Vitis AI オプティマイザー ユーザー ガイド

UG1333 (v1.3) 2021 年 2 月 3 日

XILINX®

改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	改訂内容
2021 年 2 月 3 日、バージョン 1.3	
トレーニング用データセットの準備	新規セクションを追加。
2020 年 12 月 17 日、バージョン 1.3	
文書全体	マイナー変更。
PyTorch バージョン - vai_p_pytorch	新規セクションを追加。
第 4 章: ネットワークの例	PyTorch の例 を追加。
2020 年 7 月 7 日 バージョン 1.2	
文書全体	マイナー変更。
2020 年 3 月 23 日 バージョン 1.1	
文書全体	マイナー変更。
VAI プルーナーのライセンス	新たにトピックを追加。

目次

第 1 章

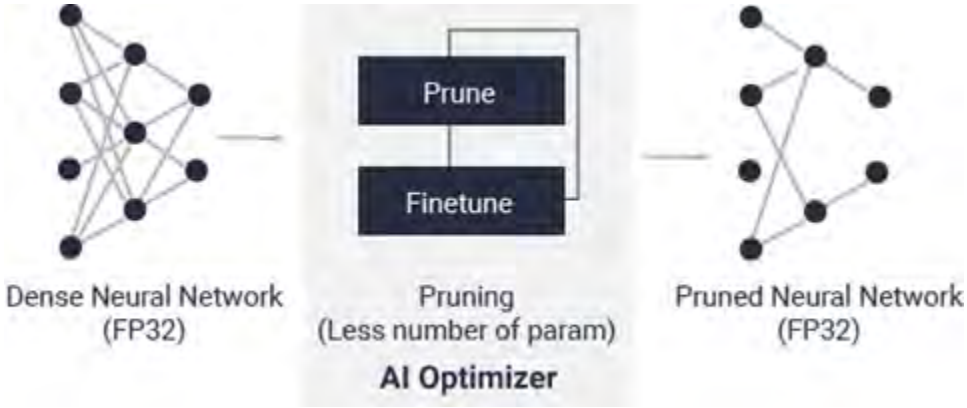
概要とインストール

Vitis AI オプティマイザーの概要

Vitis™ AI は、ザイリンクス ハードウェア プラットフォーム上で AI 推論を実行するためのザイリンクス開発キットです。機械学習の推論は計算負荷が非常に高いため、各種アプリケーションで低レイテンシかつ高スループットという要件を満たすには、広いメモリ帯域幅が必要です。

Vitis AI オプティマイザーは、ニューラル ネットワーク モデルを最適化します。現在、Vitis AI オプティマイザーに含まれるのはプルーナーと呼ばれるツールのみです。Vitis AI プルーナー (VAI プルーナー) は、冗長な接続を刈り込み (プルーニング)、必要な演算数を全体的に削減します。VAI プルーナーで生成したプルーニング済みモデルは、VAI クオンタイザーで量子化して FPGA 上で運用できます。VAI クオンタイザーと運用の詳細は、『Vitis AI ユーザー資料』 (UG1431) の 『Vitis AI ユーザー ガイド』 を参照してください。

図 1: **VAI オプティマイザー**



VAI プルーナーは、TensorFlow、PyTorch、Caffe、および Darknet の 4 つの深層学習フレームワークをサポートしています。それぞれに対応するツール名は、vai_p_tensorflow、vai_p_pytorch、vai_p_caffe、および vai_p_darknet です。「p」はプルーニングを表します。

Vitis AI オプティマイザーを実行するには、コマーシャル ライセンスが必要です。Vitis AI オプティマイザーのインストール パッケージおよびライセンスをご希望の方は、xilinx_ai_optimizer@xilinx.com までお問い合わせください。

設計プロセス別のコンテンツ ガイド

ザイリンクスの資料は、開発タスクに関連する内容を見つけやすいように、標準設計プロセスに基づいて構成されています。Versal™ ACAP デザイン プロセスのデザイン ハブは、ザイリンクス ウェブサイトからアクセスできます。この資料では、次の設計プロセスについて説明します。

- 機械学習とデータ サイエンス: Caffe、Pytorch、TensorFlow、またはその他のよく使用されるフレームワークから機械学習モデルを Vitis™ AI にインポートし、その効果を最適化および評価します。この設計プロセスに該当するトピックは、次のとおりです。
  - 第 2 章: プルーニング
  - 第 3 章: VAI プルーナーの使用方法
  - 第 4 章: ネットワークの例

インストール

Vitis AI オプティマイザーは、次の 2 通りの方法で入手できます。

- Docker イメージ: Vitis AI はオプティマイザー用の Docker 環境を提供しています。Docker イメージには 3 つのオプティマイザー関連の Conda 環境があります (vitis-ai-optimizer_tensorflow、vitis-ai-optimizer_caffe、および vitis-ai-optimizer_darknet)。これらの環境では、すべての要件が整っています。Docker での CUDA および cuDNN のバージョンは、それぞれ CUDA 10.0 と cuDNN 7.6.5 です。ライセンスの取得後、Docker で VAI プルーナーを直接実行できます。

  注記: PyTorch 用のオプティマイザーは Docker イメージに含まれていないため、Conda パッケージを使用してのみインストールできます。

- Conda パッケージ: Conda パッケージは、Ubuntu 18.04 でも利用できます。Vitis AI オプティマイザーのインストール パッケージおよびライセンスをご希望の方は、xilinx_ai_optimizer@xilinx.com までお問い合わせください。インストール手順に従って、前提となる要件および Vitis AI オプティマイザーをインストールする必要があります。

ハードウェア要件

Nvidia GPU カード (CUDA Compute Capability 3.5 以上) が必要です。Tesla P100 または Tesla V100 の使用を推奨します。

ソフトウェア要件

注記: このセクションは、Conda パッケージをインストールする場合にのみ必要です。Docker イメージの場合は、このセクションは飛ばしてください。依存ライブラリは既に Docker イメージ内にあります。

- GPU 関連のソフトウェア: 各オペレーティング システムで必要な GPU 関連ソフトウェアをインストールします。Ubuntu 16.04 の場合、CUDA 9.0、cuDNND 7 およびドライバー 384 以降をインストールします。Ubuntu 18.04 の場合、CUDA 10.0、cuDNN 7 およびドライバー 410 以降をインストールします。

- NVIDIA GPU ドライバー:

apt-get コマンドで GPU ドライバーをインストールするか、ドライバーを含む CUDA パッケージを直接インストールします。次に例を示します。

```
apt-get install nvidia-384
apt-get install nvidia-410
```

- CUDA Toolkit:

  https://developer.nvidia.com/cuda-toolkit-archive から Ubuntu のバージョンに応じた CUDA パッケージを入手し、NVIDIA CUDA runfile パッケージを直接インストールします。

- cuDNN SDK: https://developer.nvidia.com/cudnn から cuDNN を入手し、インストール ディレクトリを環境変数 `$LD_LIBRARY_PATH` に追加します。

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cudnn-7.0.5/lib64
  ```

- CUPTI: `vai_p_tensorflow` では CUPTI が必要です。これは CUDA と一緒にインストールされます。環境変数 `$LD_LIBRARY_PATH` に CUPTI ディレクトリを追加する必要があります。次に例を示します。

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/extras/CUPTI/lib64
  ```

- NCCL: `vai_p_caffe` では NCCL が必要です。https://developer.nvidia.com/nccl/nccl-legacy-downloads から NCCL をダウンロードして、インストールします。

  ```
  sudo dpkg -i nccl-repo-ubuntu1804-2.6.4-ga-cuda10.0_1-1_amd64.deb
  sudo apt update
  sudo apt install libnccl2=2.6.4-1+cuda10.0 libnccl-dev=2.6.4-1+cuda10.0
  ```

VAI プルーナー

注記: VAI プルーナーをインストールするには、まず Conda パッケージをインストールする必要があります。Docker イメージの場合は、次のセクションは飛ばしてください。VAI プルーナーは、対応する Conda 環境に含まれています。

VAI プルーナーをインストールするには、最初に Conda をインストールし、次にフレームワーク用の Conda パッケージをインストールします。

Conda のインストール

詳細は、Conda インストール ガイド を参照してください。

vai_optimizer_tensorflow

vai_p_tensorflow は TensorFlow 1.15 をベースにしています。この vai_optimizer_tensorflow パッケージをインストールし、vai_p_tensorflow を取得します。

```
$ tar xzvf vai_optimizer_tensorflow.tar.gz
$ conda install vai_optimizer_tensorflow_gpu -c file://$(pwd)/vai-bld -c
conda-forge/label/gcc7 -c conda-forge
```

vai_optimizer_pytorch

vai_optimizer_pytorch は Python ライブラリであり、API を呼び出すことで使用できます。

```
$ tar xzvf vai_optimizer_pytorch.tar.gz
$ conda install vai_optimizer_pytorch_gpu -c file://$(pwd)/vai-bld -c
pytorch
```

vai_optimizer_caffe

vai_p_caffe バイナリは、この vai_optimizer_caffe Conda パッケージに含まれています。

```
$ tar xzvf vai_optimizer_caffe.tar.gz
$ conda install vai_optimizer_caffe_gpu -c file://$(pwd)/vai-bld -c conda-
forge/label/gcc7 -c conda-forge
```

vai_optimizer_darknet

```
$ tar xzvf vai_optimizer_darknet.tar.gz
$ conda install vai_optimizer_darknet_gpu -c file://$(pwd)/vai-bld
```

VAI プルーナーのライセンス

ライセンスには、フローティング ライセンスとノード ロック ライセンスの 2 種類があります。VAI プルーナーは、環境変数 `XILINXD_LICENSE_FILE` を使用してライセンスを検出します。フローティング ライセンス サーバーには、port@hostname の形式でパスを指定する必要があります。例: `export XILINXD_LICENSE_FILE=2001@xcolicsvr1`。ノード ロック ライセンス ファイルには、特定のライセンス ファイル、またはすべての `.lic` ファイルが置かれているディレクトリを指定する必要があります。

特定のファイルを指定する場合は、次のコマンドを実行します。

```
export XILINXD_LICENSE_FILE=/home/user/license.lic
```

ディレクトリを指定する場合は、次のコマンドを実行します。

```
export XILINXD_LICENSE_FILE=/home/user/license_dir
```

複数のライセンスがある場合は、各ライセンスをコロンで区切って一度に指定できます。

```
export XILINXD_LICENSE_FILE=1234@server1:4567@server2:/home/user/license.lic
```

ノード ロック ライセンスは、`$HOME/.Xilinx directory` にコピーすることによってもインストール可能です。

XILINX

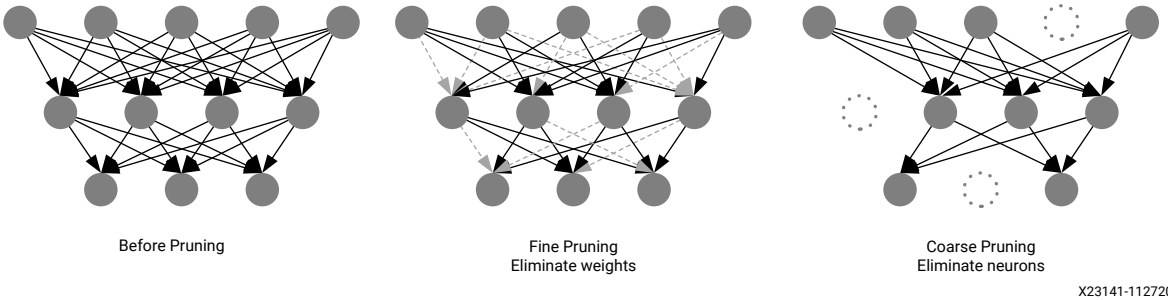
第 2 章

プルーニング

プルーニングの概要

ほとんどのニューラル ネットワークは、特定の精度を達成するために、冗長性が高く、過剰にパラメーター化されています。「プルーニング」は、精度の低下をできるだけ低く抑えながら、過剰な重みを排除するプロセスです。

図 2: **プルーニング手法**



最も単純なプルーニングは、「細粒度プルーニング」と呼ばれ、結果としてスパース型の重み行列になります。Vitis AI プルーナーは「粗粒度プルーニング」を採用しています。これは、ネットワークの精度に大きく寄与しないニューロンを排除する手法です。たたみ込み層では、粗粒度プルーニングによって 3D カーネル全体がプルーニングされるため、チャネル プルーニングとも呼ばれます。

プルーニングを実行すると、元のモデルの精度が低下します。再学習 (微調整) することで、残りの重みを調整して精度を回復します。

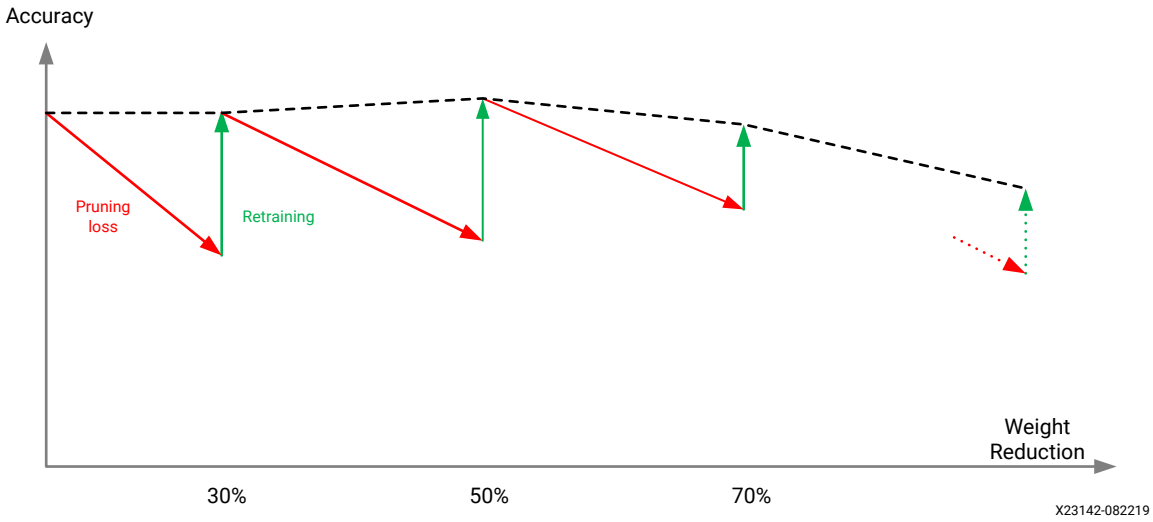
反復プルーニング

Vitis AI プルーナーは、精度の低下を最小限に抑えながらモデルのパラメーター数を削減するように設計されています。これは、次の図に示すように反復的なプロセスによって達成します。プルーニングによって低下した精度を再学習によって回復します。プルーニングと再学習を 1 つのセットとして、これを反復実行します。最初の反復では、ベースライン モデルを入力モデルとして、プルーニングと微調整を実行します。その後の反復では、直前の反復で生成された微調整済みモデルを使用して、プルーニングと微調整を繰り返します。通常、目的のスパース モデルを得るにはこのプロセスを何度か繰り返す必要があります。1 回のプルーニングでモデルのサイズを大幅に削減することはできません。モデルから大量のパラメーターを削減すると、モデルの精度が低下し、回復が難しくなります。

重要: 微調整によって精度を回復しやすくするために、リダクション パラメーター値は、反復ごとに徐々に大きくする必要があります。

プルーニングを何度か反復すると、モデルの精度を大きく低下させることなく、高いプルーニング率を達成できます。

図 3: **プルーニングの反復プロセス**



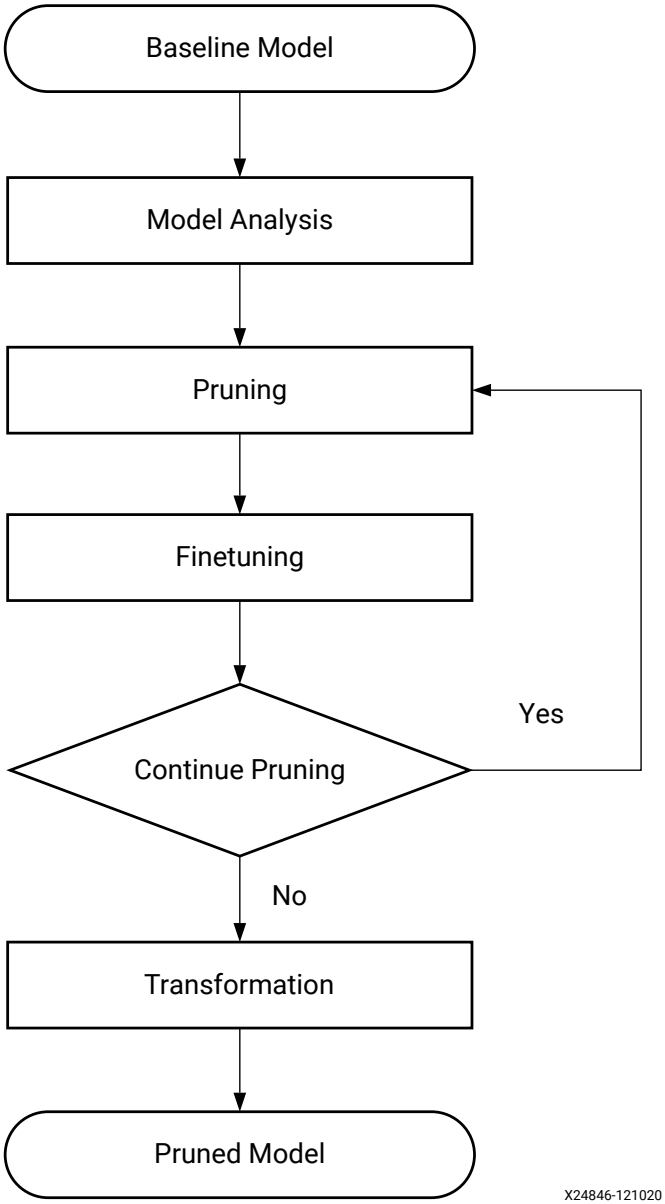
Vitis AI プルーナーの主なタスクは、次の 4 つです。

1. 解析 (ana): モデルの感度分析を実行して、最適なプルーニング手法を決定します。
2. プルーニング (prune): 入力モデルの演算数を削減します。
3. 微調整 (finetune): 再学習により、精度を回復します。
4. 変換 (transform): 重みが削減されたデンス (密) モデルを生成します。

次の手順に従ってモデルをプルーニングします。次の図にもこれらの手順を示します。

1. 元のベースライン モデルを解析する。
2. モデルをプルーニングする。
3. プルーニング済みモデルを微調整する。
4. 手順 2 と 3 を数回繰り返す。
5. プルーニング済みのスパース (疎) モデルを最終的なデンス (密) モデルに変換する。

図 4: **プルーニング ワークフロー**



よりよいプルーニング結果を得るためのガイドライン

次に、精度の低下を最小に抑えながらプルーニング率を高め、よりよいプルーニング結果を得るための推奨事項を示します。

1. モデルの解析には、できるだけ多くのデータを使用します。理想的には、検証データセットのすべてのデータを使用する必要がありますが、これにはかなりの時間を要する可能性があります。検証データセットの一部のみを使用することも可能ですが、少なくとも半分以上は使用する必要があります。

2. 微調整段階では、初期学習率、学習率の減衰ポリシーなどのいくつかのパラメーターを試し、最良の結果を次のプルーニングへの入力として使用します。
3. 微調整で使用するデータは、ベースライン モデルの学習に使用したのと同じである必要があります。
4. 微調整を複数回実行しても精度が十分に向上しない場合は、プルーニング率を下げてからプルーニングと微調整を再度実行してください。

XILINX

第 3 章

VAI プルーナーの使用方法

TensorFlow バージョン - vai_p_tensorflow

推論グラフをエクスポートする

まず、学習用と評価用の TensorFlow グラフ作成のコードを、別々のスクリプトで記述する必要があります。既にベースライン モデルの学習が完了している場合は、学習用コードはあるため、評価用のコードのみを作成します。評価用スクリプトには、`model_fn` という名前の関数を含める必要があります。この関数は、入力から出力までに必要なすべてのノードを作成します。この関数は、出力ノードの名前をそれぞれの演算または tf.estimator.Estimator にマップするディクショナリを返します。たとえば画像分類ネットワークの場合、次のスニペットに示すように、通常は top-1 および top-5 の精度を計算する演算を含むディクショナリが返されます。

```
def model_fn():
  # graph definition codes here
  # ……
return {
      'top-1': slim.metrics.streaming_accuracy(predictions, labels),
      'top-5': slim.metrics.streaming_recall_at_k(logits, org_labels, 5)
  }
```

TensorFlow Estimator API を使用してネットワークの学習と評価を実行する場合は、`model_fn` は tf.estimator のインスタンスを返す必要があります。これと同時に、`eval_input_fn` という名前の関数も用意しておく必要があります。Estimator は、この関数を使用して評価で使用するデータを取得します。

```
def cnn_model_fn(features, labels, mode):
  # codes for building graph here
…
eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)

def model_fn():
  return tf.estimator.Estimator(
      model_fn=cnn_model_fn, model_dir="./models/train/")

mnist = tf.contrib.learn.datasets.load_dataset("mnist")
train_data = mnist.train.images # Returns np.array
train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
eval_data = mnist.test.images # Returns np.array
eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

def eval_input_fn():
```

```
  return tf.estimator.inputs.numpy_input_fn(
      x={"x": eval_data},
      y=eval_labels,
      num_epochs=1,
      shuffle=False)
```

この評価用コードを使用して推論 `GraphDef` ファイルをエクスポートし、プルーニング中にネットワークの精度を評価します。`GraphDef` proto ファイルをエクスポートするには、次のコードを使用します。

```
import tensorflow as tf
from google.protobuf import text_format
from tensorflow.python.platform import gfile

with tf.Graph().as_default() as graph:
# your graph definition here
# ……
    graph_def = graph.as_graph_def()
    with gfile.GFile('inference_graph.pbtxt', 'w') as f:
      f.write(text_format.MessageToString(graph_def))
```

モデル解析を実行する

モデルのプルーニングを実行する前に、まずモデルを解析する必要があります。このプロセスの主な目的は、モデルをプルーニングする際の適切なプルーニング手法を見つけることです。

モデル解析を実行するには、モデルの精度を評価する関数を含んだ Python スクリプトを作成する必要があります。このスクリプト名を `eval_model.py` とすると、次に示す 3 つのうちいずれかの方法で、必要な関数を作成する必要があります。

- メトリクス演算の Python ディクショナリを返す `model_fn()` という名前の関数。

```
def model_fn():
  tf.logging.set_verbosity(tf.logging.INFO)
  img, labels = get_one_shot_test_data(TEST_BATCH)

  logits = net_fn(img, is_training=False)
  predictions = tf.argmax(logits, 1)
  labels = tf.argmax(labels, 1)
  eval_metric_ops = {
      'accuracy': tf.metrics.accuracy(labels, predictions),
      'recall_5': tf.metrics.recall_at_k(labels, logits, 5)
  }
  return eval_metric_ops
```

- tf.estimator.Estimator のインスタンスを返す `model_fn()` という名前の関数、およびテスト データをエスティメーターに供給する `eval_input_fn()` という名前の関数。

```
def model_fn():
  return tf.estimator.Estimator(
      model_fn=cnn_model_fn, model_dir="./models/train/")

def eval_input_fn():
  return tf.estimator.inputs.numpy_input_fn(
      x={"x": eval_data},
      y=eval_labels,
      num_epochs=1,
      shuffle=False)
```

- 1 つのパラメーターを引数にとり、メトリクス スコアを返す `evaluate()` という名前の関数。

```
def evaluate(checkpoint_path):
  with tf.Graph().as_default():
    net = ConvNet(False)
    net.build(test_only=True)
    score = net.evaluate(checkpoint_path)
    return score
```

1 番目の方法を使用してスクリプトを作成する場合、次のスニペットに示す方法で `vai_p_tensorflow` を呼び出し、モデル解析を実行します。

```
vai_p_tensorflow \
  --action=ana \
  --input_graph=inference_graph.pbtxt \
  --input_ckpt=model.ckpt \
  --eval_fn_path=eval_model.py \
  --target="recall_5" \
  --max_num_batches=500 \
  --workspace:/tmp \
  --exclude="conv node names that excluded from pruning" \
  --output_nodes="output node names of the network"
```

次に、このコマンドの引数について説明します。オプションの一覧は、vai_p_tensorflow の使用法 を参照してください。

- --action: 実行するアクション。
- --input_graph: ネットワークの推論グラフを表す `GraphDef` proto ファイル。
- --input_ckpt: プルーニングに使用するチェックポイントのパス。
- --eval_fn_path: 評価用グラフを定義する Python スクリプトのパス。
- --target: ネットワークの精度を評価するターゲット スコア。1 つのネットワークに複数のスコアがある場合、最も重要なものを選びます。
- --max_num_batches: 評価フェーズで実行するバッチの数。このパラメーターは、モデルの解析時間に影響します。この値が大きいほど解析に時間がかかり、解析精度は向上します。このパラメーターは、検証データセットのサイズまたは batch_size を最大値として設定でき、その場合、検証データセットのすべてのデータを使用してテストが実行されます。
- --workspace: 出力ファイルを格納するディレクトリ。
- --exclude: プルーニングから除外されるたたみ込みノード。
- --output_nodes: 推論グラフの出力ノード。

プルーニング ループを開始する

`ana` コマンドが完了後、モデルのプルーニングを開始できます。`prune` コマンドは `ana` コマンドとよく似ており、同じコンフィギュレーション ファイルを使用します。

```
vai_p_tensorflow \
  --action=prune \
  --input_graph=inference_graph.pbtxt \
  --input_ckpt=model.ckpt \
  --output_graph=sparse_graph.pbtxt \
```

```
  --output_ckpt=sparse.ckpt \
  --workspace=/home/deephi/tf_models/research/slim \
  --sparsity=0.1 \
  --exclude="conv node names that excluded from pruning" \
  --output_nodes="output node names of the network"
```

このコマンドでは、次の引数も使用します。

- --sparsity: プルーニング後のネットワークのスパース度。0 ～ 1 の値です。値が大きいほど、プルーニング後のモデルはよりスパースになります。

prune コマンドが完了すると、`vai_p_tensorflow` はプルーニング前後のネットワークの FLOP 数を出力します。

プルーニング済みモデルを微調整する

プルーニングを実行すると、モデルの精度はいくぶん低下します。この精度を改善するには、モデルを微調整する必要があります。プルーニング済みモデルの再調整は、初期学習率や学習率の減衰型などのハイパー パラメーターが異なる点を除けば、モデルを新規に学習させるのと基本的には同じです。

プルーニングと微調整が完了すると、プルーニングの 1 回の反復が完了したことになります。一般に、精度を大幅に低下させることなくプルーニング率を向上させるには、モデルを数回プルーニングする必要があります。1 回の「プルーニング/微調整」の反復が終わるたびに、コマンドに次の 2 つの変更を加えてから次のプルーニングを実行します。

1. `--input_ckpt` フラグを、直前の微調整プロセスで生成したチェックポイント ファイルに変更する。
2. `--sparsity` フラグの値を大きくして、次の反復のプルーニング量を増やす。

密なチェックポイントを生成する

プルーニングを数回反復すると、元のモデルよりサイズの小さいモデルが生成されます。最終的なモデルを生成するには、モデルの変換を実行します。

```
vai_p_tensorflow \
  --action=transform \
  --input_ckpt=model.ckpt-10000 \
  --output_ckpt=dense.ckpt
```

変換は、プルーニングの反復がすべて終了してから実行します。プルーニングの反復ごとに transform コマンドを実行しないでください。

これで、プルーニング済みモデルのアーキテクチャを格納した `GraphDef` ファイルと、学習済みの重みを保存したチェックポイント ファイルが生成されます。予測または量子化を実行するには、これら 2 つのファイルを 1 つの pb ファイルに結合します。フリーズの詳細は、保存したフォーマットの使用 を参照してください。

グラフをフリーズする

グラフをフリーズするには、次のコマンドを実行します。

```
freeze_graph \
    --input_graph=sparse_graph.pbtxt \
    --input_checkpoint=dense.ckpt \
    --input_binary=false \
    --output_graph=frozen.pb \
    --output_node_names="vgg_16/fc8/squeezed"
```

上記のステップがすべて完了すると、プルーニングの最終出力ファイル `frozen.pb` が生成されます。このファイルは、予測と量子化に使用できます。フリーズ済みグラフの FLOP 数を確認するには、次のコマンドを実行します。

```
vai_p_tensorflow --action=flops --input_graph=frozen.pb --input_nodes=input
--input_node_shapes=1,224,224,3 --output_nodes=vgg_16/fc8/squeezed
```

vai_p_tensorflow の使用法

vai_p_tensorflow の実行には、次の引数を使用できます。

表 1: **vai_p_tensorflow の引数**

引数	タイプ	動作	デフォルト	説明
action	文字列	–	""	実行するアクション。有効なアクションは、「ana」、「prune」、「transform」、および「flops」です。
workspace	文字列	['ana', 'prune']	""	出力ファイルを格納するディレクトリ。
input_graph	文字列	['ana', 'prune', 'flops']	""	ネットワークのアーキテクチャを定義した GraphDef protobuf ファイルのパス。
input_ckpt	文字列	['ana', 'prune', 'transform']	""	チェックポイント ファイルのパス。チェックポイント用に作成されるファイル名の接頭辞となります。
eval_fn_path	文字列	['ana']	""	モデルの評価に使用される Python ファイルのパス。
target	文字列	['ana']	""	モデルの精度を示す出力ノード名。
max_num_batches	int	['ana']	なし	評価するバッチの最大数 (デフォルトではすべて使用)。
output_graph	文字列	['prune']	""	プルーニング済みネットワークを格納する GraphDef protobuf ファイルのパス。
output_ckpt	文字列	['prune', 'transform'']	""	重みを格納するチェックポイント ファイルのパス。
gpu	文字列	['ana']	""	使用する GPU のデバイス ID (カンマ区切り)。
sparsity	float	['prune']	なし	プルーニング後のネットワークの目標スパース度。
exclude	リピート	['ana', 'prune']	なし	プルーニングから除外されるたたみ込みノード。
input_nodes	リピート	['flops']	なし	推論グラフの入力ノード。
input_node_shapes	リピート	['flops']	なし	入力ノードの形状。
output_nodes	リピート	['ana', 'prune', 'flops']	なし	推論グラフの出力ノード。
channel_batch	int	['prune']	2	プルーニング後、出力チャネルの数はこの値の倍数値となります。

PyTorch バージョン - vai_p_pytorch

PyTorch のプルーニング ツールは、実行可能なプログラムではなく Python パッケージです。モデルのプルーニングには、プルーニング API を使用します。

ベースライン モデルを準備する

簡単にするために、ここでは torchvision の ResNet18 を使用します。

```
from torchvision.models.resnet import resnet18
model = resnet18(pretrained=True)
```

プルーナーを作成する

プルーナーを作成するには、プルーニングするモデルと、モデルの入力形状および入力 dtype を指定します。形状は入力画像のサイズであり、バッチ サイズを含みません。

```
from pytorch_nndct import Pruner
from pytorch_nndct import InputSpec

pruner = Pruner(model, InputSpec(shape=(3, 224, 224), dtype=torch.float32))
```

複数の入力があるモデルについては、`InputSpec` のリストを使用してプルーナーを初期化できます。

モデル解析を実行する

モデル解析を実行するには、モデルの評価に使用できる関数を定義する必要があります。この関数の最初の引数で、評価されるモデルを指定する必要があります。

```
def evaluate(val_loader, model, criterion):
  batch_time = AverageMeter('Time', ':6.3f')
  losses = AverageMeter('Loss', ':.4e')
  top1 = AverageMeter('Acc@1', ':6.2f')
  top5 = AverageMeter('Acc@5', ':6.2f')
  progress = ProgressMeter(
      len(val_loader), [batch_time, losses, top1, top5], prefix='Test: ')

  # switch to evaluate mode
  model.eval()

  with torch.no_grad():
    end = time.time()
    for i, (images, target) in enumerate(val_loader):
      model = model.cuda()
      images = images.cuda(non_blocking=True)
      target = target.cuda(non_blocking=True)

      # compute output
      output = model(images)
      loss = criterion(output, target)

      # measure accuracy and record loss
      acc1, acc5 = accuracy(output, target, topk=(1, 5))
```

```
      losses.update(loss.item(), images.size(0))
      top1.update(acc1[0], images.size(0))
      top5.update(acc5[0], images.size(0))

      # measure elapsed time
      batch_time.update(time.time() - end)
      end = time.time()

      if i % 50 == 0:
        progress.display(i)

    # TODO: this should also be done with the ProgressMeter
    print(' * Acc@1 {top1.avg:.3f} Acc@5 {top5.avg:.3f}'.format(
        top1=top1, top5=top5))

  return top1.avg, top5.avg

def ana_eval_fn(model, val_loader, loss_fn):
  return evaluate(val_loader, model, loss_fn)[1]
```

次に、上で定義した関数を最初の引数として指定して `ana()` メソッドを呼び出します。

```
pruner.ana(ana_eval_fn, args=(val_loader, criterion))
```

ここで、'`args`' は、'`ana_eval_fn`' が要求する 2 番目の引数から始まる引数のタプルです。

モデルをプルーニングする

`prune()` メソッドを呼び出して、プルーニング済みモデルを生成します。ratio は、想定される FLOPs の削減率です。

```
model = pruner.prune(ratio=0.1)
```

プルーニング済みモデルを微調整する

微調整のプロセスは、ベースライン モデルの学習と同じです。相違点は、ベースライン モデルの重みが無作為に初期化されるのに対して、プルーニング済みモデルの重みはベースライン モデルから継承されることです。

```
class AverageMeter(object):
  """Computes and stores the average and current value"""

  def __init__(self, name, fmt=':f'):
    self.name = name
    self.fmt = fmt
    self.reset()

  def reset(self):
    self.val = 0
    self.avg = 0
    self.sum = 0
    self.count = 0

  def update(self, val, n=1):
    self.val = val
    self.sum += val * n
    self.count += n
    self.avg = self.sum / self.count
```

```
  def __str__(self):
    fmtstr = '{name} {val' + self.fmt + '} ({avg' + self.fmt + '})'
    return fmtstr.format(**self.__dict__)

def train(train_loader, model, criterion, optimizer, epoch):
  batch_time = AverageMeter('Time', ':6.3f')
  data_time = AverageMeter('Data', ':6.3f')
  losses = AverageMeter('Loss', ':.4e')
  top1 = AverageMeter('Acc@1', ':6.2f')
  top5 = AverageMeter('Acc@5', ':6.2f')

  # switch to train mode
  model.train()

  end = time.time()
  for i, (images, target) in enumerate(train_loader):
    # measure data loading time
    data_time.update(time.time() - end)

    model = model.cuda()
    images = images.cuda()
    target = target.cuda()

    # compute output
    output = model(images)
    loss = criterion(output, target)

    # measure accuracy and record loss
    acc1, acc5 = accuracy(output, target, topk=(1, 5))
    losses.update(loss.item(), images.size(0))
    top1.update(acc1[0], images.size(0))
    top5.update(acc5[0], images.size(0))

    # compute gradient and do SGD step
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # measure elapsed time
    batch_time.update(time.time() - end)
    end = time.time()

    if i % 10 == 0:
         print('Epoch: [{}] Acc@1 {} Acc@5 {}'.format(epoch, top1.avg,
top5.avg)
```

次に、学習ループを実行します。`train()` 関数のパラメーター「model」は、`prune()` メソッドから返されるオブジェクトです。

```
lr = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr, weight_decay=1e-4)

best_acc5 = 0
epochs = 10
for epoch in range(epochs):
  train(train_loader, model, criterion, optimizer, epoch)

  acc1, acc5 = evaluate(val_loader, model, criterion)

  # remember best acc@1 and save checkpoint
  is_best = acc5 > best_acc5
```

```
  best_acc5 = max(acc5, best_acc5)

  if is_best:
    model.save('resnet18_sparse.pth.tar')
    torch.save(model.state_dict(), 'resnet18_final.pth.tar')
```

注記: コードの最後の 2 行で、2 つのチェックポイント ファイルを保存しています。「model.save()」は、ベースライン モデルと同じ形状を持つスパース型の重みを保存します。削除されるチャネルは 0 に設定されます。「model.state_dict()」は、プルーニング済みの形状を持つデンス型の重みを返します。最初のチェックポイントは次のプルーニングへの入力として使用され、2 番目のチェックポイントは最終的な運用に使用されます。つまり、次に控えるプルーニングには最初のチェックポイントを使用し、最後のプルーニングであれば 2 番目のチェックポイントを使用します。

反復プルーニング

スパース チェックポイントをロードし、プルーニング率を高くします。ここでは、プルーニング率を 0.1 から 0.2 に上げています。

```
model = resnet18()
model.load_state_dict(torch.load('resnet18_sparse.pth.tar'))

pruner = Pruner(model, InputSpec(shape=(3, 224, 224), dtype=torch.float32))
model = pruner.prune(ratio=0.2)
```

新しいプルーニング済みモデルが生成されたら、再び微調整を開始できます。

vai_p_pytorch API

pytorch_nndct.InputSpec

モジュールへの各入力の dtype と形状を指定します。

引数

```
InputSpec(shape, dtype)
```

- shape: 形状タプル、想定される入力の形状。
- dtype: 想定される入力の torch.dtype。

pytorch_nndct.Pruner

チャネル プルーニングをモジュール レベルでインプリメントします。

引数

```
Pruner(module, input_specs)
```

新しいプルーナー オブジェクトを作成します。

- module: プルーニングする `torch.nn.Module` オブジェクト。
- input_specs: モジュールの入力 (`InputSpec` オブジェクトまたは `InputSpec` のリスト)。

方法

- `ana(eval_fn, args=(), gpus=None)`

  モデル解析を実行します。

  - eval_fn: 最初の引数に `torch.nn.Module` オブジェクトを取り、評価スコアを返す呼び出し可能オブジェクト。
  - args: `eval_fn` に渡される引数のタプル。
  - gpus: モデル解析に使用する GPU インデックスのタプルまたはリスト。設定しない場合、デフォルトの GPU が使用されます。

- `prune(ratio=None, threshold=None, excludes=[], output_script='graph.py')`

  指定された比またはしきい値により、ネットワークをプルーニングします。追加のプルーニング情報を指定した通常の `torch.nn.Module` と同じように機能する `PruningModule` オブジェクトを返します。

  - ratio: 想定される FLOPs 削減率。この値は単なるヒントです。プルーニング後の実際の FLOPs の減少は、必ずしもこの値と一致しません。
  - threshold: 許容できるモデルの精度低下の相対的比率。
  - excludes: プルーニングから除外する必要があるモジュール。
  - output_script: モデルの再構築に使用される生成済みスクリプトを保存するファイルパス。

- `summary(pruned_model)`

  プルーニング済みモデルのプルーニング サマリを生成します。

  - pruned_model: `prune()` メソッドで返されるプルーニング済みモジュール。

pytorch_nndct.pruning.core.PruningModule

`pytorch_nndct.Pruner.prune()` によって返されるプルーニング済みモジュールを表します。

属性

- module: 実際のプルーニング済みモジュールを表す `torch.nn.Module`。
- pruning_info: 各レイヤーのプルーニングの詳細を含むディクショナリ。

方法

```
save(path)
```

指定されたパスにスパース ステートを保存します。

- path: 保存するチェックポイント パス。

```
state_dict(destination=None, prefix='', keep_vars=False)
```

モジュールのステート全体を含むディクショナリを返します。関連する Pytorch の資料 を参照してください。

```
padded_state_dict()
```

モジュールのスパース ステートを含むディクショナリを返します。ステートの形状は元のベースライン モデルと同じで、プルーニング済みチャネルはゼロで充填されます。

Caffe バージョン - vai_p_caffe

コンフィギュレーション ファイルの作成

`vai_p_caffe` のほとんどのタスクには、入力引数としてコンフィギュレーション ファイルが必要です。一般的なコンフィギュレーション ファイルを次に示します。

```
workspace: "examples/decent_p/"
gpu: "0,1,2,3"
test_iter: 100
acc_name: "top-1"

model: "examples/decent_p/float.prototxt"
weights: "examples/decent_p/float.caffemodel"
solver: "examples/decent_p/solver.prototxt"

rate: 0.1

pruner {
  method: REGULAR
}
```

使用されている用語の定義は次のとおりです。

- workspace: テンポラリ ファイルおよび出力ファイルを格納するディレクトリ。
- gpu: アクセラレーションに使用する GPU のデバイス ID (カンマ区切り)。
- test_iter: テスト時に実行される反復数。この値を大きくすると、解析結果は向上しますが、実行時間は長くなります。このパラメーターは、検証データセット/batch_size のサイズを最大値として設定でき、その場合、検証データセットのすべてのデータを使用してテストが実行されます。
- acc_name: モデルの精度を判断するために使用する精度のレベル。
- model: モデル定義のプロトコル バッファー テキスト ファイル。学習用とテスト用に 2 つの異なるモデル定義ファイルがある場合、1 つのファイルに結合します。
- weights: プルーニングするモデルの重み。
- solver: 微調整に使用されるソルバー定義のプロトコル バッファー テキスト ファイル。
- rate: 重みのリダクション パラメーター。ベースライン モデルから演算数をどれだけ削減するかを設定します。たとえば「0.1」に設定すると、積和演算の数がベースライン モデルから 10% 削減されます。
- method: 使用されるプルーニング手法。現時点では、REGULAR のみサポートされます。

モデル解析を実行する

プルーニング プロセスの最初のステージです。これによって、適切なプルーニング手法を判断します。前のセクションの説明に従って、`config.prototxt` という名前の適切なコンフィギュレーション ファイルを作成し、次のコマンドを実行します。

```
$ ./vai_p_caffe ana -config config.prototxt
```

図 5: **モデル解析**

```
I1111 17:34:26.848263 14902 sens_analyser.cpp:208] Analysing layer [loss2/conv] done
I1111 17:34:26.848309 14902 sens_analyser.cpp:209] Analysis completed 80%
I1111 17:34:50.542733 14902 sens_analyser.cpp:208] Analysing layer [inception_5a/1x1] done
I1111 17:34:50.542935 14902 sens_analyser.cpp:209] Analysis completed 81%
I1111 17:35:13.842296 14902 sens_analyser.cpp:208] Analysing layer [inception_5a/3x3_reduce] done
I1111 17:35:13.842341 14902 sens_analyser.cpp:209] Analysis completed 83%
I1111 17:35:37.327677 14902 sens_analyser.cpp:208] Analysing layer [inception_5a/3x3] done
I1111 17:35:37.327931 14902 sens_analyser.cpp:209] Analysis completed 84%
I1111 17:36:00.633837 14902 sens_analyser.cpp:208] Analysing layer [inception_5a/double3x3_reduce] done
I1111 17:36:00.633883 14902 sens_analyser.cpp:209] Analysis completed 85%
I1111 17:36:24.055577 14902 sens_analyser.cpp:208] Analysing layer [inception_5a/double3x3a] done
I1111 17:36:24.055755 14902 sens_analyser.cpp:209] Analysis completed 87%
I1111 17:36:47.395057 14902 sens_analyser.cpp:208] Analysing layer [inception_5a/double3x3b] done
I1111 17:36:47.395103 14902 sens_analyser.cpp:209] Analysis completed 88%
I1111 17:37:10.866914 14902 sens_analyser.cpp:208] Analysing layer [inception_5a/pool_proj] done
I1111 17:37:10.867131 14902 sens_analyser.cpp:209] Analysis completed 90%
I1111 17:37:34.248847 14902 sens_analyser.cpp:208] Analysing layer [inception_5b/1x1] done
I1111 17:37:34.248894 14902 sens_analyser.cpp:209] Analysis completed 91%
I1111 17:37:57.655731 14902 sens_analyser.cpp:208] Analysing layer [inception_5b/3x3_reduce] done
I1111 17:37:57.655961 14902 sens_analyser.cpp:209] Analysis completed 92%
I1111 17:38:21.193574 14902 sens_analyser.cpp:208] Analysing layer [inception_5b/3x3] done
I1111 17:38:21.193621 14902 sens_analyser.cpp:209] Analysis completed 94%
I1111 17:38:44.628257 14902 sens_analyser.cpp:208] Analysing layer [inception_5b/double3x3_reduce] done
I1111 17:38:44.628468 14902 sens_analyser.cpp:209] Analysis completed 95%
I1111 17:39:08.187605 14902 sens_analyser.cpp:208] Analysing layer [inception_5b/double3x3a] done
I1111 17:39:08.187651 14902 sens_analyser.cpp:209] Analysis completed 97%
I1111 17:39:31.577209 14902 sens_analyser.cpp:208] Analysing layer [inception_5b/double3x3b] done
I1111 17:39:31.577422 14902 sens_analyser.cpp:209] Analysis completed 98%
I1111 17:39:55.200893 14902 sens_analyser.cpp:208] Analysing layer [inception_5b/pool_proj] done
I1111 17:39:55.200937 14902 sens_analyser.cpp:209] Analysis completed 100%
I1111 17:39:55.201654 14902 deephi_compress.cpp:195] Analysis done.
```

プルーニング ループを開始する

解析タスクが完了後、プルーニングを開始できます。prune コマンドは同じコンフィギュレーション ファイルを使用します。

```
$ ./vai_p_caffe prune -config config.prototxt
```

vai_p_caffe は、コンフィギュレーション ファイルで指定された rate パラメーターを使用してモデルのプルーニングを実行します。完了後、プルーニング前後の精度、重みの数、および必要な演算数を含むレポートが生成されます。次の図に、レポートの例を示します。

図 6: **プルーニング レポート**

```
I1113 16:13:49.488728 25830 pruning_runner.cpp:281] pruning done, output model: examples/deephi_compress/VGG16/regular_th_0.045/pruned.caffemodel
I1113 16:13:49.488790 25830 pruning_runner.cpp:295] summary of REGULAR compression with threshold0.045:
+-------------------------------------------------------+
| Item          | Before        | After         | Delta |
+-------------------------------------------------------+
| Accuracy      | 0.707850754   | 0.542000473   | -0.165850282  |
+-------------------------------------------------------+
| Weights       | 14714688      | 11016328      | -25.13380858% |
+-------------------------------------------------------+
| Operations    | 30706888832   | 20161270696   | -34.34267048% |
+-------------------------------------------------------+
To fine-tune the compressed model, please run:
deephi_compress finetune -config examples/deephi_compress/VGG16/config.prototxt
```

プルーニング済みネットワークを記述する final.prototxt という名前のファイルがワークスペースに 生成されます。

プルーニング済みモデルを微調整する

次のコマンドを実行して、プルーニングによって低下した精度を回復します。

```
$ ./vai_p_caffe finetune -config config.prototxt
```

プルーニング後のモデルを微調整するプロセスは、モデルを最初に学習させる場合と本質的には同じです。ただし、初期学習率、学習率の減衰型などのソルバー パラメーターは異なります。プルーニングの反復では、プルーニング タスクと微調整タスクが繰り返し実行されます。一般に、精度を大きく低下させることなくより多くの重みを削減するには、プルーニングの反復を複数回実行する必要があります。

プルーニングの反復を実行するたびに、コンフィギュレーション ファイルを次のように変更する必要があります。

1. ベースライン モデルを基準として rate パラメーターを増加する。
2. weights パラメーターを、前の微調整プロセスで取得した最適なモデルに変更する。

変更後のコンフィギュレーション ファイルは次のとおりです。

```
workspace: "examples/decent_p/"
gpu: "0,1,2,3"
test_iter: 100
acc_name: "top-1"

model: "examples/decent_p/float.prototxt"

#weights: "examples/decent_p/float.caffemodel"
weights: "examples/decent_p/regular_rate_0.1/_iter_10000.caffemodel"

solver: "examples/decent_p/solver.prototxt"

# change rate from 0.1 to 0.2
#rate: 0.1
rate: 0.2
pruner {
  method: REGULAR
}
```

最終モデルを生成する

プルーニングを数回反復すると、重みが削減されたモデルが生成されます。モデルを完成させるには、次の変換手順が必要です。

```
$ ./vai_p_caffe transform -model float.prototxt -weights
finetuned_model.caffemodel
```

出力ファイルの名前を指定しなければ、transformed.caffemodel という名前のデフォルト ファイルが生成されます。対応するモデル ファイルは、prune コマンドによって生成される final.prototxt です。

モデルの FLOP 数を確認するには、stat コマンドを使用します。

```
$ ./vai_p_caffe stat -model final.prototxt
```

重要: 変換は、プルーニングの反復がすべて完了した後にのみ実行する必要があります。

vai_p_caffe の使用法

vai_p_caffe の実行には、次の引数を使用できます。

表 2: **vai_p_caffe の引数**

引数	属性	デフォルト	説明
ana			
config	必須	“”	コンフィギュレーション ファイルのパス。
prune			
config	必須	“”	コンフィギュレーション ファイルのパス。
finetune			
config	必須	“”	コンフィギュレーション ファイルのパス。
transform			
model	必須	“”	ベースライン モデル定義のプロトコル バッファー テキスト ファイル。
weights	必須	“”	モデルの重みファイルのパス。
output	オプション	“”	変換された重みの出力。

表 3: **vai_p_caffe のコンフィギュレーション ファイル パラメーター**

引数	タイプ	属性	デフォルト	説明
workspace	文字列	必須	なし	出力ファイルを格納するディレクトリ。
gpu	文字列	オプション	“0”	圧縮と微調整に使用する GPU のデバイス ID (カンマ区切り)。
test_iter	int	オプション	100	テスト時に実行される反復数。
acc_name	文字列	必須	なし	対象となる精度レベル。このパラメーターは、ネットワーク精度の評価に使用されるレイヤー (layer_top) です。ネットワークに複数の評価指標がある場合、最も重要なものを一つ選択してください。分類タスクの場合、このパラメーターは top-1 または top-5 の精度となります。検出タスクの場合、このパラメーターは通常 mAP です。セグメンテーション タスクの場合、通常 mIOU を計算するレイヤーがこのパラメーターで設定されます。
model	文字列	必須	なし	モデル定義のプロトコル バッファー テキスト ファイル。学習用とテスト用に 2 つの異なるモデル定義ファイルがある場合、1 つのファイルに結合することを推奨します。
weights	文字列	必須	なし	圧縮の対象となる学習済みの重み。
solver	文字列	必須	なし	ソルバー定義のプロトコル バッファー テキスト ファイル。
rate	float	オプション	なし	求められるモデル プルーニング率。
method	列挙	オプション	REGULAR	使用されるプルーニング手法。現時点では、REGULAR のみサポートされます。
ssd_ap_version	文字列	オプション	なし	SSD ネットワーク圧縮用の ap_version の設定。11point、MaxIntegral、Integral のいずれかとなります。
exclude	リピート	オプション	なし	一部のレイヤーをプルーニングから除外するために使用。このパラメーターを使用することで、指定したたたみ込み層がプルーニングされないようにします。

表 3: **vai_p_caffe のコンフィギュレーション ファイル パラメーター** (続き)

引数	タイプ	属性	デフォルト	説明
kernel_batch	int	オプション	2	プルーニング後、出力チャネルの数はこの値の倍数値となります。

Darknet バージョン - vai_p_darknet

コンフィギュレーション ファイルの作成

次に、YOLOv3 のプルーニングに使用するメイン cfg ファイルの例を示します。この例では、VOC データセットを使用して学習した YoloV3 モデルをプルーニングし、VOC データを Darknet の標準的な方法で作成します。詳細は、YOLO ウェブサイト を参照してください。

YoloV3 ネットワークの構造上、yolo 層の前のたたみ込み層はプルーニングできません。つまり、標準的な YoloV3 cfg ファイルを使用する場合、レイヤー 81、93、および 105 を「ignore_layer」に追加する必要があります。メイン cfg のオプション一覧は、vai_p_darknet の使用法 を参照してください。

推奨: レイヤー 81、93、および 105 の前のたたみ込み層もプルーニングしないでください。レイヤー 80、92、104 を除外しないと、ana コマンドの実行時間が非常に長くなります。

```
# a cfg example to prune YoloV3
[pruning]
workspace=pruning
datacfg=pruning/voc.data
modelcfg=pruning/yolov3-voc.cfg
prunedcfg=pruning/yolov3-voc-prune.cfg
ana_out_file=pruning/ana.out
prune_out_weights=pruning/weights.prune
criteria=0
kernel_batch=2
ignore_layer=80,81,92,93,104,105
yolov3=1
threshold=0.005
```

トレーニング用データセットの準備

満足のいくプルーニング結果を得るには、元のトレーニング用データセットを使用した微調整が必要です。darknet 用のデータセットの準備については、Yolo ページ を参照してください。Pascal VOC データセットを例とすると、次のようにデータ cfg ファイル `"voc.data"` を見つけるか、作成します。

```
classes= 20
train  = /dataset/voc/train.txt
valid  = /dataset/voc/2007_test.txt
names = data/voc.names
backup = backup
```

「train」テキスト ファイルがトレーニング イメージを指定します。

```
/dataset/voc/VOCdevkit/VOC2007/JPEGImages/000012.jpg
/dataset/voc/VOCdevkit/VOC2007/JPEGImages/000017.jpg
/dataset/voc/VOCdevkit/VOC2007/JPEGImages/000023.jpg
...
```

同時に、ラベル ファイルが対応する「labels」フォルダーに置かれます。ディレクトリ階層は次のようになります。

```
/dataset/voc/VOCdevkit/VOC2007/
    |-- JPEGImages/
    |     |-- 000001.jpg
    |     |-- 000002.jpg
    |     |-- ...
    |-- labels
    |     |-- 000001.txt
    |     |-- 000002.txt
    |     |-- ...
    |-- ...
```

モデル解析を実行する

モデルのプルーニングを実行する前に、モデルを解析する必要があります。このプロセスの主な目的は、後でモデルをプルーニングする際の最適なプルーニング手法を見つけることです。解析を開始するには、次のコマンドを実行します。

```
./vai_p_darknet pruner ana pruning/cfg pruning/yolov3-voc_final.weights
```

モデルおよび検証データセットのサイズによっては、`ana` コマンドの実行に数時間かかることもあります。複数の GPU を使用すると、ana コマンドの実行時間を大幅に短縮できます。次の例は、4 つの GPU を使用してコマンドを実行しています。

```
./vai_p_darknet pruner ana pruning/cfg pruning/yolov3-voc_final.weights -
gpus 0,1,2,3
```

プルーニング ループを開始する

`ana` コマンドが正常に実行されたら、モデルのプルーニングを開始できます。`prune` は `ana` とよく似ており、同じコンフィギュレーション ファイルを使用します。

```
./vai_p_darknet pruner prune pruning/cfg pruning/yolov3-voc_final.weights
```

プルーニング ツールは、threshold の値に従ってモデルをプルーニングし、次のような出力を生成します。モデルをロード後、レイヤーごとのプルーニング率と出力重みファイルが表示されます。最後に、プルーニング済みモデルの mAP が自動的に評価されます。この例では、プルーニング後の mAP は元の値から大きく低下し、0.494531 となっています。このため、次のステップで微調整を実行して精度を回復します。ステップ サイズを小さくしてプルーニングの反復を継続する場合は、メイン cfg の threshold オプションを小さい値に変更して再度実行してください。

```
Start pruning ...
pruning slot: 0,
prune main layer 0
  kernel_batch:2, rate:0.3000, keep_num:24
pruning slot: 1,3,
prune main layer 1
  kernel_batch:2, rate:0.3000, keep_num:46
```

```
  prune slave layer 3
  prune related layer 2
  kernel_batch:2, rate:0.3000, keep_num:24
...
pruning slot: 102,
prune main layer 102
  kernel_batch:2, rate:0.5000, keep_num:128
  prune related layer 101
  kernel_batch:2, rate:0.5000, keep_num:64
Saving weights to pruning/weights.prune
calculate map
Process   100 on GPU 0
Process   200 on GPU 0
...
Process  4800 on GPU 0
Process  4900 on GPU 0
AP for class 0 = 0.501017
AP for class 1 = 0.711958
...
AP for class 18 = 0.621339
AP for class 19 = 0.472648
mAP : 0.494531
Total Detection Time: 158.943884 Seconds
```

プルーニング済みモデルを微調整する

プルーニングを実行すると、モデルの精度はいくぶん低下します。この精度を改善するには、モデルを微調整する必要があります。

微調整を開始するには、次のコマンドを実行します。

```
./vai_p_darknet pruner finetune pruning/cfg
```

一般に、複数の GPU を使用した方が微調整の速度が向上します。

```
./vai_p_darknet pruner finetune pruning/cfg -gpus 0,1,2,3
```

このコマンドを実行すると、まず基本情報が出力されます。必要に応じて、`pruning/yolov3-voc-prune.cfg` 内の学習パラメーターを変更します。

```
$./darknet pruner finetune pruning/cfg -gpus 0,1,2,3
GPUs: 0,1,2,3
Workspace exists: pruning
Finetune model  : pruning/yolov3-voc-prune.cfg
Finetune weights: pruning/weights.prune
...
```

プルーニングと微調整が完了すると、プルーニングの 1 回の反復が完了したことになります。一般に、精度を大幅に低下させることなくプルーニング率を向上させるには、数回の反復が必要です。一般的なワークフローは、次のとおりです。

1. コンフィギュレーション ファイルの threshold に小さい値を設定する。
2. モデルのプルーニングを開始する。
3. プルーニング済みモデルを微調整する。
4. threshold の値を大きくする。

5. 手順 2 に戻る。

プルーニングの反復が 1 回終わるたびに、次の 2 つの変更を加えてから次の反復を実行します。まず、メイン cfg ファイルの threshold の値を大きくします。次に、出力ファイルが新しい出力ファイルで上書きされないように、ファイル名を変更します。

次に、メイン cfg ファイルの変更例を示します。

```
[pruning]
workspace=pruning
datacfg=pruning/voc.data
modelcfg=pruning/yolov3-voc.cfg
prunedcfg=pruning/yolov3-voc-prune.cfg
ana_out_file=pruning/ana.out
# change prune_out_weights to avoid overwriting old results
prune_out_weights=pruning/weights.prune.round2
criteria=0
kernel_batch=2
ignore_layer=80,81,92,93,104,105
yolov3=1
# change threshold from 0.005 to 0.01
threshold=0.01
```

変更後のプルーニング メイン cfg ファイルを使用して prune コマンドを再度実行します。これで、次の反復を開始できます。

モデルの変換

プルーニングの反復を数回繰り返すと、元のモデルよりもはるかに小さいモデルが得られます。このプルーニング済みモデルにはゼロ チャネルが多く含まれるため、最後にこれを通常のモデルに変換する必要があります。

モデルを変換するには、次のコマンドを実行します。

```
./vai_p_darknet pruner transform pruning/cfg backup/*_final.weights
```

注記: 変換は、プルーニングの反復がすべて完了してから実行します。プルーニングの反復ごとに transform コマンドを実行する必要はありません。

圧縮率やプルーニング済みモデルに含まれる演算の数は、`stat` コマンドを使用するとわかります。

```
./vai_p_darknet pruner stat model-transform.cfg
```

次に、このコマンドの出力例を示します。

```
...
layer 104, ops:1595576320
layer 104, params:295168
layer 105, ops:104036400
layer 105, params:19275
Total operations: 41495485135
Total params: 43693647
```

モデルの評価

変換済みモデルは、標準的な方法を使用してテストできます。メイン cfg ファイルの modelcfg を、変換済みモデルに変更します。

```
modelcfg=pruning/model-transform.cfg
```

検出出力を生成し、別のデータセットで提供されるツールを使用してさらに mAP を評価するには、次のコマンドを実行します。

```
./vai_p_darknet pruner valid pruning/cfg weights.transform
```

モデルの変換と運用

ザイリンクス ハードウェアは Caffe および TensorFlow モデルの運用しかサポートしていないため、Darknet モデルは Caffe モデルに変換する必要があります。Caffe には、オープン ソースの変換ツールが付属しています。Darknet モデルを Caffe モデルに変換後、『Vitis AI ユーザー資料』(UG1431) の『Vitis AI ユーザー ガイド』の手順に従って FPGA 上で運用してください。

vai_p_darknet の使用法

プルーニング ツールには、表 1: vai_p_tensorflow の引数 に示す 7 つのコマンドがあります。`ana`、`prune`、`finetune`、および `transform` はメイン コマンドで、プルーニング プロセスに対応しています。`stat`、`map`、および `valid` コマンドは補助ツールです。いずれのコマンドにも、メイン cfg ファイルが必要です。表 2: vai_p_caffe の引数 に、メイン cfg ファイルのオプション一覧を示します。表 2: vai_p_caffe の引数 の最後の列 (「使用コマンド」) は、そのオプションがどのコマンドで使用されるかを示しています。ここでは、各コマンドを最初の 1 文字で表しています。たとえば、「workspace」オプションはすべてのコマンドで使用されるため、「`APFTSMV`」と記載しています。「is_yolov3」は `ana` および `prune` コマンドでのみ使用されるため、「`AP_____`」と記載しています。

表 4: **vai_p_darknet の 7 つのコマンド**

コマンド	説明/使用方法

ana	モデルを解析し、適切なプルーニング手法を判断します。このコマンドは入力モデル (メイン cfg の「modelcfg」とコマンド ラインの weights) を解析し、prunedcfg ファイルを生成し、ana の結果を出力します。デフォルトでは cfg で指定した「workspace」の場所に出力され、デフォルトのファイル名はそれぞれ `{$prefix}-prune.cfg`、`ana.out` となります。出力ファイル名は、cfg で「prunedcfg」と「ana_out_file」を指定して変更できます。メイン cfg のオプション一覧は、表 2: vai_p_caffe の引数 を参照してください。GPU インデックスで複数の GPU を指定すると、ana コマンドの実行時間を短縮できます。
	
	`Usage:`
	`./vai_p_darknet pruner ana cfg weights [-gpus gpu_ids]`
	`Example:`
	`./vai_p_darknet pruner ana pruning/cfg pruning/yolov3.weights -gpus 0,1,2,3`

表 4: **vai_p_darknet の 7 つのコマンド** (続き)

コマンド	説明/使用方法

prune	入力モデルをプルーニングします。このコマンドは、ana コマンドの解析結果およびメイン cfg ファイルの設定に従って入力モデル (メイン cfg の「modelcfg」とコマンド ラインの weights) をプルーニングします。パラメーターには、「criteria」、「kernel_batch」、「ignore_layer」、「threshold」、および「is_yolov3」があります。設定の詳細は、表 2: vai_p_caffe の引数 を参照してください。
	通常、ana コマンドを実行すると prunedcfg ファイルが作成されます。メイン cfg で「prunedcfg」を指定していない場合、このコマンドを実行するとデフォルトのパスにこの名前のファイルが 1 つ自動で生成されます。出力されるプルーニング済み重みファイルは、メイン cfg ファイルの「prune_out_weights」で定義します。定義していない場合、デフォルトで「workspace」の場所に「weights.prune」が出力されます。
	`Usage:`
	`./vai_p_darknet pruner prune cfg weights`
	`Example:`
	`./vai_p_darknet pruner prune pruning/cfg pruning/yolov3.weights -gpus 0,1,2,3`

finetune	プルーニング済みモデルを微調整して、モデルの精度を改善します。このコマンドは、プルーニング済みモデルを微調整します。メイン cfg ファイルの「prunedcfg」で指定したモデル記述を読み出します。重みファイルに関しては、コマンド ラインで指定した重みファイルが優先されます。指定しない場合は、メイン cfg ファイルの「prune_out_weights」が使用されます。finetune コマンドは標準的な Darknet 学習プロセスに従い、モデル スナップショットをデフォルトで「backup」ディレクトリに保存します。
	`Usage:`
	`./vai_p_darknet pruner finetune cfg [weights] [-gpus gpu_ids]`
	`Example:`
	`./vai_p_darknet pruner finetune pruning/cfg pruning/weights.prune -gpus 0,1,2,3`

transform	プルーニング済みモデルを通常のモデルに変換します。チャネルをプルーニングすると、プルーニング済みモデル (メイン cfg の「prunedcfg」とコマンド ラインの weights) には多くのゼロが含まれます。transform コマンドは不要なゼロを除去し、プルーニング済みモデルを通常のモデルに変換します。出力モデル cfg とモデル重みは、メイン cfg の「transform_out_cfg」と「transform_out_weights」で指定します。指定しない場合は、「model-transform.cfg」と「weights.transform」がデフォルトのファイル名となります。
	`Usage:`
	`./vai_p_darknet pruner transform cfg weights`
	`Example:`
	`./vai_p_darknet pruner transform pruning/cfg backup/*_final.weights -gpus 0,1,2,3`

stat	モデルに必要な浮動小数点演算の数をカウントします。このコマンドに必要なのは、modelcfg のみです。ほかのコマンド同様、コマンド ラインで「modelcfg」の項目を含むメイン cfg を使用できます。または、コマンド ラインで modelcfg を直接使用することもできます。
	`Usage`
	`./vai_p_darknet pruner stat cfg`
	`Example:`
	`./vai_p_darknet pruner stat pruning/cfg`
	`./vai_p_darknet pruner stat pruning/yolov3.cfg`

map	ビルトインの方法を使用してモデル (メイン cfg の「modelcfg」 + コマンド ラインの weights) の mAP をテストします。この mAP はザイリンクス ツールで使用するもので、標準の方法では実行できません。検出結果を生成し、標準の Python スクリプトを使用して mAP を計算するには、valid コマンドを使用してください。
	`Usage`
	`./vai_p_darknet pruner map cfg weights`
	`Example:`
	`./vai_p_darknet pruner map pruning/cfg backup/*_final.weights -gpus 0,1,2,3`

表 4: **vai_p_darknet の 7 つのコマンド** (続き)

コマンド	説明/使用方法

valid	指定したモデルを使用して予測を実行し、標準の検出結果を出力します。さらにこの結果に基づいて、別のデータセットで提供されるツールを使用して mAP を評価できます。このコマンドは、オープン ソース Darknet の「darknet detector valid」と同じです。
	`Usage`
	`./vai_p_darknet pruner valid cfg weights [-out outfile]`
	`Example:`
	`./vai_p_darknet pruner valid pruning/cfg backup/*_final.weights -gpus 0,1,2,3`

表 5: **メイン cfg ファイルのオプション一覧**

オプション	説明	デフォルト値	使用コマンド
workspace	プルーニングのワークスペース	“./pruning”	APFTSMV
datacfg	データ cfg ファイル (標準の Darknet と同じ)	“./pruning/voc.data”	A_F__MV
modelcfg	モデル構造を記述したモデル cfg ファイル	""	AP__SMV
prunedcfg	プルーニング済みモデルのモデル cfg ファイル (元の modelcfg にいくつかのフラグを追加したもの)	""	APFT___
ana_out_file	ana 出力ファイル	"./pruning/ana.out"	AP_____
criteria	プルーニング時の並べ替え基準。0: L1 norm で並べ替え、1: L2 norm で並べ替え	0	AP_____
kernel_batch	一度にプルーニングする最小チャネル数	2	AP_____
ignore_layer	プルーニングから除外するレイヤーの ID (カンマ区切り)	""	AP_____
is_yolov3	YOLOv3 を示すフラグ	1	AP_____
eval_images	ビルトイン関数で mAP を評価する際に使用する画像の数。-1 の場合、検証データセットをすべて使用	-1	A____M_
threshold	プルーニングのしきい値。このしきい値の範囲内で、各レイヤーのプルーニング率を計算します。詳細は、例を参照してください。	0.005	_P_____
prune_out_file	プルーニング済み重みの出力ファイル。	""	_PF____
snapshot	微調整の際に、何回の反復後にモデルを保存するかを指定。	4000	__F____
transform_out_cfg	変換済みモデルのモデル cfg ファイル。	model-transform.cfg	___T___
transform_out_weights	変換済みモデルの重みファイル。	weights.transform	___T___

XILINX

第 4 章

ネットワークの例

TensorFlow の例

MNIST

MNIST データセット には、手書き数字画像の学習用サンプルが 60,000 セット、検証用サンプルが 10,000 セット用意されています。各サンプルは、28 x 28 ピクセルのモノクロ画像です。

ここでは、下位の API および tf.estimator.Estimator を使用して簡単なたたみ込みニューラル ネットワーク分類器を構築し、`vai_p_tensorflow` を使用してこれをプルーニングする方法を紹介します。

TensorFlow の下位 API

データセットをダウンロードして変換する

`data_utils.py` という名前のファイルを作成し、次のコードを追加します。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import gzip, os, sys
from six.moves import urllib

import numpy as np
import tensorflow as tf

os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

# The URLs where the MNIST data can be downloaded.
_DATA_URL = 'http://yann.lecun.com/exdb/mnist/'
_TRAIN_DATA_FILENAME = 'train-images-idx3-ubyte.gz'
_TRAIN_LABELS_FILENAME = 'train-labels-idx1-ubyte.gz'
_TEST_DATA_FILENAME = 't10k-images-idx3-ubyte.gz'
_TEST_LABELS_FILENAME = 't10k-labels-idx1-ubyte.gz'
_LABELS_FILENAME = 'labels.txt'
_DATASET_DIR = 'data/mnist'

_IMAGE_SIZE = 28
_NUM_CHANNELS = 1
_NUM_LABELS = 10

# The names of the classes.
_CLASS_NAMES = [
    'zero',
    'one',
```

```
    'two',
    'three',
    'four',
    'five',
    'size',
    'seven',
    'eight',
    'nine',
]

def _extract_images(filename, num_images):
  """Extract the images into a numpy array.

  Args:
    filename: The path to an MNIST images file.
    num_images: The number of images in the file.

  Returns:
    A numpy array of shape [number_of_images, height, width, channels].
  """
  print('Extracting images from: ', filename)
  with gzip.open(filename) as bytestream:
    bytestream.read(16)
    buf = bytestream.read(
        _IMAGE_SIZE * _IMAGE_SIZE * num_images * _NUM_CHANNELS)
    data = np.frombuffer(buf, dtype=np.uint8)
    data = data.reshape(num_images, _IMAGE_SIZE, _IMAGE_SIZE, _NUM_CHANNELS)
  return data


def _extract_labels(filename, num_labels):
  """Extract the labels into a vector of int64 label IDs.

  Args:
    filename: The path to an MNIST labels file.
    num_labels: The number of labels in the file.

  Returns:
    A numpy array of shape [number_of_labels]
  """
  print('Extracting labels from: ', filename)
  with gzip.open(filename) as bytestream:
    bytestream.read(8)
    buf = bytestream.read(1 * num_labels)
    labels = np.frombuffer(buf, dtype=np.uint8).astype(np.int64)
  return labels

def int64_feature(values):
  """Returns a TF-Feature of int64s.

  Args:
    values: A scalar or list of values.

  Returns:
    A TF-Feature.
  """
  if not isinstance(values, (tuple, list)):
    values = [values]
  return tf.train.Feature(int64_list=tf.train.Int64List(value=values))

def bytes_feature(values):
  """Returns a TF-Feature of bytes.
```

```
  Args:
    values: A string.

  Returns:
    A TF-Feature.
  """
  return tf.train.Feature(bytes_list=tf.train.BytesList(value=[values]))


def _image_to_tfexample(image_data, class_id):
  return tf.train.Example(features=tf.train.Features(feature={
      'image/encoded': bytes_feature(image_data),
      'image/class/label': int64_feature(class_id)
  }))

def _add_to_tfrecord(data_filename, labels_filename, num_images,
                     tfrecord_writer):
  """Loads data from the binary MNIST files and writes files to a TFRecord.

  Args:
    data_filename: The filename of the MNIST images.
    labels_filename: The filename of the MNIST labels.
    num_images: The number of images in the dataset.
    tfrecord_writer: The TFRecord writer to use for writing.
  """
  images = _extract_images(data_filename, num_images)
  labels = _extract_labels(labels_filename, num_images)

  shape = (_IMAGE_SIZE, _IMAGE_SIZE, _NUM_CHANNELS)
  with tf.Graph().as_default():
    image = tf.placeholder(dtype=tf.uint8, shape=shape)
    encoded_png = tf.image.encode_png(image)

    with tf.Session('') as sess:
      for j in range(num_images):
        sys.stdout.write('\r>> Converting image %d/%d' % (j + 1,
num_images))
        sys.stdout.flush()

        png_string = sess.run(encoded_png, feed_dict={image: images[j]})
        example = _image_to_tfexample(png_string, labels[j])
        tfrecord_writer.write(example.SerializeToString())


def _get_output_filename(dataset_dir, split_name):
  """Creates the output filename.

  Args:
    dataset_dir: The directory where the temporary files are stored.
    split_name: The name of the train/test split.

  Returns:
    An absolute file path.
  """
  return '%s/mnist_%s.tfrecord' % (dataset_dir, split_name)


def _download_dataset(dataset_dir):
  """Downloads MNIST locally.

  Args:
    dataset_dir: The directory where the temporary files are stored.
  """
```

```
  for filename in [_TRAIN_DATA_FILENAME,
                   _TRAIN_LABELS_FILENAME,
                   _TEST_DATA_FILENAME,
                   _TEST_LABELS_FILENAME]:
    filepath = os.path.join(dataset_dir, filename)

    if not os.path.exists(filepath):
      print('Downloading file %s...' % filename)
      def _progress(count, block_size, total_size):
        sys.stdout.write('\r>> Downloading %.1f%%' % (
            float(count * block_size) / float(total_size) * 100.0))
        sys.stdout.flush()
      filepath, _ = urllib.request.urlretrieve(_DATA_URL + filename,
                                               filepath,
                                               _progress)
      print()
      with tf.gfile.GFile(filepath) as f:
        size = f.size()
      print('Successfully downloaded', filename, size, 'bytes.')

def _write_label_file(labels_to_class_names, dataset_dir,
                      filename=_LABELS_FILENAME):
  """Writes a file with the list of class names.

  Args:
    labels_to_class_names: A map of (integer) labels to class names.
    dataset_dir: The directory in which the labels file should be written.
    filename: The filename where the class names are written.
  """
  labels_filename = os.path.join(dataset_dir, filename)
  with tf.gfile.Open(labels_filename, 'w') as f:
    for label in labels_to_class_names:
      class_name = labels_to_class_names[label]
      f.write('%d:%s\n' % (label, class_name))

def _clean_up_temporary_files(dataset_dir):
  """Removes temporary files used to create the dataset.

  Args:
    dataset_dir: The directory where the temporary files are stored.
  """
  for filename in [_TRAIN_DATA_FILENAME,
                   _TRAIN_LABELS_FILENAME,
                   _TEST_DATA_FILENAME,
                   _TEST_LABELS_FILENAME]:
    filepath = os.path.join(dataset_dir, filename)
    tf.gfile.Remove(filepath)


def download_and_convert(dataset_dir, clean=False):
  """Runs the download and conversion operation.

  Args:
    dataset_dir: The dataset directory where the dataset is stored.
  """
  if not tf.gfile.Exists(dataset_dir):
    tf.gfile.MakeDirs(dataset_dir)

  training_filename = _get_output_filename(dataset_dir, 'train')
  testing_filename = _get_output_filename(dataset_dir, 'test')

  if tf.gfile.Exists(training_filename) and
tf.gfile.Exists(testing_filename):
```

```
    print('Dataset files already exist. Exiting without re-creating them.')
    return

  _download_dataset(dataset_dir)

  # First, process the training data:
  with tf.python_io.TFRecordWriter(training_filename) as tfrecord_writer:
    data_filename = os.path.join(dataset_dir, _TRAIN_DATA_FILENAME)
    labels_filename = os.path.join(dataset_dir, _TRAIN_LABELS_FILENAME)
    _add_to_tfrecord(data_filename, labels_filename, 60000, tfrecord_writer)

  # Next, process the testing data:
  with tf.python_io.TFRecordWriter(testing_filename) as tfrecord_writer:
    data_filename = os.path.join(dataset_dir, _TEST_DATA_FILENAME)
    labels_filename = os.path.join(dataset_dir, _TEST_LABELS_FILENAME)
    _add_to_tfrecord(data_filename, labels_filename, 10000, tfrecord_writer)

  # Finally, write the labels file:
  labels_to_class_names = dict(zip(range(len(_CLASS_NAMES)), _CLASS_NAMES))
  _write_label_file(labels_to_class_names, dataset_dir)

  if clean:
      _clean_up_temporary_files(dataset_dir)
  print('\nFinished converting the MNIST dataset!')

def _parse_function(tfrecord_serialized):
  """Parse TFRecord serialized object into image and label with specified
shape
  and data type.

  Args:
    TFRecord_serialized: tf.data.TFRecordDataset.

  Returns:
    Parsed image and label
  """
  features = {'image/encoded': tf.FixedLenFeature([], tf.string),
              'image/class/label': tf.FixedLenFeature([], tf.int64)}
  parsed_features = tf.parse_single_example(tfrecord_serialized, features)
  image = parsed_features['image/encoded']
  label = parsed_features['image/class/label']
  image = tf.image.decode_png(image)
  image = tf.divide(image, 255)
  return image, label

def get_init_data(train_batch,
              test_batch,
              dataset_dir=_DATASET_DIR,
              test_only=False,
              num_parallel_calls=8):
  """Build input data pipline, which must be initial by sess.run(init)

  Args:
    train_batch: batch size of train data set
    test_batch: batch size of test data set
    dataset_dir: Optional. Where to store data set
    test_only: If only build test data input pipline set
    num_parallel_calls: number of parallel read data

  Returns:
    img: input image data tensor
    label: input label data tensor
    train_init: train data initializer
```

```
    test_init:test data initializer
  """
  with tf.name_scope('data'):
    testing_filename = _get_output_filename(dataset_dir, 'test')
    test_data = tf.data.TFRecordDataset(testing_filename)
    test_data = test_data.map(_parse_function, \
            num_parallel_calls=num_parallel_calls)
    test_data = test_data.batch(test_batch)
    test_data = test_data.prefetch(test_batch)

    iterator = tf.data.Iterator.from_structure(test_data.output_types,
                                               test_data.output_shapes)
    test_init = iterator.make_initializer(test_data)    # initializer for
train_data
    img, label = iterator.get_next()
    # reshape the image from [28,28,1], to make it work with tf.nn.conv2d
    img = tf.reshape(img, shape=[-1, _IMAGE_SIZE , _IMAGE_SIZE ,
_NUM_CHANNELS])
    label = tf.one_hot(label, _NUM_LABELS)

    train_init = None
    if not test_only:
      training_filename = _get_output_filename(dataset_dir, 'train')
      train_data = tf.data.TFRecordDataset([training_filename])
      train_data = train_data.shuffle(10000)
      train_data = train_data.map(_parse_function,\
              num_parallel_calls=num_parallel_calls)
      train_data = train_data.batch(train_batch)
      train_data = train_data.prefetch(train_batch)
      train_init = iterator.make_initializer(train_data)  # initializer for
train_data
    return img, label, train_init, test_init

def get_one_shot_test_data(
        test_batch,
        dataset_dir=_DATASET_DIR,
        num_parallel_calls=8):
  """Build input test data pipline, which no need to be initial. For
`vai_p_tensorflow
  --ana`

  Args:
    test_batch: batch size of test data set
    dataset_dir: Optional. Where to store data set
    num_parallel_calls: number of parallel read data

  Returns:
    img: input image data tensor
    label: input label data tensor
  """
  #do not need initial
  with tf.name_scope('data'):
    testing_filename = _get_output_filename(dataset_dir, 'test')
    test_data = tf.data.TFRecordDataset([testing_filename])
    test_data = test_data.map(_parse_function,
                              num_parallel_calls=num_parallel_calls)
    test_data = test_data.batch(test_batch)
    test_data = test_data.prefetch(test_batch)

    iterator = test_data.make_one_shot_iterator()
    img, label = iterator.get_next()
    # reshape the image from [28,28,1] to make it work with tf.nn.conv2d
    img = tf.reshape(img, shape=[-1, _IMAGE_SIZE , _IMAGE_SIZE ,
```

```
_NUM_CHANNELS])
    label = tf.one_hot(label, _NUM_LABELS)
    return img, label

if __name__ == '__main__':
  download_and_convert(_DATASET_DIR)
```

dataset_utils には、train_batch と test_batch を引数にとり、画像、ラベル テンソル、および学習データ用とテスト データ用それぞれの初期化演算を返す関数 get_init_data taking train_batch があり、これを学習および評価時に実行します。

data_utils.py をモジュールとしてインポートし、入力データ パイプラインとすることも、シェルで実行して MNIST データセットをダウンロードし、次のコマンドを使用して TFRecord フォーマットに変換することもできます。

```
$ python data_utils.py
```

これにより、次のファイルが生成されます。

```
data/minist/label.txt
data/minist/mnist_test.tfrecord data/minist/mnist_train.tfrecord
data/minist/t10k-images-idx3-ubyte.gz
data/minist/t10k-labels-idx1-ubyte.gz
data/minist/train-images-idx3-ubyte.gz
data/minist/train-labels-idx1-ubyte.gz
```

CNN MNIST 分類器を作成する

low_level_cnn.py という名前のファイルを作成し、次のコードを追加します。

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
import tensorflow as tf
from data_utils import get_one_shot_test_data

TEST_BATCH=100

def conv_relu(inputs, filters, k_size, stride, padding, scope_name):
    '''
    A method that does convolution + relu on inputs
    '''
    with tf.variable_scope(scope_name, reuse=tf.AUTO_REUSE) as scope:
        in_channels = inputs.shape[-1]
        kernel = tf.get_variable('kernel',
                                 [k_size, k_size, in_channels, filters],

initializer=tf.truncated_normal_initializer())
        biases = tf.get_variable('biases',
                                 [filters],
                                 initializer=tf.random_normal_initializer())
        conv = tf.nn.conv2d(inputs, kernel, strides=[1, stride, stride, 1],
padding=padding)
    return tf.nn.relu(tf.nn.bias_add(conv, biases), name=scope.name)

def maxpool(inputs, ksize, stride, padding='VALID', scope_name='pool'):
    '''A method that does max pooling on inputs'''
    with tf.variable_scope(scope_name, reuse=tf.AUTO_REUSE) as scope:
        pool = tf.nn.max_pool(inputs,
                              ksize=[1, ksize, ksize, 1],
                              strides=[1, stride, stride, 1],
```

```
                                padding=padding)
    return pool

def fully_connected(inputs, out_dim, scope_name='fc'):
    '''
    A fully connected linear layer on inputs
    '''
    with tf.variable_scope(scope_name, reuse=tf.AUTO_REUSE) as scope:
        in_dim = inputs.shape[-1]
        w = tf.get_variable('weights', [in_dim, out_dim],
                            initializer=tf.truncated_normal_initializer())
        b = tf.get_variable('b', [out_dim],
                            initializer=tf.constant_initializer(0.0))
        out = tf.matmul(inputs, w) + b
    return out

def net_fn(image, n_classes=10, keep_prob=0.5, is_training=True):
    conv1 = conv_relu(inputs=image,
                      filters=32,
                      k_size=5,
                      stride=1,
                      padding='SAME',
                      scope_name='conv1')
    pool1 = maxpool(conv1, 2, 2, 'VALID', 'pool1')
    conv2 = conv_relu(inputs=pool1,
                      filters=64,
                      k_size=5,
                      stride=1,
                      padding='SAME',
                      scope_name='conv2')
    pool2 = maxpool(conv2, 2, 2, 'VALID', 'pool2')
    feature_dim = pool2.shape[1] * pool2.shape[2] * pool2.shape[3]
    pool2 = tf.reshape(pool2, [-1, feature_dim])
    fc = fully_connected(pool2, 1024, 'fc')
    keep_prob = keep_prob if is_training else 1
    dropout = tf.nn.dropout(tf.nn.relu(fc), keep_prob, name='relu_dropout')
    logits = fully_connected(dropout, n_classes, 'logits')
    return logits
net_fn.default_image_size=28

def model_fn():
  tf.logging.set_verbosity(tf.logging.INFO)
  img, labels = get_one_shot_test_data(TEST_BATCH)

  logits = net_fn(img, is_training=False)
  predictions = tf.argmax(logits, 1)
  labels = tf.argmax(labels, 1)
  eval_metric_ops = {
      'accuracy': tf.metrics.accuracy(labels, predictions),
      'recall_5': tf.metrics.recall_at_k(labels, logits, 5)
  }
  return eval_metric_ops
```

`net_fn` 関数は、ネットワーク アーキテクチャを定義します。この関数は MNIST 画像データを引数にとり、logits テンソルを返します。関数 `model_fn` は入力データ パイプラインを読み出し、評価指標演算のディクショナリを返します。

モデルの作成、学習、評価

`train_eval_utils.py` という名前のファイルを作成し、次のコードを追加します。

```
import os, time, sys
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

import tensorflow as tf

from low_level_cnn import net_fn
from data_utils import get_init_data

class ConvNet(object):
  def __init__(self, training=True):
    self.lr = 0.001
    self.train_batch = 128
    self.test_batch = 100
    self.keep_prob = tf.constant(0.75)
    self.gstep = tf.Variable(0, dtype=tf.int64, trainable=False,
name='global_step')
    self.n_classes = 10
    self.skip_step = 100
    self.n_test = 10000
    self.training = training

  def loss(self):
    '''
    define loss function
    use softmax cross entropy with logits as the loss function
    compute mean cross entropy, softmax is applied internally
    '''
    with tf.name_scope('loss'):
      entropy = tf.nn.softmax_cross_entropy_with_logits(labels=self.label,
logits=self.logits)
      self.loss = tf.reduce_mean(entropy, name='loss')

  def optimize(self):
    '''
    Define training op
    using Adam optimizer to minimize cost
    '''
    self.opt = tf.train.AdamOptimizer(self.lr).minimize(self.loss,
global_step=self.gstep)

  def eval(self):
    '''
    Count the number of right predictions in a batch
    '''
    with tf.name_scope('predict'):
      preds = tf.nn.softmax(self.logits)
      correct_preds = tf.equal(tf.argmax(preds, 1), tf.argmax(self.label,
1))
      self.accuracy = tf.reduce_sum(tf.cast(correct_preds, tf.float32))

  def summary(self):
    '''
    Create summaries to write on TensorBoard
    '''
    with tf.name_scope('summaries'):
      tf.summary.scalar('accuracy', self.accuracy)
      if self.training:
        tf.summary.scalar('loss', self.loss)
        tf.summary.histogram('histogram_loss', self.loss)
```

```
      self.summary_op = tf.summary.merge_all()

  def build(self, test_only=False):
    '''
    Build the computation graph
    '''
    self.img, self.label, self.train_init, self.test_init = \
            get_init_data(self.train_batch, self.test_batch,
test_only=test_only)

    self.logits = net_fn(self.img, n_classes=self.n_classes, \
            keep_prob=self.keep_prob, is_training=self.training)
    if self.training:
      self.loss()
      self.optimize()
    self.eval()
    self.summary()

  def train_one_epoch(self, sess, saver, writer, epoch, step):
    start_time = time.time()
    sess.run(self.train_init)
    total_loss = 0
    n_batches = 0
    tf.logging.info(time.strftime('time:%Y-%m-%d
%H:%M:%S',time.localtime(time.time())))
    try:
      while True:
        _, l, summaries = sess.run([self.opt, self.loss, self.summary_op])
        writer.add_summary(summaries, global_step=step)
        if (step + 1) % self.skip_step == 0:
          tf.logging.info('Loss at step {0}: {1}'.format(step+1, l))
        step += 1
        total_loss += l
        n_batches += 1
    except tf.errors.OutOfRangeError:
      pass
    #saver.save(sess, 'checkpoints/convnet_mnist/mnist-convnet', step)
    tf.logging.info('Average loss at epoch {0}: {1}'.format(epoch,
total_loss/n_batches))
    tf.logging.info('train one epoch took: {0} seconds'.format(time.time()
- start_time))
    return step

  def eval_once(self, sess, writer=None, step=None):
    start_time = time.time()
    sess.run(self.test_init)
    total_correct_preds = 0
    eval_step = 0
    try:
      while True:
        eval_step += 1
        accuracy_batch, summaries = sess.run([self.accuracy,
self.summary_op])
        writer.add_summary(summaries, global_step=step) if writer else None
        total_correct_preds += accuracy_batch
    except tf.errors.OutOfRangeError:
      pass
    tf.logging.info('Evaluation took: {0} seconds'.format(time.time() -
start_time))
    tf.logging.info('Accuracy : {0} \n'.format(total_correct_preds/
self.n_test))

  def train_eval(self, n_epochs=10, save_ckpt=None, restore_ckpt=None):
```

```
    '''
    The train function alternates between training one epoch and evaluating
    '''
    if restore_ckpt:
      writer = tf.summary.FileWriter('./graphs/convnet/finetune',
tf.get_default_graph())
    else:
      writer = tf.summary.FileWriter('./graphs/convnet/train',
tf.get_default_graph())
    with tf.Session() as sess:
      sess.run(tf.global_variables_initializer())
      saver = tf.train.Saver()
      if restore_ckpt:
        saver.restore(sess, restore_ckpt)
      step = self.gstep.eval()
      for epoch in range(n_epochs):
        step = self.train_one_epoch(sess, saver, writer, epoch, step)
        self.eval_once(sess, writer, step)
      saver.save(sess, save_ckpt)
    writer.close()
    tf.logging.info("Finish")

  def evaluate(self, restore_ckpt):
    '''
    The evaluating function
    '''
    with tf.Session() as sess:
      saver = tf.train.Saver()
      saver.restore(sess, restore_ckpt)
      step = self.gstep.eval()
      self.eval_once(sess)
    tf.logging.info("Finish")
```

ConvNet は、グラフの作成およびモデルの学習と評価が可能なクラスです。このフレームワークは、データ ユーティリティ、ネット定義、およびメトリクスを結合します。ConvNet クラスをインスタンシエートした後、クラス メソッド `build` を呼び出して学習用または評価用グラフ (`test_only` 引数 を true にするかどうかで指定) を作成することによって、モデルの学習と評価を実行できます。

モデルに学習させる

モデルに学習させるには、`train.py` という名前のファイルを作成し、次のコードを追加します。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
from train_eval_utils import ConvNet

tf.app.flags.DEFINE_string(
    'save_ckpt', '', 'Where to save checkpoint.')
FLAGS = tf.app.flags.FLAGS

def main(unused_argv):
  tf.logging.set_verbosity(tf.logging.INFO)
  tf.logging.info("Training model from scratch")
  net = ConvNet(True)
```

```
  net.build()
  net.train_eval(10, FLAGS.save_ckpt)

if __name__ == '__main__':
  tf.app.run()
```

シェルで `train.py` を実行します。

```
$ WORKSPACE=./models
$ BASELINE_CKPT=${WORKSPACE}/train/model.ckpt
$ mkdir -p $(dirname "${BASELINE_CKPT}")
$ python train.py --save_ckpt=${BASELINE_CKPT}
```

実行によって次のようなログが出力されます。

```
INFO:tensorflow:time:2019-01-09 16:14:44
INFO:tensorflow:Loss at step 500: 421.8246154785156
INFO:tensorflow:Loss at step 600: 305.761474609375
INFO:tensorflow:Loss at step 700: 167.25115966796875
INFO:tensorflow:Loss at step 800: 399.25732421875
INFO:tensorflow:Loss at step 900: 246.513000488828125
INFO:tensorflow:Average loss at epoch 1: 390.06004813383385
INFO:tensorflow:train one epoch took: 2.353825569152832 seconds
INFO:tensorflow:Evaluation took: 0.22740554809570312 seconds
INFO:tensorflow:Accuracy : 0.9435
```

数分後、学習済みチェックポイント `models/train/model.ckpt` が生成されます。

推論 GraphDef ファイルをエクスポートする

`export_inf_graph.py` という名前のファイルを作成し、次のコードを追加します。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf

from tensorflow.python.platform import gfile
from google.protobuf import text_format
from low_level_cnn import net_fn

tf.app.flags.DEFINE_integer(
    'image_size', None,
    'The image size to use, otherwise use the model default_image_size.')

tf.app.flags.DEFINE_integer(
    'batch_size', None,
    'Batch size for the exported model. Defaulted to "None" so batch size
can '
    'be specified at model runtime.')

tf.app.flags.DEFINE_string('dataset_name', 'imagenet',
                           'The name of the dataset to use with the model.')

tf.app.flags.DEFINE_string(
    'output_file', '', 'Where to save the resulting file to.')

FLAGS = tf.app.flags.FLAGS
```

```
def main(_):
  if not FLAGS.output_file:
    raise ValueError('You must supply the path to save to with --
output_file')
  tf.logging.set_verbosity(tf.logging.INFO)

  with tf.Graph().as_default() as graph:
    network_fn = net_fn
    image_size = FLAGS.image_size or network_fn.default_image_size
    image = tf.placeholder(name='image', dtype=tf.float32, \
                           shape=[FLAGS.batch_size, image_size,
image_size, 1])
    network_fn(image, is_training=False)
    graph_def = graph.as_graph_def()

    with gfile.GFile(FLAGS.output_file, 'w') as f:
      f.write(text_format.MessageToString(graph_def))
    tf.logging.info("Finish export inference graph")

if __name__ == '__main__':
    tf.app.run()
```

`export_inf_graph.py` を実行します。

```
$ WORKSPACE=./models
$ BASELINE_GRAPH=${WORKSPACE}/mnist.pbtxt
$ python export_inf_graph.py --output_file=${BASELINE_GRAPH}
```

モデル解析を実行する

ここまでの手順で学習済みチェックポイントと GraphDef ファイルが用意できました。これで、プルーニング プロセスを開始できます。次のシェル スクリプトを実行して、`vai_p_tensorflow` 関数を呼び出します。

```
WORKSPACE=./models
BASELINE_GRAPH=${WORKSPACE}/mnist.pbtxt
BASELINE_CKPT=${WORKSPACE}/train/model.ckpt
INPUT_NODES="image"
OUTPUT_NODES="logits/add"
action=ana

vai_p_tensorflow \
  --action=${action} \
  --input_graph=${BASELINE_GRAPH} \
  --input_ckpt=${BASELINE_CKPT} \
  --eval_fn_path=low_level_cnn.py \
  --target="accuracy" \
  --max_num_batches=100 \
  --workspace=${WORKSPACE} \
  --input_nodes="${INPUT_NODES}" \
  --input_node_shapes="1,28,28,1" \
  --output_nodes=\"${OUTPUT_NODES}\"
```

次のようなログが出力されます。

```
INFO:tensorflow:Starting evaluation at 2019-01-09-08:43:15
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from ./models/train/model.ckpt
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Evaluation [10/100]
```

```
INFO:tensorflow:Evaluation [20/100]
INFO:tensorflow:Evaluation [30/100]
INFO:tensorflow:Evaluation [40/100]
INFO:tensorflow:Evaluation [50/100]
INFO:tensorflow:Evaluation [60/100]
INFO:tensorflow:Evaluation [70/100]
INFO:tensorflow:Evaluation [80/100]
INFO:tensorflow:Evaluation [90/100]
INFO:tensorflow:Evaluation [100/100]
INFO:tensorflow:Finished evaluation at 2019-01-09-08:43:21
```

モデルをプルーニングする

次に、モデルをプルーニングします。`vai_p_tensorflow` 関数を呼び出すシェル スクリプトを作成します。

```
WORKSPACE=./models
BASELINE_GRAPH=${WORKSPACE}/mnist.pbtxt
BASELINE_CKPT=${WORKSPACE}/train/model.ckpt
PRUNED_GRAPH=${WORKSPACE}/pruned/graph.pbtxt
PRUNED_CKPT=${WORKSPACE}/pruned/sparse.ckpt
INPUT_NODES="image"
OUTPUT_NODES="logits/add"
action=prune

mkdir -p $(dirname "${PRUNED_GRAPH}")
vai_p_tensorflow \
  --action=${action} \
  --input_graph=${BASELINE_GRAPH} \
  --input_ckpt=${BASELINE_CKPT} \
  --output_graph=${PRUNED_GRAPH} \
  --output_ckpt=${PRUNED_CKPT} \
  --workspace=${WORKSPACE} \
  --input_nodes="${INPUT_NODES}" \
  --input_node_shapes="1,28,28,1" \
  --output_nodes="${OUTPUT_NODES}" \
  --sparsity=0.5 \
  --gpu="0,1,2,3" \
  2>&1 | tee prune.log
```

プルーニング済みモデルを微調整する

`ft.py` という名前のファイルを作成し、次のコードを追加します。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
from train_eval_utils import ConvNet

tf.app.flags.DEFINE_string(
    'checkpoint_path', '', 'Where to restore checkpoint.')
tf.app.flags.DEFINE_string(
    'save_ckpt', '', 'Where to save checkpoint.')
FLAGS = tf.app.flags.FLAGS

def main(unused_argv):
  tf.logging.set_verbosity(tf.logging.INFO)
  tf.logging.info("Finetuning model")

  tf.set_pruning_mode()
```

```
  net = ConvNet(True)
  net.build()
  net.train_eval(10, FLAGS.save_ckpt, FLAGS.checkpoint_path)

if __name__ == '__main__':
  tf.app.run()
```

注記: モデルを作成する前に、`tf.set_pruning_mode()` を呼び出す必要があります。この API を使用すると、「スパース (疎) 学習」モードが有効になります。このモードでは、学習中にプルーニング済みチャネルの重みは 0 に維持され、アップデートされません。この関数を呼び出さずにプルーニング済みモデルを微調整すると、プルーニング済みチャネルはアップデートされ、最終的には通常の非スパース モデルが生成されます。

プルーニング済みモデルの微調整は、モデルを新規に学習させるのと基本的には同じで、`ft.py` を実行します。

```
WORKSPACE=./models
FT_CKPT=${WORKSPACE}/ft/model.ckpt
PRUNED_CKPT=${WORKSPACE}/pruned/sparse.ckpt
python -u ft.py \
    --save_ckpt=${FT_CKPT} \
    --checkpoint_path=${PRUNED_CKPT} \
    2>&1 | tee ft.log
```

次のようなログが出力されます。

```
INFO:tensorflow:time:2019-01-09 17:17:10
INFO:tensorflow:Loss at step 1000: 13.077235221862793
INFO:tensorflow:Loss at step 1100: 41.67073440551758
INFO:tensorflow:Loss at step 1200: 31.98809242248535
INFO:tensorflow:Loss at step 1300: 34.46034240722656
INFO:tensorflow:Loss at step 1400: 32.12882995605469
INFO:tensorflow:Average loss at epoch 2: 28.96098704302489
INFO:tensorflow:train one epoch took: 3.0082509517669678 seconds
INFO:tensorflow:Evaluation took: 0.23403644561767578 seconds
INFO:tensorflow:Accuracy : 0.9539
```

最後に、微調整済みモデルを変換およびフリーズして、デンス (密) モデルを生成する必要があります。

```
WORKSPACE=./models
FT_CKPT=${WORKSPACE}/ft/model.ckpt
TRANSFORMED_CKPT=${WORKSPACE}/pruned/transformed.ckpt
PRUNED_GRAPH=${WORKSPACE}/pruned/graph.pbtxt
FROZEN_PB=${WORKSPACE}/pruned/mnist.pb
OUTPUT_NODES="logits/add"

vai_p_tensorflow \
    --action=transform \
    --input_ckpt=${FT_CKPT} \
--output_ckpt=${TRANSFORMED_CKPT}


freeze_graph \
    --input_graph="${PRUNED_GRAPH}" \
    --input_checkpoint="${TRANSFORMED_CKPT}" \
    --input_binary=false \
    --output_graph="${FROZEN_PB}" \
--output_node_names=${OUTPUT_NODES}
```

これで、models/pruned ディレクトリに `mninst.pb` という名前のフリーズ済み GraphDef ファイルが作成されます。

Estimator

CNN MNIST 分類器を作成する

`est_cnn.py` という名前のファイルを作成し、次のコードを追加します。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# Imports
import numpy as np
import tensorflow as tf

# Our application logic will be added here
def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

  # Convolutional Layer #1
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Pooling Layer #1
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

  # Convolutional Layer #2 and Pooling Layer #2
  conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=64,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)
  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

  # Dense Layer
  pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
  dense = tf.layers.dense(inputs=pool2_flat, units=1024,
activation=tf.nn.relu)
  dropout = tf.layers.dropout(
      inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

  # Logits Layer
  logits = tf.layers.dense(inputs=dropout, units=10)

  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
      # `logging_hook`.
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }

  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
```

```
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels,
logits=logits)

  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)

# Load training and eval data
mnist = tf.contrib.learn.datasets.load_dataset("mnist")
train_data = mnist.train.images # Returns np.array
train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
eval_data = mnist.test.images # Returns np.array
eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

def train_input_fn():
  return tf.estimator.inputs.numpy_input_fn(
      x={"x": train_data},
      y=train_labels,
      batch_size=100,
      num_epochs=None,
      shuffle=True)

def eval_input_fn():
  return tf.estimator.inputs.numpy_input_fn(
      x={"x": eval_data},
      y=eval_labels,
      num_epochs=1,
      shuffle=False)

def model_fn():
  return tf.estimator.Estimator(
      model_fn=cnn_model_fn, model_dir="./models/train/")
```

`cnn_model_fn` 関数は、TensorFlow のエスティメーター API で定義されたインターフェイスに準拠しています。この関数は MNIST の特徴データ、ラベル、およびモードを引数に取り、たたみ込み層と活性化層を作成し、予測、精度の低下、および学習演算を返します。

`train_input_fn` と `eval_input_fn` は、それぞれ学習中と評価中にネットワークにデータを供給する関数です。

ベースライン モデルの学習

エスティメーターを作成し、エスティメーター上で `train()` を呼び出してモデルに学習させるには、`train.py` という名前のファイルを作成し、次のコードを追加します。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from est_cnn import model_fn, train_input_fn, eval_input_fn

# Imports
import numpy as np
import tensorflow as tf

tf.logging.set_verbosity(tf.logging.INFO)

def main(unused_argv):
  mnist_classifier = model_fn()

  mnist_classifier.train(
      input_fn=train_input_fn(),
      max_steps=20000)

  eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn())
  print(eval_results)

if __name__ == "__main__":
  tf.app.run()
```

`train.py` を実行します。

```
$ python train.py
```

モデルの学習中、次のような出力が表示されます。

```
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 0 into ./models/train/model.ckpt.
INFO:tensorflow:loss = 2.294087, step = 0
INFO:tensorflow:global_step/sec: 201.741
INFO:tensorflow:loss = 2.2876544, step = 100 (0.496 sec)
INFO:tensorflow:global_step/sec: 228.126
INFO:tensorflow:loss = 2.2656975, step = 200 (0.439 sec)
INFO:tensorflow:global_step/sec: 225.094
INFO:tensorflow:loss = 2.2483034, step = 300 (0.444 sec)
INFO:tensorflow:global_step/sec: 234.019
…
INFO:tensorflow:Saving dict for global step 20000: accuracy = 0.9684,
global_step = 20000, loss = 0.10172604
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 20000: ./
models/train/model.ckpt-20000
{'accuracy': 0.9684, 'loss': 0.10172604, 'global_step': 20000}
```

テスト データセットでは 96.84% の精度を達成しています。

推論 GraphDef ファイルをエクスポートする

`export_inf_graph.py` という名前のファイルを作成し、次のコードを追加します。

```
from google.protobuf import text_format
from est_cnn import cnn_model_fn
from tensorflow.keras import backend as K
from tensorflow.python.platform import gfile
import tensorflow as tf

tf.app.flags.DEFINE_string(
    'output_file', '', 'Where to save the resulting file to.')

FLAGS = tf.app.flags.FLAGS

def main(_):
  if not FLAGS.output_file:
    raise ValueError('You must supply the path to save to with --
output_file')
  tf.logging.set_verbosity(tf.logging.INFO)

  with tf.Graph().as_default() as graph:
    image = tf.placeholder(name='image', dtype=tf.float32,
                           shape=[1, 28, 28, 1])
    label = tf.placeholder(name='label', dtype=tf.int32, shape=[1])

    cnn_model_fn({"x": image}, label, tf.estimator.ModeKeys.EVAL)
    graph_def = graph.as_graph_def()
    with gfile.GFile(FLAGS.output_file, 'w') as f:
      f.write(text_format.MessageToString(graph_def))
    print("Finish export inference graph")

if __name__ == '__main__':
  tf.app.run()
```

モデル解析を実行する

ここまでの手順で学習済みチェックポイントと GraphDef ファイルが用意できました。これで、プルーニングを開始できます。

`vai_p_tensorflow` 関数を呼び出すシェル スクリプトを作成します。

```
WORKSPACE=./models

BASELINE_GRAPH=${WORKSPACE}/mnist.pbtxt
BASELINE_CKPT=${WORKSPACE}/train/model.ckpt-20000
INPUT_NODES="image"
OUTPUT_NODES="softmax_tensor"

action=ana
vai_p_tensorflow \
    --action=${action} \
    --input_graph=${BASELINE_GRAPH} \
    --input_ckpt=${BASELINE_CKPT} \
    --eval_fn_path=est_cnn.py \
    --target="accuracy" \
```

```
    --max_num_batches=500 \
    --workspace=${WORKSPACE} \
    --input_nodes="${INPUT_NODES}" \
    --input_node_shapes="1,28,28,1" \
--output_nodes=\"${OUTPUT_NODES}\"
```

`est_cnn.py` では、モデルの精度を計算する tf.metrics.accuracy の演算を「accuracy」という名前で既に定義してあります。

```
eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
```

この演算を使用してモデルの精度を評価するには、--target="accuracy" と設定します。

モデルをプルーニングする

```
PRUNED_GRAPH=${WORKSPACE}/pruned/graph.pbtxt
PRUNED_CKPT=${WORKSPACE}/pruned/sparse.ckpt

action=prune
vai_p_tensorflow \
    --action=${action} \
    --input_graph=${BASELINE_GRAPH} \
    --input_ckpt=${BASELINE_CKPT} \
    --output_graph=${PRUNED_GRAPH} \
    --output_ckpt=${PRUNED_CKPT} \
    --workspace=${WORKSPACE} \
    --input_nodes="${INPUT_NODES}" \
    --input_node_shapes="1,28,28,1" \
    --output_nodes="${OUTPUT_NODES}" \
    --sparsity=0.2 \
--gpu="0,1,2,3"
```

プルーニング済みモデルを微調整する

`ft.py` という名前のファイルを作成し、次のコードを追加します。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from est_cnn import cnn_model_fn, train_input_fn

# Imports
import numpy as np
import tensorflow as tf

tf.app.flags.DEFINE_string(
    'checkpoint_path', None, 'Path of a specific checkpoint to finetune.')

FLAGS = tf.app.flags.FLAGS

tf.logging.set_verbosity(tf.logging.INFO)

def main(unused_argv):
  tf.set_pruning_mode()
  ws = tf.estimator.WarmStartSettings(
      ckpt_to_initialize_from=FLAGS.checkpoint_path)
  mnist_classifier = tf.estimator.Estimator(
      model_fn=cnn_model_fn, model_dir="./models/ft/", warm_start_from=ws)
```

```
  mnist_classifier.train(
      input_fn=train_input_fn(),
      max_steps=20000)

if __name__ == "__main__":
  tf.app.run()
```

tf.estimator.WarmStartSettings を使用してプルーニング済みチェックポイントをロードし、そこから微調整を開始します。

`ft.py` を実行して、プルーニング済みモデルを微調整します。

```
python -u ft.py --checkpoint_path=${PRUNED_CKPT}
```

次のようなログが出力されます。

```
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 0 into ./models/ft/model.ckpt.
INFO:tensorflow:loss = 0.3675258, step = 0
INFO:tensorflow:global_step/sec: 162.673
INFO:tensorflow:loss = 0.31534952, step = 100 (0.615 sec)
INFO:tensorflow:global_step/sec: 210.058
INFO:tensorflow:loss = 0.2782951, step = 200 (0.476 sec)
...
INFO:tensorflow:loss = 0.022076223, step = 19800 (0.503 sec)
INFO:tensorflow:global_step/sec: 206.588
INFO:tensorflow:loss = 0.06927078, step = 19900 (0.484 sec)
INFO:tensorflow:Saving checkpoints for 20000 into ./models/ft/model.ckpt.
INFO:tensorflow:Loss for final step: 0.07726018.
```

最後に、微調整済みモデルを変換およびフリーズして、デンス (密) モデルを生成します。

```
FT_CKPT=${WORKSPACE}/ft/model.ckpt-20000
TRANSFORMED_CKPT=${WORKSPACE}/pruned/transformed.ckpt
FROZEN_PB=${WORKSPACE}/pruned/mnist.pb

vai_p_tensorflow \
    --action=transform \
    --input_ckpt=${FT_CKPT} \
--output_ckpt=${TRANSFORMED_CKPT}

freeze_graph \
--input_graph="${PRUNED_GRAPH}" \
--input_checkpoint="${TRANSFORMED_CKPT}" \
--input_binary=false  \
--output_graph="${FROZEN_PB}" \
--output_node_names=${OUTPUT_NODES}
```

これで、`mninst.pb` という名前のフリーズ済み GraphDef ファイルが作成されます。

VGG-16

ここでは、実際のモデルに対して vai_p_tensorflow を実行する方法を紹介します。VGG (https://arxiv.org/abs/1409.1557) は、大規模な画像認識用のネットワークです。この例では、TensorFlow-Slim 画像分類モデル ライブラリにある学習済み VGG-16 モデルを使用します。

Tensorflow-Slim のリポジトリと、その中にある学習済み VGG16 モデルをダウンロードします。

```
$ git clone https://github.com/tensorflow/models.git
$ cd models/research/slim
# mkdir models/vgg16 && cd models/vgg16
$ wget http://download.tensorflow.org/models/vgg_16_2016_08_28.tar.gz
$ tar xzvf vgg_16_2016_08_28.tar.gz
```

次の説明を参照して、ImageNet データセットを準備します。

https://github.com/tensorflow/models/blob/master/research/inception/README.md#getting-started

`vgg16_eval.py` という名前のグラフ評価用スクリプトを作成します。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import math
import tensorflow as tf

from tensorflow.python.summary import summary
from tensorflow.python.training import monitored_session
from tensorflow.python.training import saver as tf_saver

from datasets import dataset_factory
from nets import nets_factory
from preprocessing import preprocessing_factory

slim = tf.contrib.slim

dataset_name='imagenet'
dataset_split_name='validation'
dataset_dir='/dataset/imagenet/tf_records'
model_name='vgg_16'
labels_offset=1
batch_size=100
num_preprocessing_threads=4

def model_fn():
  tf.logging.set_verbosity(tf.logging.INFO)

  tf_global_step = slim.get_or_create_global_step()

  ######################
  # Select the dataset #
  ######################
  dataset = dataset_factory.get_dataset(dataset_name,
                                        dataset_split_name,
                                        dataset_dir)

####################
  # Select the model #
  ####################
  network_fn = nets_factory.get_network_fn(
      model_name,
      num_classes=(dataset.num_classes - labels_offset),
      is_training=False)

  ##############################################################
  # Create a dataset provider that loads data from the dataset #
  ##############################################################
```

```
  provider = slim.dataset_data_provider.DatasetDataProvider(
      dataset,
      shuffle=False,
      common_queue_capacity=2 * batch_size,
      common_queue_min=batch_size)
  [image, label] = provider.get(['image', 'label'])
  label -= labels_offset

  #####################################
  # Select the preprocessing function #
  #####################################
  preprocessing_name = model_name
  image_preprocessing_fn = preprocessing_factory.get_preprocessing(
      preprocessing_name,
      is_training=False)

  eval_image_size = network_fn.default_image_size

  image = image_preprocessing_fn(image, eval_image_size, eval_image_size)

  images, labels = tf.train.batch(
      [image, label],
      batch_size=batch_size,
      num_threads=num_preprocessing_threads,
      capacity=5 * batch_size)

####################
  # Define the model #
  ####################
  logits, _ = network_fn(images)

  variables_to_restore = slim.get_variables_to_restore()

  predictions = tf.argmax(logits, 1)
  org_labels = labels
  labels = tf.squeeze(labels)

  eval_metric_ops = {
      'top-1': slim.metrics.streaming_accuracy(predictions, labels),
      'top-5': slim.metrics.streaming_recall_at_k(logits, org_labels, 5)
  }
  return eval_metric_ops
```

バイナリ ファイルではなく、可読性の高いテキスト ファイルを出力するように `models/research/slim/export_inference_graph.py` を編集します。

```
+ from google.protobuf import text_format

-   with gfile.GFile(FLAGS.output_file, 'wb') as f:
-     f.write(graph_def.SerializeToString())
+   with gfile.GFile(FLAGS.output_file, 'w') as f:
+     f.write(text_format.MessageToString(graph_def))
```

推論グラフをエクスポートします。

```
python export_inference_graph.py \
    --model_name=vgg_16 \
    --output_file=vgg_16_inf_graph.pbtxt \
    --dataset_dir=/opt/dataset/tf_records
```

モデル解析を実行します。

```
vai_p_tensorflow \
  --action=ana \
  --input_graph=vgg_16_inf_graph.pbtxt \
  --input_ckpt=vgg_16.ckpt \
  --eval_fn_path=vgg_16_eval.py \
  --target=top-5 \
  --max_num_batches=500 \
  --workspace=/home/deephi/models/research/slim/models/vgg16 \
  --exclude="vgg_16/fc6/Conv2D, vgg_16/fc7/Conv2D, vgg_16/fc8/Conv2D" \
--output_nodes="vgg_16/fc8/squeezed"
```

`vgg_16_eval.py` で、変数 batch_size の初期値を 100 に定義しています。ImageNet の検証用データセットには 50,000 個のサンプルがあるため、評価時にデータセットのすべてのサンプルをテストできるように、max_num_steps を 500 に設定します。

重要: 接頭辞に vgg_16/fc が付いたノードは、ネットワークの出力ラベルの数に影響を与えます。したがって、データセットとの形状の不一致を避けるため、これらのノードは除外しておく必要があります。

モデルのプルーニングを実行します。

```
vai_p_tensorflow \
  --action=prune \
  --input_graph=vgg_16_inf_graph.pbtxt \
  --input_ckpt=vgg_16.ckpt \
  --output_graph=sparse_graph.pbtxt \
  --output_ckpt=sparse.ckpt \
  --workspace=/home/deephi/models/research/slim/models/vgg16 \
  --sparsity=0.15 \
  --exclude="vgg_16/fc6/Conv2D, vgg_16/fc7/Conv2D, vgg_16/fc8/Conv2D" \
  --output_nodes="vgg_16/fc8/squeezed"
```

`models/research/slim/train_image_classifier.py` を開いて、`main()` 関数の最初に次の行を挿入します。

```
def main():
+   tf.set_pruning_mode()
```

プルーニング済みモデルを微調整します。

```
python train_image_classifier.py \
    --model_name=vgg_16 \
    --train_dir=./models/vgg16/ft \
    --dataset_name=imagenet \
    --dataset_dir=/opt/dataset/tf_records \
    --dataset_split_name=train \
    --checkpoint_path=./models/vgg16/sparse.ckpt \
    --labels_offset=0 \
    --save_interval_secs=600 \
    --batch_size=32 \
    --num_clones=4 \
    --weight_decay=5e-4 \
    --optimizer=adam \
    --learning_rate=1e-2 \
    --learning_rate_decay_type=polynomial \
    --decay_steps=200000 \
--max_number_of_steps=200000
```

密なチェックポイントを生成し、グラフをフリーズします。

```
vai_p_tensorflow \
--action=transform \
--input_ckpt=./models/vgg16/ft/model.ckpt-200000 \
--output_ckpt=./models/vgg16/dense.ckpt

freeze_graph.py \
--input_graph=./models/vgg16/sparse_graph.pbtxt \
--input_checkpoint=./models/vgg16/dense.ckpt \
--input_binary=false  \
--output_graph=./models/vgg16/vgg16_pruned.pb \
--output_node_names=”vgg_16/fc8/squeezed"
```

量子化のプロセスはほとんど同じため、このユーザー ガイドには Resnet_v1_50 の量子化のみを含めています。詳細は、こちら を参照してください。

PyTorch の例

ResNet18

モデル学習用のコードを作成します。

```
import argparse
import os
import shutil
import time
import torch
import torchvision.datasets as datasets
import torchvision.transforms as transforms

from torchvision.models.resnet import resnet18

from pytorch_nndct import Pruner
from pytorch_nndct import InputSpec

parser = argparse.ArgumentParser()
parser.add_argument(
    '--data_dir',
    default='/scratch/workspace/dataset/imagenet/pytorch',
    help='Data set directory.')
parser.add_argument(
    '--pretrained',
    default='/scratch/workspace/wangyu/nndct_test_data/models/resnet18.pth',
    help='Trained model file path.')
parser.add_argument(
    '--ratio',
    default=0.1,
    type=float,
    help='Desired pruning ratio. The larger this value, the smaller'
    'the model after pruning.')
parser.add_argument(
    '--ana',
    default=False,
    type=bool,
    help='Whether to perform model analysis.')
```

```
args, _ = parser.parse_known_args()

class AverageMeter(object):
  """Computes and stores the average and current value"""

  def __init__(self, name, fmt=':f'):
    self.name = name
    self.fmt = fmt
    self.reset()

  def reset(self):
    self.val = 0
    self.avg = 0
    self.sum = 0
    self.count = 0

  def update(self, val, n=1):
    self.val = val
    self.sum += val * n
    self.count += n
    self.avg = self.sum / self.count

  def __str__(self):
    fmtstr = '{name} {val' + self.fmt + '} ({avg' + self.fmt + '})'
    return fmtstr.format(**self.__dict__)

def save_checkpoint(state, is_best, filename='checkpoint.pth.tar'):
  torch.save(state, filename)
  if is_best:
    shutil.copyfile(filename, 'model_best.pth.tar')

class ProgressMeter(object):

  def __init__(self, num_batches, meters, prefix=""):
    self.batch_fmtstr = self._get_batch_fmtstr(num_batches)
    self.meters = meters
    self.prefix = prefix

  def display(self, batch):
    entries = [self.prefix + self.batch_fmtstr.format(batch)]
    entries += [str(meter) for meter in self.meters]
    print('\t'.join(entries))

  def _get_batch_fmtstr(self, num_batches):
    num_digits = len(str(num_batches // 1))
    fmt = '{:' + str(num_digits) + 'd}'
    return '[' + fmt + '/' + fmt.format(num_batches) + ']'

def accuracy(output, target, topk=(1,)):
  """Computes the accuracy over the k top predictions
    for the specified values of k"""
  with torch.no_grad():
    maxk = max(topk)
    batch_size = target.size(0)

    _, pred = output.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(target.view(1, -1).expand_as(pred))

    res = []
    for k in topk:
      correct_k = correct[:k].view(-1).float().sum(0, keepdim=True)
      res.append(correct_k.mul_(100.0 / batch_size))
```

```
    return res

def adjust_learning_rate(optimizer, epoch, lr):
  """Sets the learning rate to the initial LR decayed by every 2 epochs"""
  lr = lr * (0.1**(epoch // 2))

  for param_group in optimizer.param_groups:
    param_group['lr'] = lr

def train(train_loader, model, criterion, optimizer, epoch):
  batch_time = AverageMeter('Time', ':6.3f')
  data_time = AverageMeter('Data', ':6.3f')
  losses = AverageMeter('Loss', ':.4e')
  top1 = AverageMeter('Acc@1', ':6.2f')
  top5 = AverageMeter('Acc@5', ':6.2f')
  progress = ProgressMeter(
      len(train_loader), [batch_time, data_time, losses, top1, top5],
      prefix="Epoch: [{}]".format(epoch))

  # switch to train mode
  model.train()

  end = time.time()
  for i, (images, target) in enumerate(train_loader):
    # measure data loading time
    data_time.update(time.time() - end)

    model = model.cuda()
    images = images.cuda()
    target = target.cuda()

    # compute output
    output = model(images)
    loss = criterion(output, target)

    # measure accuracy and record loss
    acc1, acc5 = accuracy(output, target, topk=(1, 5))
    losses.update(loss.item(), images.size(0))
    top1.update(acc1[0], images.size(0))
    top5.update(acc5[0], images.size(0))

    # compute gradient and do SGD step
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # measure elapsed time
    batch_time.update(time.time() - end)
    end = time.time()

    if i % 10 == 0:
      progress.display(i)

def evaluate(val_loader, model, criterion):
  batch_time = AverageMeter('Time', ':6.3f')
  losses = AverageMeter('Loss', ':.4e')
  top1 = AverageMeter('Acc@1', ':6.2f')
  top5 = AverageMeter('Acc@5', ':6.2f')
  progress = ProgressMeter(
      len(val_loader), [batch_time, losses, top1, top5], prefix='Test: ')

  # switch to evaluate mode
  model.eval()
```

```
  with torch.no_grad():
    end = time.time()
    for i, (images, target) in enumerate(val_loader):
      model = model.cuda()
      images = images.cuda(non_blocking=True)
      target = target.cuda(non_blocking=True)

      # compute output
      output = model(images)
      loss = criterion(output, target)

      # measure accuracy and record loss
      acc1, acc5 = accuracy(output, target, topk=(1, 5))
      losses.update(loss.item(), images.size(0))
      top1.update(acc1[0], images.size(0))
      top5.update(acc5[0], images.size(0))

      # measure elapsed time
      batch_time.update(time.time() - end)
      end = time.time()

      if i % 50 == 0:
        progress.display(i)

    # TODO: this should also be done with the ProgressMeter
    print(' * Acc@1 {top1.avg:.3f} Acc@5 {top5.avg:.3f}'.format(
        top1=top1, top5=top5))

  return top1.avg, top5.avg
```

モデル解析に使用する関数を定義します。

```
def ana_eval_fn(model, val_loader, loss_fn):
  return evaluate(val_loader, model, loss_fn)[1]
```

resnet18 モデルを作成し、プルーニング API を追加してプルーニングを実行します。

```
if __name__ == '__main__':
  model = resnet18().cpu()
  model.load_state_dict(torch.load(args.pretrained))

  batch_size = 128
  workers = 4
  traindir = os.path.join(args.data_dir, 'train')
  valdir = os.path.join(args.data_dir, 'validation')

  normalize = transforms.Normalize(
      mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

  train_dataset = datasets.ImageFolder(
      traindir,
      transforms.Compose([
          transforms.RandomResizedCrop(224),
          transforms.RandomHorizontalFlip(),
          transforms.ToTensor(),
          normalize,
      ]))

  train_loader = torch.utils.data.DataLoader(
      train_dataset,
      batch_size=batch_size,
```

```
      shuffle=True,
      num_workers=workers,
      pin_memory=True)

  val_dataset = datasets.ImageFolder(
      valdir,
      transforms.Compose([
          transforms.Resize(256),
          transforms.CenterCrop(224),
          transforms.ToTensor(),
          normalize,
      ]))

  val_loader = torch.utils.data.DataLoader(
      val_dataset,
      batch_size=batch_size,
      shuffle=False,
      num_workers=workers,
      pin_memory=True)

  criterion = torch.nn.CrossEntropyLoss().cuda()

  pruner = Pruner(model, InputSpec(shape=(3, 224, 224),
dtype=torch.float32))
  if args.ana:
    pruner.ana(ana_eval_fn, args=(val_loader, criterion), gpus=[0, 1, 2, 3])
  model = pruner.prune(ratio=args.ratio)
  pruner.summary(model)

  lr = 1e-4
  optimizer = torch.optim.Adam(model.parameters(), lr, weight_decay=1e-4)

  best_acc5 = 0
  epochs = 1
  for epoch in range(epochs):
    adjust_learning_rate(optimizer, epoch, lr)

    train(train_loader, model, criterion, optimizer, epoch)

    acc1, acc5 = evaluate(val_loader, model, criterion)

    # remember best acc@1 and save checkpoint
    is_best = acc5 > best_acc5
    best_acc5 = max(acc5, best_acc5)

    if is_best:
      model.save('resnet18_sparse.pth.tar')
      torch.save(model.state_dict(), 'resnet18_final.pth.tar')
```

学習済み ResNet18 モデルをダウンロードします。

```
wget https://download.pytorch.org/models/resnet18-5c106cde.pth -O
resnet18.pth
```

ImageNet データセットを準備します (`http://image-net.org/download-images` 参照)。

1 回目のプルーニングをモデル解析と共に実行します。

```
$ python -u resnet18_pruning.py --data_dir imagenet_dir --pretrained
resnet18.pth --ratio 0.1  --ana True
```

2 回目以降のプルーニングにはモデル解析は必要ありません。プルーニング率を高くして、前のプルーニングで保存されたスパース チェックポイントを学習済みの重みとして使用するだけです。

```
$ python -u resnet18_pruning.py --data_dir imagenet_dir --pretrained
resnet18_sparse.pth.tar --ratio 0.2
```

Caffe の例

VGG-16

ベースライン モデル

VGG は、大規模な画像認識用のネットワークです。VGG16 のアーキテクチャについては、https://arxiv.org/abs/1409.1556 を参照してください。

コンフィギュレーション ファイルを作成する

`config.prototxt` という名前のファイルを作成します。

```
workspace: "examples/decent_p/"
gpu: "0,1,2,3"
test_iter: 100
acc_name: "top-1"

model: "examples/decent_p/vgg.prototxt"
weights: "examples/decent_p/vgg.caffemodel"
solver: "examples/decent_p/solver.prototxt"

rate: 0.1

pruner {
  method: REGULAR
}
```

モデル解析を実行する

```
$ ./vai_p_caffe ana -config config.prototxt
```

モデルをプルーニングする

```
$ ./vai_p_caffe prune -config config.prototxt
```

プルーニング済みモデルを微調整する

微調整には、次のソルバー設定を使用できます。

```
net: "vgg16/train_val.prototxt"
test_iter: 1250
test_interval: 1000
test_initialization: true
display: 100
```

```
average_loss: 100
base_lr: 0.004
lr_policy: "poly"
power: 1
gamma: 0.1
max_iter: 500000
momentum: 0.9
weight_decay: 0.0001
snapshot: 1000
snapshot_prefix: "vgg16/snapshot/res"
solver_mode: GPU
iter_size: 1
```

次のコマンドを使用して微調整を開始します。

```
$ ./vai_p_caffe finetune -config config.prototxt
```

推定所要時間: ImageNet の学習用データセット (ILSVRC2012) を使用した場合、30 エポックで約 70 時間 (120 万の画像、4 x NVIDIA Tesla V100)。

最終出力を生成する

プルーニングの反復を数回実行後、ベースラインに対して必要な演算が 33% に削減されたプルーニング モデルが取得されます。

次を実行してモデルを完成させます。

```
$ ./vai_p_caffe transform -model baseline.prototxt -weights
finetuned_model.caffemodel -output
final.caffemodel
```

プルーニングの結果

- データセット: ImageNet (ILSVRC2012)
- Input Size: 224 x 224
- GPU プラットフォーム: 4 x NVIDIA Tesla V100
- FLOPs: 30G
- パラメーター数: 24M

表 6: **XFPN のプルーニング結果**

丸め処理	FLOPs	パラメーター	Top-1/Top-5 精度
0	100%	100%	0.7096/0.8984
1	50%	57.3%	0.7020/0.8970
2	9.7%	35.8%	0.6912/0.8913

SSD (Single Shot Multibox Detector)

ベースライン モデル

SSD (https://arxiv.org/abs/1512.02325) は、画像内のオブジェクトを検出するためのディープ ニューラル ネットワークです。この例では、モデルのバックボーンとして VGG16 を使用しています。

コンフィギュレーション ファイルを作成する

`config.prototxt` という名前のファイルを作成します。

```
workspace: "examples/decent_p/ssd/"

model: "examples/decent_p/ssd/float.prototxt"
weights: "examples/decent_p/ssd/float.caffemodel"
solver: "examples/decent_p/ssd/solver.prototxt"

gpu: "0,1,2,3"
test_iter: 10
acc_name: "detection_eval"
ssd_ap_version: "11point"

rate: 0.15

pruner {
  method: REGULAR

  exclude {
    layer_top:
      "conv4_3_norm_mbox_loc"
    layer_top:
      "conv4_3_norm_mbox_conf"
    layer_top: "fc7_mbox_loc"
    layer_top: "fc7_mbox_conf"
    layer_top:
      "conv6_2_mbox_loc"
    layer_top:
      "conv6_2_mbox_conf"
    layer_top:
      "conv7_2_mbox_loc"
    layer_top:
      "conv7_2_mbox_conf"
    layer_top:
      "conv8_2_mbox_loc"
layer_top:
      "conv8_2_mbox_conf"
layer_top:
      "conv9_2_mbox_loc"
    layer_top:
      "conv9_2_mbox_conf"
}
}
```

SSD ネットワークの性質上、一部のたたみ込み層のフィルター数は固定する必要があるため、これらのレイヤーはプルーニングから除外します。上記の例では、「exclude」セクションに除外するレイヤー トップの名前を指定しています。通常、たたみ込み層はラベルを使用して直接計算すると、プルーニングできません。たとえば、ラベルを使用してたたみ込み層の出力を計算して top-5 の精度を得る必要がある場合、プルーニングから除外します。ラベルのクラス数は固定されているため、このたたみ込み層の出力サイズがラベルと一致していることを確認する必要があります。

モデル解析を実行する

```
$ ./vai_p_caffe ana -config config.prototxt
```

モデルをプルーニングする

```
$ ./vai_p_caffe prune -config config.prototxt
```

プルーニング済みモデルを微調整する

微調整の初期パラメーターとして、次のソルバー設定を使用できます。

```
net: "float.prototxt"
test_iter: 229
test_interval: 500
base_lr: 0.001
display: 10
max_iter: 120000
lr_policy: "multistep"
gamma: 0.1
momentum: 0.9
weight_decay: 0.0005
snapshot: 500
snapshot_prefix: "SSD_"
solver_mode: GPU
device_id: 4
debug_info: false
snapshot_after_train: true
test_initialization: false
average_loss: 10
stepvalue: 80000
stepvalue: 100000
stepvalue: 120000
iter_size: 1
type: "SGD"
eval_type: "detection"
ap_version: "11point"
```

```
$ ./vai_p_caffe finetune -config config.prototxt
```

推定所要時間: Cityscapes の学習用データセットを使用した場合、650 エポックで約 50 時間 (2975 画像、4 x NVIDIA Tesla V100)。

最終出力を生成する

モデルを完成させるには、次を実行します。

```
$ ./vai_p_caffe transform -model baseline.prototxt -weights
finetuned_model.caffemodel -output
final.caffemodel
```

プルーニングの結果

- データセット: Cityscapes (4 クラス)
- Input Size: 500 x 500

- GPU プラットフォーム: 4 x NVIDIA Tesla V100
- FLOPs: 173G
- パラメーター数: 24M

表 7: **SSD のプルーニング結果**

丸め処理	FLOPs	パラメーター	mAP
0	100%	100%	0.571
1	50%	29%	0.587
2	9.7%	9.7%	0.559

XFPN

ベースライン モデル

XFPN は、セグメンテーション タスク用のネットワークです。主に GoogleNet_v1 と FPN で構成されています。ネットワークの最後の数レイヤーは次のように定義されます。

```
layer {
    bottom: "add_p2"
    top: "pred"
    name: "toplayer_p2"
    type: "Deconvolution"
    convolution_param {
        num_output: 19
        kernel_size: 4
        pad: 1
        stride: 2
        weight_filler {
            type: "gaussian"
            std: 0.01
        }
        bias_filler {
            type: "constant"
            value: 0
        }
    }
}

layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "pred"
    bottom: "label"
    top: "loss"
    loss_weight: 1
    include {
    phase: TRAIN
    }
    loss_param {
    ignore_label: 255
    }
}
layer {
    name: "result"
    type: "Softmax"
```

```
    bottom: "pred"
    top: "result"
    include {
        phase: TEST

    }
}
layer {
    name: "segmentation_eval_classIOU"
    type: "SegmentPixelIOU"
    bottom: "result"
    bottom: "label"
    top: "segmentation_eval"
    include {
        phase: TEST
    }
}
```

コンフィギュレーション ファイルを作成する

`config.prototxt` という名前のファイルを作成します。

```
workspace: "./workspace/segmentation/pruning"

gpu: "0,1,2,3"
#test_iter = validation_data_number/val_batch_size e.g. 500/4
test_iter: 125
acc_name: "segmentation_eval_classIOU"
eval_type: "segmentation"

# dataset classes number #e.g. cityscapes: 19
classiou_class_num: 19

model: "./workspace/segmentation/trainval.prototxt"
weights: "./workspace/segmentation/snapshots/_iter_200000.caffemodel"
solver: "./workspace/segmentation/solver.prototxt"

rate: 0.1

pruner {
    method: REGULAR

    exclude {
    layer_top: "pred"
    }
}
```

モデル解析を実行する

```
$ ./vai_p_caffe ana -config config.prototxt
```

モデルをプルーニングする

```
$ ./vai_p_caffe prune -config config.prototxt
```

プルーニング済みモデルを微調整する

微調整の初期パラメーターとして、次のソルバー設定を使用できます。

```
net: "./workspace/segmentation/trainval.prototxt"
test_iter: 125
test_interval: 5000
test_initialization: true
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "multistep"
gamma: 0.1
stepvalue: 75000
stepvalue: 85000
display: 10
max_iter: 200000
snapshot: 5000
snapshot_prefix: "./workspace/segmentation/snapshots/"
solver_mode: GPU
iter_size: 1
average_loss: 20
eval_type: "segmentation"
classiou_class_num: 19
```

次のコマンドを使用して微調整を開始します。

```
$ ./vai_p_caffe finetune -config config.prototxt
```

推定所要時間: Cityscapes の学習用データセットを使用した場合、270 エポックで約 40 時間 (2975 画像、4 x NVIDIA Tesla V100)。

最終出力を生成する

モデルを完成させるには、次を実行します。

```
$ ./vai_p_caffe transform -model baseline.prototxt -weights
finetuned_model.caffemodel -output
final.caffemodel
```

プルーニングの結果

- データセット: Cityscapes
- 入力サイズ: 2048 x 1024
- GPU プラットフォーム: 4 x NVIDIA Tesla V100
- FLOPs: 136G

表 8: **XFPN のプルーニング結果**

丸め処理	FLOPs	mIOU
0	100%	71.25
1	90%	70.88

表 8: **XFPN のプルーニング結果** (続き)

丸め処理	FLOPs	mIOU
2	83%	69.94

Darknet の例

一般に、YOLOov2 と YOLOv3 のプルーニングには Darknet を使用します。YOLOv3 の例は、Darknet バージョン - vai_p_darknet で紹介しています。この例を参照して、YOLOv3 ネットワークをプルーニングしてください。

付録 A

その他のリソースおよび法的通知

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、ザイリンクス サポート サイトを参照してください。

Documentation Navigator およびデザイン ハブ

ザイリンクス Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado® IDE で [Help]→[Documentation and Tutorials] をクリックします。
- Windows で [スタート]→[すべてのプログラム]→[Xilinx Design Tools]→[DocNav] をクリックします。
- Linux コマンド プロンプトに「`docnav`」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hub View] タブをクリックします。
- ザイリンクス ウェブサイトでデザイン ハブ ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの Documentation Navigator ページを参照してください。DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

参考資料

次の文書は、このユーザー ガイドの補足資料として役立ちます。日本語版のバージョンは、英語版より古い場合があります。

1. 『Vitis AI ユーザー ガイド』 (UG1414)

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社 (本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ) に開示される情報 (以下「本情報」といいます) は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず (商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない (否認する) ものとします。また、(2) ザイリンクスは、本情報 (貴殿または貴社による本情報の使用を含む) に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない (契約上、不法行為上 (過失の場合を含む)、その他のいかなる責任の法理によるかを問わない) ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害 (第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます) が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、https://japan.xilinx.com/legal.htm#tos で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。https://japan.xilinx.com/legal.htm#tos で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモーティブ製品 (製品番号に「XA」が含まれる) は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能 (「セーフティ 設計」) がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション (「セーフティ アプリケーション」) における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとします。セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品の責任の制限を規定する適用法令および規則にのみ従うものとします。

著作権

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。