１つのモデルを各ワーカーにコピーしてワーカー毎にモデル学習を行う手法である。パラメータ更新の度に同期を取る同期型と、同期を取らずに各々のワーカーでパラメータ更新を進め更新済パラメータをパラメータサーバーを介してやりとりする非同期型がある。

非同期型のほうが、待たなくてよい分早いが、ベストモデルを次の学習に使えるという保障がない分、学習が不安定になりやすい。

モデル並列化は、モデルを分割して複数のワーカーに配置し、１つのデータセットを各ワーカーにコピーしてワーカー毎にモデル学習を行う手法である。誤差関数の計算等では各ワーカーのモデル出力をワーカー間でやり取りする必要が生じることから、PCを別にすると大量の通信が発生しボトルネックとなってしまう。よって、この手法をとる場合は、ワーカー＝同一PC内の複数CPU、とするのが主流である。

GPUによる高速化は、GPU内部に多数配置されている低性能コアを使って高速化を図る手法である。 深層学習モデルは１つ１つの演算は線形代数主体の簡易な演算のためCPUのような高性能なコアを必要としていない。その代わりに演算量が膨大なため多数のコアを必要とする。そうした深層学習モデルの特性にGPUはマッチするため、現在多用される手法である。

GPUを扱う主なフレームワークはCUDAとOpenCLがあるが、CUDAが主流となっている。 tensorflowなどのディープラーニングの主要フレームワークはCUDAに対応済であり、CUDAプログラミングのスキルがなくてもGPUによる高速化を行える。

1.2 軽量化

軽量化とは、モデルの精度を維持しつつ、パラメータや演算回数を提言する手法であり、主に以下がある。

- 量子化
- 蒸留
- プルーニング

量子化とは、数値のbit長を落とす（浮動小数64bit→32bit）ことで、メモリと処理時間の削減を図ることである。現在多く用いられる浮動小数のbit長は、倍精度(64bit)、単精度(32bit)、半精度(16bit)である。bit長を落とすと計算精度が低下するが、ほとんどの機械学習モデルではそれほどの精度を必要としておらず、半精度(16bit)で十分である。

蒸留とは、規模が大きなモデルから軽量モデルを作ることである。学習済の精度の高いモデルの知識を軽量なモデルへ継承させることで実現する。具体的な方法は以下。

1. 教師モデル（学習済の精度の高いモデル）と生徒モデル（軽量モデル）の２つを用意
2. 順伝播を教師モデル、生徒モデル両方で行い、誤差を出力
3. 誤差の逆伝播を生徒モデルのほうのみ行い、生徒モデルのパラメータを更新する。教師モデルは逆伝播を行わない

プルーニングとは、推論に必要なパラメータだけを残してあとは削除する手法である。具体的には、重みが小さい（閾値以下）のパラメータを削除し、削除後に最終出力に至らなくなったパス上のパラメータも削除する。パラメータを削除するとモデルの精度低下が懸念されるが、精度低下の度合は意外と小さく、9割削除しても精度は10%程度しか落ちない、という実験結果もある。

2. 実装演習

TensorFlowチュートリアル掲載コードを参考に、float16量子化モデルを作成する。量子化なしモデルとの精度、処理速度、サイズ比較を行い、量子化効果を確認する。

```
# 参考: TensorFlowチュートリアル
#  https://www.tensorflow.org/lite/performance/post_training_float16_quant?hl=ja

import logging
logging.getLogger("tensorflow").setLevel(logging.DEBUG)

import tensorflow as tf
from tensorflow import keras
import numpy as np
import pathlib

# Load MNIST dataset
mnist = keras.datasets.mnist
(train_images, train_labels), (test_imgs, test_lbls) = mnist.load_data()

# Normalize the input image so that each pixel value is between 0 to 1.
train_images = train_images / 255.0
test_imgs = test_imgs / 255.0

# ----------------
#  通常モデル作成
# ----------------
# Define the model architecture
model = keras.Sequential([
  keras.layers.InputLayer(input_shape=(28, 28)),
  keras.layers.Reshape(target_shape=(28, 28, 1)),
  keras.layers.Conv2D(filters=12, kernel_size=(3, 3), activation=tf.nn.relu),
  keras.layers.MaxPooling2D(pool_size=(2, 2)),
  keras.layers.Flatten(),
  keras.layers.Dense(10)
])

# Train the digit classification model
model.compile(optimizer='adam',
              loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.fit(
  train_images,
  train_labels,
  epochs=1,
  validation_data=(test_imgs, test_lbls)
)

converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
```

```
# ファイルダンプ
tflite_models_dir = pathlib.Path("./fig_section3/")
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_file = tflite_models_dir/"mnist_model.tflite"
tflite_model_file.write_bytes(tflite_model)

# モデルをインタープリタに読み込む
interpreter = tf.lite.Interpreter(model_path=str(tflite_model_file))
interpreter.allocate_tensors()

# --------------------
#  量子化モデルに変換
# --------------------
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]

tflite_fp16_model = converter.convert()

# ファイルダンプ
tflite_model_fp16_file = tflite_models_dir/"mnist_model_quant_f16.tflite"
tflite_model_fp16_file.write_bytes(tflite_fp16_model)

# モデルをインタープリタに読み込む
interpreter_fp16 = tf.lite.Interpreter(model_path=str(tflite_model_fp16_file))
interpreter_fp16.allocate_tensors()

# -----------------
#  モデル比較
# -----------------
import time

# A helper function to evaluate the TF Lite model using "test" dataset.
def evaluate_model(model_interp, test_images, test_labels):
    input_index = model_interp.get_input_details()[0]["index"]
    output_index = model_interp.get_output_details()[0]["index"]

    # Run predictions on every image in the "test" dataset.
    start = time.time()

    prediction_digits = []
    for test_image in test_images:
        # Pre-processing: add batch dimension and convert to float32 to match with
        # the model's input data format.
        test_image = np.expand_dims(test_image, axis=0).astype(np.float32)
        model_interp.set_tensor(input_index, test_image)

        # Run inference.
        model_interp.invoke()

        # Post-processing: remove batch dimension and find the digit with highest
        # probability.
        output = model_interp.tensor(output_index)
        digit = np.argmax(output()[0])
        prediction_digits.append(digit)
```

```
    elapsed_time = (time.time() - start) * 1000

    # Compare prediction results with ground truth labels to calculate accuracy.
    accurate_count = 0
    for index in range(len(prediction_digits)):
        if prediction_digits[index] == test_labels[index]:
            accurate_count += 1
    accuracy = accurate_count * 1.0 / len(prediction_digits)

    return accuracy, elapsed_time

# 通常モデルの評価
acc_normal, time_normal = evaluate_model(interpreter, test_imgs, test_lbls)

# 量子化モデルの評価
acc_quant, time_quant = evaluate_model(interpreter_fp16, test_imgs, test_lbls)

print("Normal Model : accuracy=", acc_normal,
        ", time[ms]=", time_normal)

print("Quantization Model(fp16) : accuracy=", acc_quant,
        ", time[ms]=", time_quant)
```

実行結果は以下。

- 精度、処理時間ともに、通常モデルと量子化モデルでほぼ差がない
- ファイルサイズが、量子化モデル(float16)は通常モデルの約半分
    - ⇒　半分のサイズで同等の性能（精度、処理時間）を発揮できている

モデル	**精度(accuracy)**	**処理時間[ms]**	**ファイルサイズ[KB]**
通常モデル	0.9601	1941	83
量子化モデル(float16)	0.9601	1938	44

3. 確認テスト

※section3は確認テストなし