Vitis AI Optimizer User Guide

UG1333 (v2.5) June 15, 2022

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

AMD
XILINX

AMD
XILINX

Table of Contents

Chapter 1

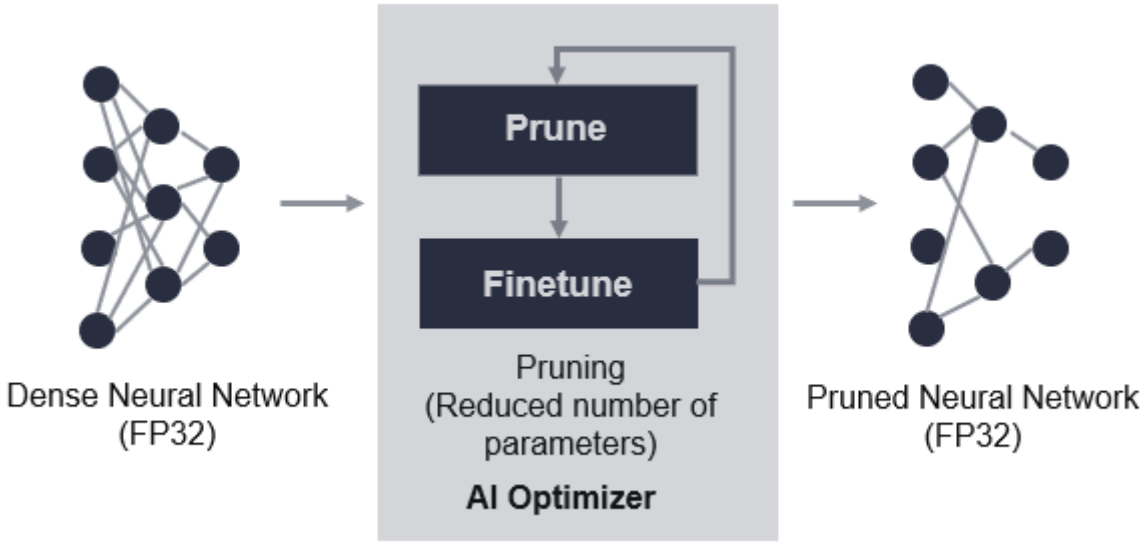
Overview and Installation

Vitis AI Optimizer Overview

Vitis™ AI is a Xilinx® development kit for AI inference on Xilinx hardware platforms. Inference in machine learning is computation-intensive and requires high memory bandwidth to meet the low-latency and high-throughput requirements of various applications.

The Vitis AI optimizer provides the ability to optimize neural network models. Currently, the Vitis AI optimizer includes only one tool called the pruner. The Vitis AI pruner prunes redundant kernels in neural networks thereby reducing the overall computational cost for inference. The pruned models produced by the Vitis AI pruner are then quantized by the Vitis AI quantizer and deployed to a Xilinx FPGA, SoC, or ACAP devices. For more information on the Vitis AI quantizer and deployment, see the *Vitis AI User Guide* (UG1414).

Figure 1: **VAI Optimizer**



The Vitis AI pruner supports four deep learning frameworks. The frameworks and their corresponding tool names (_p_ denotes pruning) are listed in the following table:

Table 1: **Vitis AI Pruner Frameworks and Tool Names**

Framework	Tool Name
TensorFlow	vai_p_tensorflow (TF1.15), vai_p_tensorflow2 (TF2.x)
PyTorch	vai_p_pytorch

The Vitis AI optimizer requires a commercial license. Contact xilinx_ai_optimizer@xilinx.com to obtain access to the Vitis AI optimizer installation package and license.

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal® ACAP design process Design Hubs and the Design Flow Assistant materials can be found on the Xilinx.com website. This document covers the following design processes:

- **Machine Learning and Data Science:** Importing a machine learning model from a Caffe, Pytorch, TensorFlow, or other popular framework onto Vitis™ AI, and then optimizing and evaluating its effectiveness. Topics in this document that apply to this design process include:
  - Chapter 2: Pruning
  - Chapter 3: Working with the Vitis AI Pruner
  - Chapter 4: Example Networks

Installation

Vitis AI provides a Docker environment for the Vitis AI Optimizer. To build and run the Docker image, refer to the "GPU docker" section in the https://github.com/Xilinx/Vitis-AI#installation documentation. The following conda environments are available in the Docker image:

- vitis-ai-optimizer_pytorch
- vitis-ai-optimizer_tensorflow
- vitis-ai-optimizer_tensorflow2

The Docker encapsulates the required tools and libraries necessary for pruning in these frameworks. If you have a license, you can run the Vitis AI Optimizer directly in the Docker.

Minimum System Requirements

Table 2: **Minimum System Requirements**

Component	Requirement
Operating System	• Ubuntu 18.04, 20.04 (Linux kernel <=5.8) • CentOS 7.8, 7.9, 8.1, 8.2 • RHEL 8.3, 8.4
CPU	x86-64
GPU	NVIDIA GPU supports CUDA® 10.2 or higher, such as NVIDIA P100, V100, A100
CUDA Driver	Driver compatible to CUDA version, NVIDIA-384 or higher for CUDA10.2, NVIDIA-450 or higher for CUDA 11.0
Docker version	19.03 or higher

***Note*:** Only vai_p_pytorch and vai_p_tensorflow2 tools support the NVIDIA A100 Tensor Core GPU.

Supported Frameworks and Features

The following tables shows the features that are supported by the Vitis AI pruner for different frameworks:

Table 3: **Frameworks and Features Supported by Vitis AI Pruner**

Framework	Versions	Features		
		Iterative	One-step	OFA
PyTorch	Supports 1.4 - 1.10	Yes	Yes	Yes
TensorFlow 1.15	Supports 1.15	Yes	No	No
TensorFlow 2.x	Supports 2.3 - 2.8.0	Yes	No	No

***Note*:** Vitis AI 2.5 and later releases do not support Caffe and Darknet. To use the Vitis AI Pruner on these two frameworks, use an earlier version of Vitis AI.

Vitis AI Pruner License

If you have previously purchased a license, you can generate the required license file in your Xilinx licensing account. The Vitis AI pruner finds the license using an environment variable called `XILINXD_LICENSE_FILE`. There are two types of license:

- **Floating license:** To install, specify the path in the `port@hostname` format. For example, `export XILINXD_LICENSE_FILE=2001@xcolicsvr1`.

Further details related to license server installation and usage can be found in the Xilinx Licensing Solution Center.

Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973), while intended for Vivado installations, provides additional detailed instructions that are also relevant for installation of the Vitis AI Optimizer license.

- **Node-locked License:** To install, specify a particular license file or directory where all the `.lic` files are located.

  To specify a particular file:

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

  To specify a directory:

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

  If you have multiple licenses, specify them at the same time separated by a colon:

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

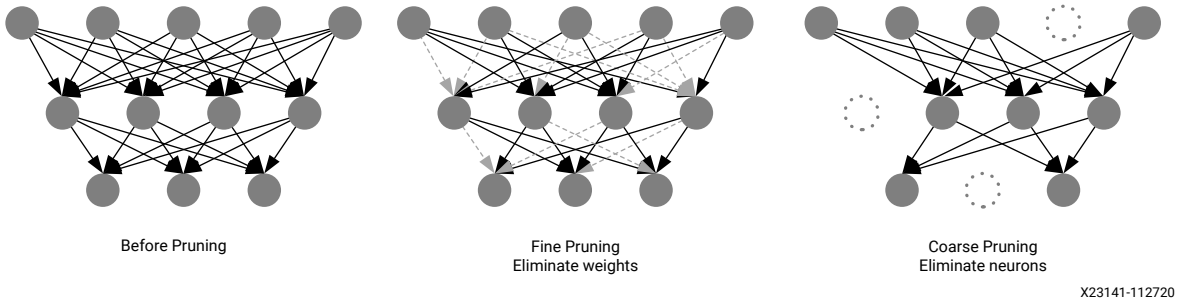
  A node-locked license can also be installed by copying it to `$HOME/.Xilinx` directory.

Pruning

Overview

Neural networks are typically over-parameterized with significant redundancy. Pruning is the process of eliminating redundant weights while keeping the accuracy loss as low as possible.

Figure 2: **Coarse Pruning and Fine Pruning**



Industry research has led to several techniques that serve to reduce the computational cost of neural networks for inference. These techniques include:

- Fine-grained pruning
- Coarse-grained pruning
- Neural Architecture Search (NAS)

The simplest form of pruning is called fine-grained pruning and results in sparse matrices (i.e., matrices which have many elements equal to zero), which requires the addition of specialized hardware and techniques for weight skipping and compression. Xilinx does not currently implement fine-grained pruning.

The Vitis™ AI pruner employs coarse-grained pruning, which eliminates neurons that do not contribute significantly to the accuracy of the network. For convolutional layers, the coarse-grained method prunes the entire 3D kernel and hence is also known as channel pruning. Inference acceleration can be achieved without specialized hardware for coarse-grained pruned models. Pruning always reduces the accuracy of the original model. Retraining (fine-tuning) adjusts the remaining weights to recover accuracy.

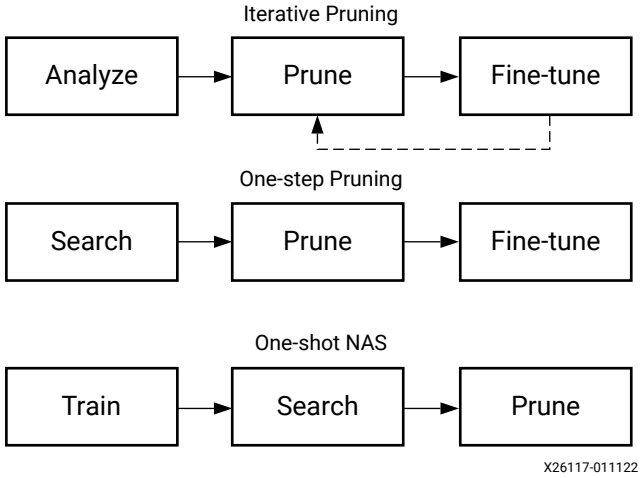
Coarse-grained pruning works well on large models with common convolutions, for example, ResNet and VGGNet, but when it comes to depthwise convolution based models such as MobileNet-v2, the accuracy of the pruned model drops dramatically even at a small pruning rate.

In addition to pruning, the Vitis AI provides a one-shot neural architecture search (NAS) based approach to reduce the computational cost of inference. This method requires a four-step process:

1. Train
2. Search
3. Prune
4. Fine-tune (optional)

Compared with coarse-grained pruning, one-shot NAS implementations assemble multiple candidate "subnetworks" into a single, over-parameterized graph known as a Supernet. The training optimization algorithm attempts to optimize all candidate networks simultaneously using supervised learning. Upon the completion of this training process, candidate subnetworks are ranked based on computational cost and accuracy. The developer selects the best candidate to meet their requirements. The one-shot NAS method is effective in compressing models that implement both depthwise convolutions and conventional convolutions but requires a long training time and a higher level of skill on the part of the developer.

Figure 3: **Workflow of Three Pruning Methods**



Coarse-grained Pruning

The Vitis AI pruner implements two methods of coarse-grained pruning. Depending on the workflow, the methods can be classified as iterative and one-step. The one-step method is more advanced and is preferred.

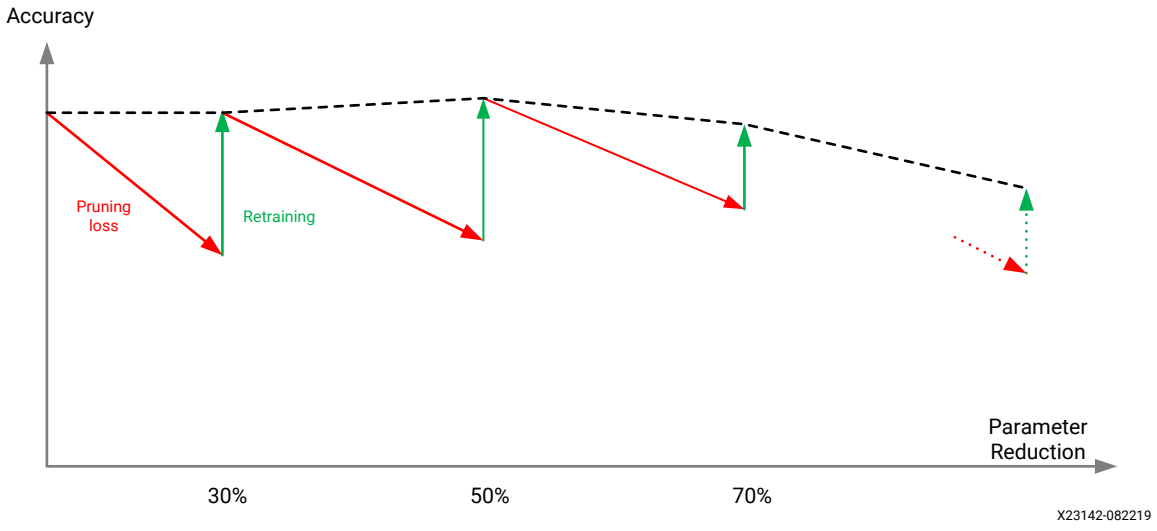
Iterative Pruning

The pruner is designed to reduce the number of model parameters while minimizing the accuracy loss. This is done iteratively as shown in the following figure. Pruning results in accuracy loss while retraining recovers accuracy. Pruning, followed by retraining, forms one iteration. In the first iteration of pruning, the input model is the baseline model, and it is pruned and fine-tuned. In subsequent iterations, the fine-tuned model obtained from the previous iterations becomes the new baseline. This process is usually repeated several times until a desired sparse model is obtained. The iterative approach is required because a model cannot be pruned in a single pass while maintaining accuracy. If in a single pass too many parameters are removed, the accuracy loss becomes a step function. Such loss is challenging to recover.

IMPORTANT! *The reduction parameter is gradually increased in every iteration to improve accuracy during the fine-tuning stage.*

Leveraging the process of iterative pruning, higher pruning rates can be achieved without any significant loss of model performance.

Figure 4: **Iterative Pruning**



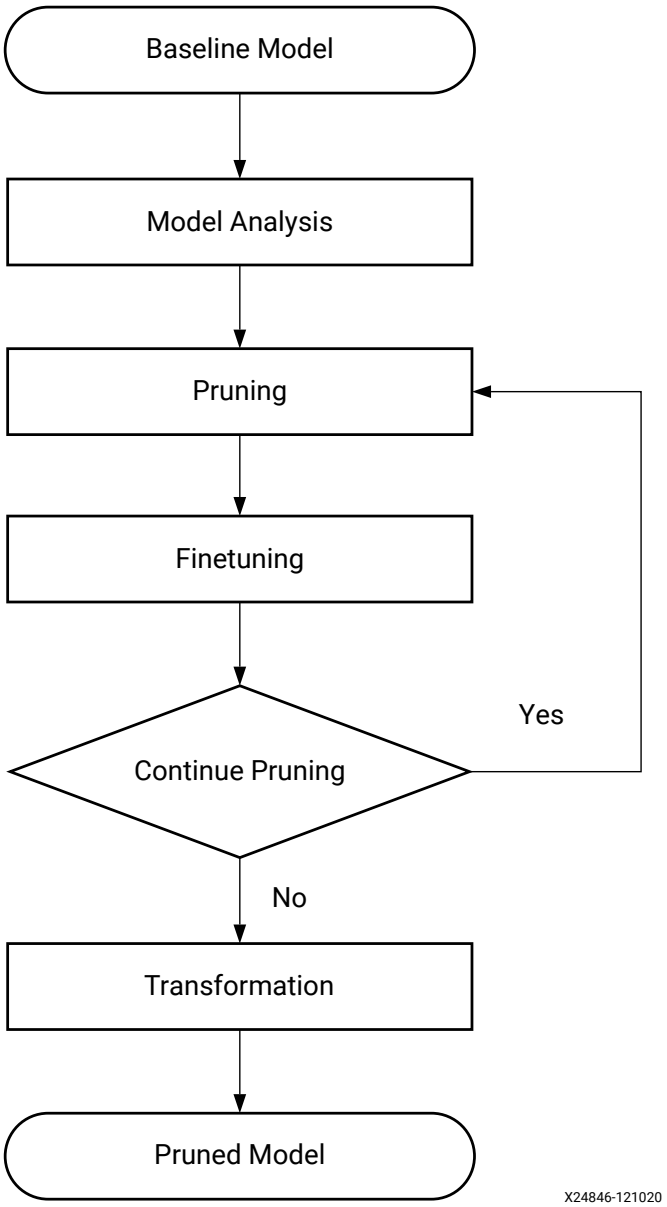
The four primary stages in iterative pruning are as follows:

- **Analysis:** Perform a sensitivity analysis on the model to determine the optimal pruning strategy.
- **Pruning:** Reduce the number of computations in the input model.
- **Fine-tuning:** Retrain the pruned model to recover accuracy.
- **Transformation:** Generate a dense model with reduced weights.

Follow these steps to prune a model. The steps are also shown in the following figure.

1. Analyze the original baseline model.
2. Prune the model.
3. Fine-tune the pruned model.
4. Repeat steps 2 and 3 several times.
5. Transform the pruned sparse model to a final dense encrypted model to be used with the Vitis AI Quantizer.

Figure 5: **Iterative Pruning Workflow**



Guidelines for Better Pruning Results

The following is a list of suggestions to optimize pruning results. Following these guidelines has been found to help developers achieve higher pruning ratios and reduced accuracy loss.

- Use as much data as possible to perform a model analysis. Ideally, you should use all the data in the validation dataset but this can be time consuming. You can also use partial validation set data, but you need to make sure at least half of the data set is used.

- During the fine-tuning stage, experiment with a few parameters, including the initial learning rate, the learning rate decay policy and use the best result as the input to the next round of pruning.
- The data used in fine-tuning should be a subset of the original dataset used to train the baseline model.
- If the accuracy does not improve sufficiently after conducting several fine-tuning experiments, try reducing the pruning rate parameter and then re-run pruning and fine-tuning.

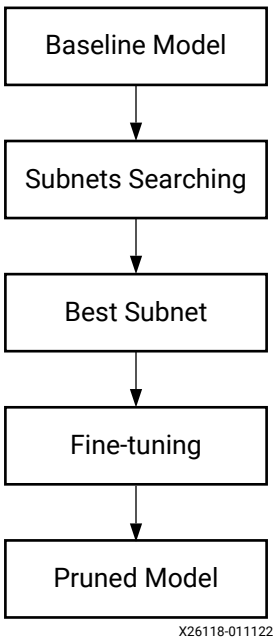
One-step Pruning

One-step pruning implements the EagleEye[1] algorithm. It introduces a strong positive correlation between different pruned models and their corresponding fine-tuned accuracy by a simple yet efficient evaluation component, that is the adaptive batch normalization. It enables you to get the subnetwork with the highest potential accuracy without actually fine-tuning the models. In short, the one-step pruning method searches for a bunch of subnetworks (i.e., generated pruned models) that meet the required model size, and applies an evaluation for selecting the most promising one from them. The selected subnetwork is then retrained to recover the accuracy.

The pruning steps are as follows:

1. Search for subnetworks that meet the required pruning ratio.
2. Select a potential network from a bunch of subnetworks with an evaluation component.
3. Fine-tune the pruned model.

Figure 6: **One-step Pruning Workflow**



Note:

1. Bailin Li et al., EagleEye: Fast Sub-net Evaluation for Efficient Neural Network Pruning, arXiv:2007.02491

Comparing Iterative Pruning and One-Step Pruning

A comparison of these two methods is shown in the following table.

Table 4: **Iterative vs. One-step Pruning**

Criteria	Iterative Pruning	One-step Pruning
Prerequisites	-	BatchNormalization in network
Time taken	More than one-step pruning	Less than iterative pruning
Retraining requirement	Required	Required
Code organization	Evaluation function	Evaluation function Calibration function

Once-for-All (OFA)

Once-For-All (OFA)[1] is a compression scheme based on One-Shot NAS. You often perform tasks such as compressing a trained model and deploying it on one or more devices. Conventional schemes require you to repeat the network design process and retrain the designed network from scratch for each device, which is computationally prohibitive.

OFA introduces a new solution to tackle this challenge: designing a once-for-all network that can be directly deployed under diverse architectural configurations. Therefore, training cost can be amortized. The inference is performed by selecting only a part of the once-for-all network.

OFA reduces the model size across many more dimensions than pruning. It builds a family of models of varying depth, width, kernel size, and image resolution and jointly optimizes all the candidate models. Once trained, the best accuracy-flops trade-off subnetwork can be discovered by evolutionary search.

For each layer in the original model, Vitis OFA allows you to use an arbitrary pruning ratio for channels and arbitrary kernel sizes. The original model is split into many child networks with shared weights and becomes a super network. The child networks can do forward passes and update using a part of the convolution weights. All subnetworks should be jointly optimized during training. When all child networks are trained well, you can search for the best FLOPs-accuracy trade-offs subnetwork from the supernetwork.

To get the most extreme compression effect, the Vitis OFA pruner can optimize the search space based on the ratio of the number of depthwise convolutions to the number of regular convolutions. If the number of depthwise convolutions is more than the number of regular convolutions, the Vitis OFA pruner mainly compresses the kernel size > 1 (kernel size is supported for both odd and even numbers) convolutions. In the other case, the Vitis OFA pruner compresses all convolutions (kernel size >= 1), it would reduce the search space of 1x1 convolutions to avoid the channel width of supernetwork is too small, which affects the accuracy of supernetwork.

OFA uses the original model as a teacher model to guide the training of subnetworks. Knowledge distillation allows the output of teacher models to be used as softened labels to provide more information about intraclasses and interclasses. The Vitis OFA uses adaptive soft KDLoss[2] and the sandwich rule[3] to improve performance and efficiency. Compared with OFA, the Vitis OFA reduces the training time by half.

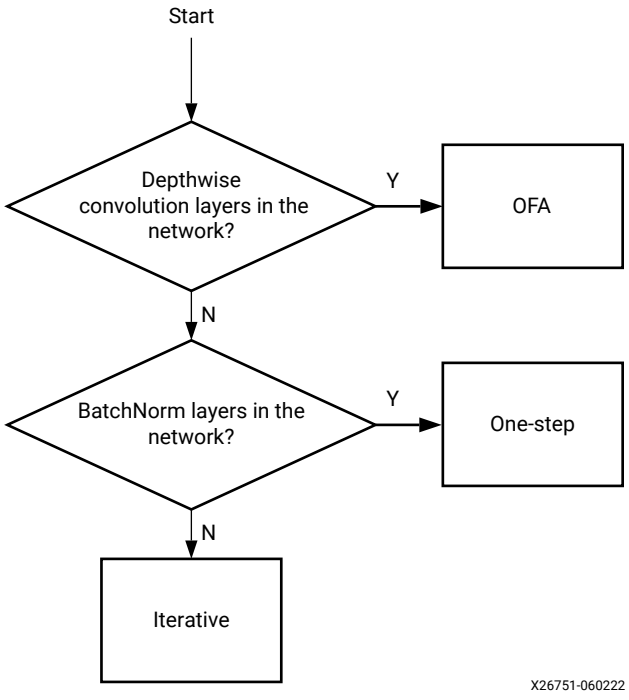
Note:

1. Han Cai et al., Once-for-All: Train One Network and Specialize it for Efficient Deployment, arXiv:1908.09791
2. Dilin Wang et al., AlphaNet: Improved Training of Supernets with Alpha-Divergence, arXiv:2102.07954
3. Jiahui Yu et al., BigNAS: Scaling Up Neural Architecture Search with Big Single-Stage Models, arXiv:2003.11142

Guidelines for Selecting Pruning Methods

There are three pruning methods available in Vitis AI Optimizer for PyTorch. Refer to the following decision tree to choose the right method for your network.

Figure 7: **Flowchart for Selecting Pruning Methods in PyTorch**



Working with the Vitis AI Pruner

TensorFlow (1.15) Version - vai_p_tensorflow

Exporting an Inference Graph

TensorFlow Model

First, build a TensorFlow graph for training and evaluation. Each part must be written in a separate script. If you have trained a baseline model before and you have the training codes, then you only need to prepare the codes for evaluation.

The evaluation script must contain a function named `model_fn` that creates all the needed nodes from input to output. The function should return a dictionary that maps the names of output nodes to their operations or a `tf.estimator.Estimator`. For example, if your network is an image classifier, the returned dictionary usually includes operations to calculate top-1 and top-5 accuracy as shown in the following snippet:

```
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)
  }
```

Or, if you use the TensorFlow Estimator API to train and evaluate your network, your `model_fn` function must return an instance of the `tf.estimator`. At the same time, you must also provide a function called `eval_input_fn`, which the Estimator uses to get the data used in the evaluation.

```
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)
```

Evaluation codes are used to export an inference `GraphDef` file and evaluate network performance during pruning.

To export a `GraphDef` proto file, use the following code:

```
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))
```

Keras Model

For the Keras model, there is no explicit graph definition. Get a `GraphDef` object first and then export it. An example of `tf.keras` predefined ResNet50 is given here:

```
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.python.framework import graph_util

tf.keras.backend.set_learning_phase(0)
model = tf.keras.applications.ResNet50(weights=None,
    include_top=True,
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000)
model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy())
graph_def = K.get_session().graph.as_graph_def()

# "probs/Softmax": Output node of ResNet50 graph.
graph_def = graph_util.extract_sub_graph(graph_def, ["probs/Softmax"])
tf.train.write_graph(graph_def,
    "./",
    "inference_graph.pbtxt",
    as_text=True)
```

Preparing a Baseline Model

TensorFlow Model

TensorFlow saves variables in binary checkpoint files that map variable names to tensor values. `vai_p_tensorflow` takes a checkpoint file as input to load trained weights. `tf.train.Saver` provides methods to specify paths for the checkpoint files to write to or read from.

The following code snippet calls the `tf.train.Saver.save` method to save variables to checkpoint files:

```
with tf.Session() as sess:
# your graph building codes here
# ……
sess.run(train_op)

# Save the variables to disk.
save_path = saver.save(sess, "/tmp/model.ckpt")
print("Model saved in path: %s" % save_path)
```

The saved checkpoint files look like this:

```
model.ckpt.data-00000-of-00001
model.ckpt.index
model.ckpt.meta
```

Keras Model

`tf.keras` allows model weights to be saved in two formats: HDF5 and TensorFlow. Only the TensorFlow format is currently supported by the tool. If the model weights have been saved in the HDF5 format, convert it to the TensorFlow format before proceeding.

```
import tensorflow as tf
tf.keras.backend.set_learning_phase(0)
model = tf.keras.applications.ResNet50(weights="imagenet",
    include_top=True,
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000)
model.save_weights("model.ckpt", save_format='tf')
```

The converted checkpoint files look like this:

```
model.ckpt.data-00000-of-00001
model.ckpt.data-00001-of-00002
model.ckpt.index
```

Performing Model Analysis

Analyze the model before pruning it to find a suitable pruning strategy to prune the model.

To run model analysis, provide a Python script containing the functions that evaluate model performance. Assuming that your script is `eval_model.py`, provide the required functions in one of three ways:

- A function named `model_fn()` that returns a Python dictionary of metric ops:

```
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
```

- A function named `model_fn()` that returns an instance of `tf.estimator.Estimator` and a function named `eval_input_fn()` that feeds test data to the estimator:

```
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)
```

- A function named `evaluate()` that takes a single parameter as argument that returns the metric score:

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

If you are using the tf.keras API, this is the recommended way:

```
import tensorflow as tf

def evaluate(checkpoint_path):
net = tf.keras.applications.ResNet50(weights=None,
include_top=True,
input_tensor=None,
input_shape=None,
pooling=None,
classes=1000)
```

```
net.load_weights(checkpoint_path)
metric_top_5 = tf.keras.metrics.SparseTopKCategoricalAccuracy()
accuracy = tf.keras.metrics.SparseCategoricalAccuracy()
loss = tf.keras.losses.SparseCategoricalCrossentropy()

# eval_data: validation dataset. You can refer to 'tf.keras.Model.evaluate'
method to generate your validation dataset.
# EVAL_NUM: the number of validation dataset
res = net.evaluate(eval_data,
steps=EVAL_NUM/batch_size,
workers=16,
verbose=1)
eval_metric_ops = {'Recall_5': res[-1]}
return eval_metric_ops
```

If you use the first way to write the script, see the following snippet for calling `vai_p_tensorflow` to perform model analysis. The results of the analysis is saved in a file named `.ana` in the `workspace` directory. Do not move or delete this file because the pruner will load this file for pruning.

```
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"
```

Following are the arguments in this command. See vai_p_tensorflow Usage for a full list of options.

- **--action:** The action to perform.
- **--input_graph:** A `GraphDef` proto file that represents the inference graph of the network.
- **--input_ckpt:** The path to a checkpoint to use for pruning.
- **--eval_fn_path:** The path to a Python script defining an evaluation graph.
- **--target:** The target score that evaluates the performance of the network. If there are more than one score in the network, choose the one that is most important.
- **--max_num_batches:** The number of batches to run in the evaluation phase. This parameter affects the time taken to analyze the model. The larger this value, the more time required for the analysis and the more accurate the analysis is. The maximum value of this parameter is the size of the validation set or the batch_size, that is, all the data in the validation set is used for evaluation.
- **--workspace:** Directory for saving output files.
- **--exclude:** Convolution nodes excluded from pruning.

- **--output_nodes:** Output nodes of the inference graph.

Starting Pruning Loop

Once the `ana` command has executed successfully, you can start pruning the model. The command `prune` is very similar to command `ana`, requiring the same configuration file:

```
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"
```

There is one new argument in this command:

- **--sparsity:** The sparsity of network after pruning. It has a value between 0 and 1. The larger the value, the sparser the model is after pruning.

After the `prune` command executes, the `vai_p_tensorflow` outputs FLOPs of network before and after pruning.

Fine-tuning the Pruned Model

The performance of the pruned model will decrease and you need to fine-tune it to improve its performance. Fine-tuning a pruned model is basically the same as training model from scratch, except that the hype-parameters, such as the initial learning rate and the learning rate decay type, are different.

Pruning and fine-tuning comprises one iteration step. To achieve a higher pruning rate without significant loss of performance, prune the model several times. After every iteration of "prune-finetune", make the following changes to the commands before you run the next pruning:

1. Modify the `--input_ckpt` flag to a checkpoint file generated in previous fine-tuning process.
2. Increase the value of the `--sparsity` flag in the next iteration.

Generating Dense Checkpoints

A few iterations of pruning generates a model that is smaller than its original size. To get a final model, perform a transformation on the model.

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

***Note*:** Transformation is only required after all iterations of pruning are completed. Do not run the `transform` command after each iteration of pruning.

Freezing the Graph

Now, you have a `GraphDef` file containing the architecture of the pruned model and a checkpoint file saving trained weights. For prediction or quantization, merge these two files into a single pb file.

Freeze the graph using the following command:

```
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"
```

After completing all the previous steps, you should get the final output file, `frozen.pb`, of the pruning. Use this file for prediction or quantization. To get the FLOPs of the frozen graph, run the following command:

```
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 Usage

The following arguments are available when running vai_p_tensorflow:

Table 5: **vai_p_tensorflow Arguments**

Argument	Type	Action	Default	Description
action	string	-	""	Which action to run. Valid actions include 'ana', 'prune', 'transform', and 'flops'.
workspace	string	['ana', 'prune']	""	Directory for saving output files.
input_graph	string	['ana', 'prune', 'flops']	""	Path of a GraphDef protobuf file that defines the network's architecture.
input_ckpt	string	['ana', 'prune', 'transform']	""	Path of a checkpoint file. It is the prefix of filenames created for the checkpoint.

Table 5: **vai_p_tensorflow Arguments** *(cont'd)*

Argument	Type	Action	Default	Description
eval_fn_path	string	['ana']	""	A Python file path used for model evaluation.
target	string	['ana']	""	The output node name that indicates the performance of the model.
max_num_batches	int	['ana']	None	Maximum number of batches to evaluate. By default, use all.
output_graph	string	['prune']	""	Path of a GraphDef protobuf file for saving the pruned network.
output_ckpt	string	['prune', 'transform'']	""	Path of a checkpoint file for saving weights.
gpu	string	['ana']	""	GPU device IDs to use separated by ','.
sparsity	float	['prune']	None	The desired sparsity of network after pruning.
exclude	repeated	['ana', 'prune']	None	Convolution nodes excluded from pruning.
input_nodes	repeated	['flops']	None	Input nodes of the inference graph.
input_node_shapes	repeated	['flops']	None	Shape of input nodes.
output_nodes	repeated	['ana', 'prune', 'flops']	None	Output nodes of the inference graph.
channel_batch	int	['prune']	2	The number of output channels is a multiple of this value after pruning.

TensorFlow (2.x) Version – vai_p_tensorflow2

vai_p_tensorflow2 only supports Keras models created by the Functional API or the Sequential API. Subclassed models are not supported.

Creating a Model

Here, a simple MNIST convnet from the Keras vision example is used.

```
model = keras.Sequential([
    keras.Input(shape=input_shape),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(), layers.Dropout(0.5),
    layers.Dense(num_classes, activation="softmax"),
])
```

Creating a Pruning Runner

To create an input specification with shape and dtype and to use this specification to get a pruning runner, use the following command:

```
from tf_nndct.optimization import IterativePruningRunner

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
```

Pruning a Model

To prune a model, follow these steps:

1. Define a function to evaluate model performance. The function must satisfy two requirements:
   - The first argument must be an `keras.Model` instance to be evaluated
   - Returns a Python number to indicate the performance of the model

   ```
   def evaluate(model):
     model.compile(loss="categorical_crossentropy", optimizer="adam",
   metrics=["accuracy"])
     score = model.evaluate(x_test, y_test, verbose=0)
     return score[1]
   ```

2. Use this evaluation function to run model analysis:

   ```
   runner.ana(evaluate)
   ```

3. Determine a pruning ratio. The ratio indicates the reduction in the amount of floating-point computation of the model in forward pass.

   $$[\text{MACs of pruned model}] = (1 - \text{ratio}) * [\text{MACs of original model}]$$

   The value of ratio should be in (0, 1):

   ```
   sparse_model = runner.prune(ratio=0.2)
   ```

   Note: `ratio` is only an approximate target value and the actual pruning ratio may not be exactly equal to this value.

The returned model from `prune()` is sparse which means the pruned channels are set to zeros and model size remains unchanged. The sparse model has been used in the iterative pruning process. The sparse model is converted to a pruned dense model only after pruning is completed.

Besides returning a sparse model, the pruning runner generates a specification file in the `.vai` directory that describes how each layer will be pruned.

Fine-tuning a Sparse Model

Training a sparse model is no different from training a normal model. The model will maintain sparsity internally. There is no need for any additional actions other than adjusting the hyper-parameters.

```
sparse_model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])
sparse_model.fit(x_train, y_train, batch_size=128, epochs=15,
validation_split=0.1)
sparse_model.save_weights("model_sparse_0.2", save_format="tf")
```

Note: When calling `save_weights`, use the "tf" format to save the weights.

Performing Iterative Pruning

Load the checkpoint saved from the previous fine-tuning stage to the model. Increase the ratio value to get a sparser model. Then continue to fine-tune this sparse model. Repeat this pruning and fine-tuning loop a couple of times until the sparsity reaches the desired value.

```
model.load_weights("model_sparse_0.2")

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
sparse_model = runner.prune(ratio=0.5)
```

Getting the Pruned Model

When the iterative pruning is completed, a sparse model is generated which has the same number of parameters as the original model but with many of them now set to zero.

Call `get_slim_model()` to remove zeroed parameters and from the sparse model and get a truly pruned model:

```
model.load_weights("model_sparse_0.5")

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
runner.get_slim_model()
```

By default, the runner uses the latest pruning specification to generate the slim model. You can see what the latest specification file is with the following command:

```
$ cat .vai/latest_spec
$ ".vai/mnist_ratio_0.5.spec"
```

If this file does not match your sparse model, you can explicitly specify the file path to be used:

```
runner.get_slim_model(".vai/mnist_ratio_0.5.spec")
```

vai_p_tensorflow2 APIs

tf_nndct.IterativePruningRunner

Runner for structured pruning of Keras model in an iterative way. This API has the following methods:

- `__init__(model, input_specs)`

  Creates a new `IterativePruningRunner` object.

  - **model:** A baseline model to prune. The mode should be an instance of `keras.Model`.
  - **input_specs:** A single or a list of `tf.TensorSpec` used to represent model input specifications.

- `ana(eval_fn, excludes=None, forced=False)`

  Performs model analysis. The analysis result will be saved in `'.vai'` directory and this cached result will be used directly in subsequent calls unless `forced` is set to True.

  - **eval_fn:** Callable object that takes a `keras.Model` object as its first argument and returns the evaluation score.
  - **excludes:** A list of layer name or layer instance to be excluded from pruning.
  - **forced:** When set to True, run model analysis is run instead of using the cached analysis result.

- `prune(ratio=None, threshold=None, spec_path=None, excludes=None, mode='sparse')`

  Prune the baseline model and returns a sparse model. The degree of model reduction can be specified in three ways: ratio, threshold, or pruning specification. The first method is the preferred method; the latter two are more suitable for experiments with manual tuning.

  - **ratio:** The expected percentage of MACs reduction of baseline model. This is a guidance value and the actual MACs reduction may not strictly be equal to this value.
  - **threshold:** Relative proportion of model performance loss between the baseline model and the pruned model.
  - **spec_path:** Pruning specification path used to prune the model.
  - **excludes:** A list of layer name or layer instance to be excluded from pruning.
  - **mode:** The mode in which the baseline model is pruned to return a sparse model.

- `get_slim_model(spec_path=None)`

Get a slim model from a sparse model. Use the latest pruning specification to do this transformation by default. If the sparse model was not generated from the latest specification, a specification path can be provided explicitly.

- **spec_path:** Path of pruning specification used to transform a sparse model to a slim model.

PyTorch Version - vai_p_pytorch

The pruning tool on PyTorch is a Python package and not an executable program. vai_p_pytorch provides three methods of model pruning:

- Iterative Pruning
- One-step Pruning
- Once-for-All (OFA)

Iterative and one-step pruning are suitable for networks with common convolution layers, but do not work very well on depthwise convolution based networks like MobileNet-v2. Convolutional neural networks (CNN) generally contain BatchNormalization layers and one-step pruning is preferred for these networks because it is faster and works better. If there are no BatchNormalization layers in the network, such as in VGGNet, then iterative pruning should be used.

OFA is applicable to both depthwise and common convolutions. It is important to know that OFA is theoretically the best of these three methods though it is not easy to get good pruning results. The result depends on how well the supernetwork can be trained; a long training time and strong training skills are required.

To summarize, if there are BatchNormalization layers in the network, use one-step pruning. Otherwise, use iterative pruning. If you are not satisfied with the pruning results, OFA can serve as an alternative approach.

Coarse-grained Pruning

The steps for pruning a model according to this pruning method are as follows:

Creating a Model

For simplicity, ResNet18 from `torchvision` is used here. In real life applications, the process of creating a model can be complicated.

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

Creating a Pruning Runner

Import modules and prepare input signature:

```
from pytorch_nndct import get_pruning_runner

# The input signature should have the same shape and dtype as the model
input.
input_signature = torch.randn([1, 3, 224, 224], dtype=torch.float32)
```

Create an iterative pruning runner:

```
runner = get_pruning_runner(model, input_signature, 'iterative')
```

Or, a one-step pruning runner:

```
runner = get_pruning_runner(model, input_signature, 'one_step')
```

Pruning a Model

Iterative Pruning

The method includes two stages: model analysis and pruned model generation. After the model analysis is completed, analysis result is saved in the file named `.vai/xxx.sens`. You can prune a model iteratively using this file. In other words, you should prune the model to the target sparsity gradually to avoid the failure to improve the model performance in the retraining stage that is caused by setting a too high pruning ratio.

1. Define an evaluation function. The function must take a model as its first argument and return a score.

   ```
   def eval_fn(model, dataloader):
     top1 = AverageMeter('Acc@1', ':6.2f')
     model.eval()
     with torch.no_grad():
     for i, (images, targets) in enumerate(dataloader):
       images = images.cuda()
       targets = targets.cuda()
       outputs = model(images)
       acc1, _ = accuracy(outputs, targets, topk=(1, 5))
       top1.update(acc1[0], images.size(0))
     return top1.avg
   ```

2. Run model analysis and get a pruned model.

   ```
   runner.ana(eval_fn, args=(val_loader,))

   model = pruning_runner.prune(removal_ratio=0.2)
   ```

Run analysis only once for the same model. You can prune the model iteratively without re-running analysis because there is only one pruned model generated for a specific pruning ratio. The subnetwork obtained by pruning may not be very good because an approximate algorithm is used to generate this unique pruned model according to the analysis result. The one-step pruning method can generate a better subnetwork.

One-step Pruning

The method also include two stages: adaptive-BN-based searching for pruning strategy and pruned model generation. After searching, a file named `.vai/xxx.search` is generated in which the search result (pruning strategies and corresponding evaluation scores) is stored. You can get the final pruned model in one-step.

`num_subnet` provides the number of candidate subnetworks satisfying the sparsity requirement to be searched. The best subnetwork can be selected from these candidates. The higher the value, the longer it takes to search, but the higher the probability of finding a better subnetwork.

```
# Adaptive-BN-based searching for pruning strategy. 'calibration_fn' is a
function for calibrating BN layer's statistics.
runner.search(gpus=['0'], calibration_fn=calibration_fn,
calib_args=(val_loader,), eval_fn=eval_fn, eval_args=(val_loader,),
num_subnet=1000, removal_ratio=0.7)

model = runner.prune(removal_ratio=0.7, index=None)
```

The `eval_fn` is the same with iterative pruning method. A `calibration_fn` function that implements adaptive-BN is shown in the following example code. You should define your code similarly.

```
def calibration_fn(model, dataloader, number_forward=100):
  model.train()
  with torch.no_grad():
    for index, (images, target) in enumerate(dataloader):
      images = images.cuda()
      model(images)
    if index > number_forward:
      break
```

The one-step pruning method has several advantages over the iterative approach.

- The generated pruned models are more accurate. All subnetworks that meet the requirements are evaluated.
- The workflow is simpler because you can obtain the final pruned model in one step without iterations.
- Retraining a slim model is faster than a sparse model.

There are two disadvantages to one-step pruning: One is that the random generation of pruning strategy is unstable. The other is that the subnetwork searching must be performed once for every pruning ratio.

Retraining a Model

Retraining a model is the same as training a baseline model.

```
optimizer = torch.optim.Adam(model.parameters(), 1e-3, weight_decay=5e-4)
best_acc1 = 0

for epoch in range(args.epoches):
  train(train_loader, model, criterion, optimizer, epoch)
  acc1, acc5 = evaluate(val_loader, model, criterion)

  is_best = acc1 > best_acc1
  best_acc1 = max(acc1, best_acc1)
  if is_best:
    torch.save(model.state_dict(), 'model_pruned.pth')
    # Sparse model has one more special method in iterative pruning.
    if hasattr(model, 'slim_state_dict'):
      torch.save(model.slim_state_dict(), 'model_slim.pth')
```

Generating a Pruned Model

There are two ways to generate a final slim model. Usually the slim model is used for quantizing or evaluating directly.

With Pruning API

```
method = 'iterative' # or 'one_step'
runner = get_pruning_runner(model, input_signature, method)
slim_model = runner.prune(removal_ratio=0.2, mode='slim')
slim_model.load_state_dict(torch.load('model_pruned.pth'))
```

Without Pruning API

```
def slim_model_from_state_dict_and_pruning_info(model, state_dict,
json_path):
  """Modify modules according to pruning information saved in json file and
  load the state dict to the model.


  Args:
  model: An torch.nn.Module instance to load state dict.
  state_dict: A state dict to be loaded.
  json_path: A json file to save pruning information.


  Returns:
  A modified model generated by pruning information saved in json file.


  """
  with open(json_path, 'r') as f:
    pruning_info = json.load(f)
  tensorname_to_nodename = pruning_info['MapOfTensornameToNodename']

  def change_conv_module(module, tensor_name):
    if module.groups == 1:
      if pruning_info[node_name]['in_dim'] != 0:
```

```
        module.in_channels = pruning_info[node_name]['in_dim']
      if pruning_info[node_name]['out_dim'] != 0:
        module.out_channels = pruning_info[node_name]['out_dim']
    else:
      module.groups = pruning_info[node_name]['out_dim']
      if pruning_info[node_name]['in_dim'] != 0:
        module.in_channels = pruning_info[node_name]['in_dim']
      if pruning_info[node_name]['out_dim'] != 0:
        module.out_channels = pruning_info[node_name]['out_dim']

  def change_tensor(tensor_name):
    node_name = tensorname_to_nodename[tensor_name]
    tensor = state_dict[tensor_name]
    removed_inputs = pruning_info[node_name]['removed_inputs']
    removed_outputs = pruning_info[node_name]['removed_outputs']
    if tensor.ndim == 4 or tensor.ndim == 2:
      new_tensor = np.delete(tensor, removed_outputs, axis=0)
      new_tensor = np.delete(new_tensor, removed_inputs, axis=1)
    elif tensor.ndim == 1:
      new_tensor = np.delete(tensor, removed_outputs, axis=0)
    else:
      pass
    return new_tensor

  for key, module in model.named_modules():
    weight_key = key + '.weight'
    if weight_key in tensorname_to_nodename:
      node_name = tensorname_to_nodename[weight_key]
      bias_key = key + '.bias'
    if isinstance(module, (nn.BatchNorm2d, nn.BatchNorm3d)):
      state_dict[weight_key] = change_tensor(weight_key)
      module.weight = nn.Parameter(state_dict[weight_key])
      state_dict[bias_key] = change_tensor(bias_key)
      module.bias = nn.Parameter(state_dict[bias_key])
      running_mean_key = key + '.running_mean'
      state_dict[running_mean_key] = change_tensor(running_mean_key)
      module.running_mean = state_dict[running_mean_key]
      running_var_key = key + '.running_var'
      state_dict[running_var_key] = change_tensor(running_var_key)
      module.running_var = state_dict[running_var_key]
      if pruning_info[node_name]['out_dim'] != 0:
        module.num_features = pruning_info[node_name]['out_dim']
    elif isinstance(module, (nn.Conv2d, nn.Conv3d)):
      change_conv_module(module, node_name)
      state_dict[weight_key] = change_tensor(weight_key)
      module.weight = nn.Parameter(state_dict[weight_key])
      if bias_key in state_dict:
        state_dict[bias_key] = change_tensor(bias_key)
        module.bias = nn.Parameter(state_dict[bias_key])
    elif isinstance(module, (nn.ConvTranspose2d, nn.ConvTranspose3d)):
      change_conv_module(module, node_name)
      state_dict[weight_key] = change_tensor(weight_key)
      module.weight = nn.Parameter(state_dict[weight_key])
      if bias_key in state_dict:
        state_dict[bias_key] = change_tensor(bias_key)
        module.bias = nn.Parameter(state_dict[bias_key])
    elif isinstance(module, nn.Linear):
      state_dict[weight_key] = change_tensor(weight_key)
      module.weight = nn.Parameter(state_dict[weight_key])
      if bias_key in state_dict:
        state_dict[bias_key] = change_tensor(bias_key)
        module.bias = nn.Parameter(state_dict[bias_key])
      if pruning_info[node_name]['out_dim'] != 0:
```

```
        module.out_features = pruning_info[node_name]['out_dim']
      if pruning_info[node_name]['in_dim'] != 0:
        module.in_features = pruning_info[node_name]['in_dim']
    else:
      pass
  model.load_state_dict(state_dict)
  return model

slim_model = slim_model_from_state_dict_and_pruning_info(model, state_dict,
json_path)
```

Once-for-All (OFA)

Steps for the once-for-all method are as follows:

Creating a Model

For simplicity, mobilenet_v2 from `torchvision` is used here.

```
from torchvision.models.mobilenet import mobilenet_v2
model = mobilenet_v2(pretrained=True)
```

Creating an OFA Pruner

The pruner requires two arguments:

- The model to be pruned
- The inputs needed by the model for inference

```
import torch
from pytorch_nndct import OFAPruner

inputs = torch.randn([1, 3, 224, 224], dtype=torch.float32)
pruner = OFAPruner(model, inputs)
```

***Note*:** The input does not need to be real data. You can use randomly generated dummy data if it has the same shape and type as the real data.

Generating an OFA Model

Call `ofa_model()` to get an OFA model. This method finds all the `nn.Conv2d` / `nn.ConvTranspose2d`and `nn.BatchNorm2d` modules, then replaces those modules with `DynamicConv2d` / `DynamicConvTranspose2d` and `DynamicBatchNorm2d`.

A list of pruning ratio is required to specify what the OFA model will be.

For each convolution layer in the OFA model, an arbitrary pruning ratio can be used in the output channel. The maximum and minimum values in this list represent the maximum and minimum compression rates of the model. Other values in the list represent the subnetworks to be optimized. By default, the pruning ratio is set to [0.5, 0.75, 1].

For a subnetwork sampled from the OFA model, the out channels of a convolution layer is one of the numbers in the pruning ratio list multiplied by its original number. For example, for a pruning ratio list of [0.5, 0.75, 1] and a convolution layer nn.Conv2d(16, 32, 5), the out channels of this layer in a sampled subnetwork is one of [0.5*32, 0.75*32, 1*32].

Because the first and last layers have a significant impact on network performance, they are commonly excluded from pruning. By default, this method automatically identifies the first convolution and the last convolution, then puts them into the list of excludes. Setting `auto_add_excludes` equals False can cancel this feature.

```
ofa_model = ofa_pruner.ofa_model([0.5, 0.75, 1], excludes = None,
auto_add_excludes=True)
```

Training an OFA Model

This method uses the sandwich rule to jointly optimize all the OFA subnetworks. The `sample_random_subnet()` function can be used to get a subnetwork. The dynamic subnetwork can do a forward/backward pass.

In each training step, given a mini-batch of data, the sandwich rule samples a 'max' subnetwork, a 'min' subnetwork, and two random subnetworks. Each subnetwork does a separate forward/backward pass with the given data and then all the subnetworks update their parameters together.

```
# using sandwich rule and sampling subnet.
for i, (images, target) in enumerate(train_loader):

  images = images.cuda(non_blocking=True)
  target = target.cuda(non_blocking=True)

  # total subnets to be sampled
  optimizer.zero_grad()

  teacher_model.train()
  with torch.no_grad():
    soft_logits = teacher_model(images).detach()

  for arch_id in range(4):
    if arch_id == 0:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'max')
    elif arch_id == 1:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'min')
    else:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'random')

    output = model(images)

    loss = kd_loss(output, soft_logits) + cross_entropy_loss(output,
target)
    loss.backward()

  torch.nn.utils.clip_grad_value_(ofa_model.parameters(), 1.0)
  optimizer.step()
  lr_scheduler.step()
```

Searching Constrained Subnetworks

After the training is completed, you can conduct an evolutionary search based on the neural-network-twins to get a subnetwork with the best trade-offs between FLOPs and accuracy using a minimum and maximum FLOPs range.

```
pareto_global = ofa_pruner.run_evolutionary_search(ofa_model,
calibration_fn, (train_loader,) eval_fn, (val_loader,), 'acc1', 'max',
min_flops=230, max_flops=250)

ofa_pruner.save_subnet_config(pareto_global, 'pareto_global.txt')
```

The searching result looks like the following:

```
{
"230": {
    "net_id": "net_evo_0_crossover_0",
    "mode": "evaluate",
    "acc1": 69.04999542236328,
    "flops": 228.356192,
    "params": 3.096728,
    "subnet_setting": [...]
}
"240": {
    "net_id": "net_evo_0_mutate_1",
    "mode": "evaluate",
    "acc1": 69.22000122070312,
    "flops": 243.804128,
    "params": 3.114,
    "subnet_setting": [...]
}}
```

Getting a Subnetwork

Call `get_static_subnet()` to get a specific subnetwork. The static_subnet can be used for finetuning and doing quantization.

```
pareto_global = ofa_pruner.load_subnet_config('pareto_global.txt')
static_subnet, static_subnet_config, flops, params = \
ofa_pruner.get_static_subnet(ofa_model, pareto_global['240']
['subnet_setting'])
```

vai_p_pytorch APIs

pytorch_nndct.get_pruning_runner

This API has the following methods:

```
get_pruning_runner(model, inputs, method)
```

- **model:** A torch.nn.Module object to prune.

- **inputs:** A single or a list of torch.Tensor used as inputs for model inference. It need not be real data. It can be a randomly generated tensor of the same shape and data type as the real data.
- **method:** Either be 'iterative' or 'one_step'.

pytorch_nndct.IterativePruningRunner

This API has the following methods:

- `__init__(model, inputs)`
  - **model:** A `torch.nn.Module` object to prune.
  - **inputs:** A single or a list of torch.Tensor used as inputs for model inference. It does not need to be real data. It can be a randomly generated tensor of the same shape and data type as the real data.
- `ana(eval_fn, args=(), gpus=None, excludes=None, forced=False)`
  - **eval_fn:** Callable object that takes a `torch.nn.Module` object as its first argument and returns the evaluation score.
  - **args:** A tuple of arguments that will be passed to eval_fn.
  - **gpus:** A tuple or list of GPU indices to be used. If not set, the default GPU will be used.
  - **excludes:** A list of node names or torch modules to be excluded from pruning.
  - **forced:** If False, skip model analysis and use cached result.
- `prune(removal_ratio=None, threshold=None, spec_path=None, excludes=None, mode='sparse')`
  - **removal_ratio:** The expected percentage of MACs reduction.
  - **threshold:** Relative proportion of model performance loss that can be tolerated.
  - **spec_path:** Pre-defined pruning specification.
  - **excludes:** A list of node names or torch modules to be excluded from pruning.
  - **mode:** One of ['sparse', 'slim']. Should always using 'sparse' in iterative loop. A slim model is used for quantization-aware training.

pytorch_nndct.OneStepPruningRunner

This API has the following methods:

- `__init__(model, inputs)`
  - **model:** A `torch.nn.Module` object to prune.
  - **inputs:** A single or a list of torch.Tensor used as inputs for model inference. It does not need to be real data. It can be a randomly generated tensor of the same shape and data type as the real data.

- `search(gpus=['0'], calibration_fn=None, calib_args=(), num_subnet=10, removal_ratio=0.5, excludes=[], eval_fn=None, eval_args=())`
  - **gpus:** A tuple or list of GPU indices to be used. If not set, the default GPU will be used.
  - **calibration_fn:** Callable object that takes a torch.nn.Module object as its first argument. It is used for calibrating statistics of the BatchNormalization layers.
  - **calib_args:** A tuple of arguments that is passed to calibration_fn.
  - **num_subnet:** Number of subnetworks that satisfy the flops constraint.
  - **removal_ratio:** The expected percentage of MACs reduction.
  - **excludes:** Modules that need to exclude from pruning.
  - **eval_fn:** Callable object that takes a `torch.nn.Module` object as its first argument and returns the evaluation score.
  - **eval_args:** A tuple of arguments that is passed to eval_fn.
- `prune(mode='slim', index=None, removal_ratio=None, pruning_info_path=None)`
  - **mode:** One of ['sparse', 'slim']. Should always use 'slim' mode for one-step method.
  - **index:** Subnetwork index. By default, the optimal subnetwork is selected automatically.
  - **removal_ratio:** The expected percentage of MACs reduction.
  - **pruning_info_path:** A .json file. Save detailed pruning information for current model. A slim model can be generated with the file and origin model.

pytorch_nndct.OFAPruner

This API has the following methods:

- `__init__(model, inputs)`
  - **model:** A `torch.nn.Module` object to prune.
  - **inputs:** A single or a list of torch. Tensor used as inputs for model inference. It does not need to be real data. It can be a randomly generated tensor of the same shape and data type as the real data.
- `ofa_model(expand_ratio, channel_divisble=8, excludes=None, auto_add_excludes=True, save_search_space=False)`
  - **expand_ratio:** A list of prune ratio of each convolution layer. For each convolution layer in the OFA model, arbitrary pruning ratio can be used in output channel.

    The maximum and minimum values in this list represent the maximum and minimum compression rates of the model. Other values represent subnetworks to be optimized. By default, the pruning ratio is set to [0.5, 0.75, 0.1].
  - **channel_divisible:** A channel number that is divisible by a given divisor.
  - **excludes:** A list of modules to be excluded from pruning.

  - **auto_add_excludes:** Bool. If True, this method automatically identifies the first convolution and the last convolution, then put them into the list of excludes. If False, not doing the above. Defaults to True.
  - **save_search_space:** Bool. If True, save the search space of the model as a file of 'searchspace.config'. Customers can check the search space for each layer. Defaults to False.

- `sample_subnet(model, mode)`

  Returns a subnetwork and its configuration for a given mode. The subnetwork can do a forward/backward process using a part of the weights from the OFA model and its settings.

  - **model:** The OFA model.
  - **mode:** One of ['random', 'max', 'min'].

- `reset_bn_running_stats_for_calibration(model)`

  Resets the running stats of the Batch Normalization layers.

  - **model:** The OFA model.

- ```
  run_evolutionary_search(model, calibration_fn, calib_args, eval_fn,
  eval_args, evaluation_metric, min_or_max_metric, min_flops, max_flops,
  flops_step=10, parent_popu_size=16, iteration=10, mutate_size=8,
  mutate_prob=0.2, crossover_size=4)
  ```

  Runs an evolutionary search to find the best subnetwork whose flops are in a given range.

  - **model:** The OFA model.
  - **calibration_fn:** A BatchNormalization calibration function. All subnetworks share weights in an OFA model, but batch normalization statistics (mean and variance) are not stored when training the OFA model. After the training is completed, the batch normalization statistics must be re-calibrated using the training data for each sampled subnetwork for evaluation.
  - **calib_args:** The arguments for calibration_fn.
  - **eval_fn:** A function to evaluate the model.
  - **eval_args:** The arguments for eval_fn.
  - **evaluation_metric:** A string of evaluation_metric to record the result.
  - **min_or_max_metric:** One of ['max', 'min']. The maximum or minimum value of the evaluation metric to be recorded in the evolutionary search. For example, when the evaluation metric has an accuracy of top1, record the maximum value of each iteration in the evolutionary search. However, when the evaluation metric is mean squared error (mse) or mean absolute error (mae), record the minimum value.
  - **min_flops:** The minimum flops of searched subnetworks.
  - **max_flops:** The maximum flops of searched subnetworks.
  ```

- **flops_step:** The step of flops for searching. Divides the interval [min_flops, max flops] into segments by flops_step. For each segment, searches the best FLOPs-accuracy trade-offs subnetwork.
- **parent_popu_size:** The number of initial parent population for sampling the given number of random subnetworks whose flops are in the given range. The larger this number is, the longer the search will take and the more likely it is that the best results will be obtained.
- **iteration:** The number of iterations for searching or the number of cycles of the whole algorithm.
- **mutate_size:** The size of mutation. Each value of the subnetwork setting is replaced with another value of the candidate list with probability of mutate_prob.
- **mutate_prob:** The probability of mutation.
- **crossover_size:** The size of crossover. Sampling two subnetwork settings and swapping any value in the two subnetwork settings randomly.

- `save_subnet_config(setting_config, file_name)`

  Saving dynamic/static subnetwork settings with JSON.

  - **setting_config:** The configurations for the dynamic subnetwork setting.
  - **file_name:** Filepath to save the subnetwork settings.

- `load_subnet_config(file_name)`

  - **file_name:** Filepath to load the subnetwork settings.

Example Networks

TensorFlow Examples

MNIST

The MNIST dataset has a training set of 60,000 examples and a test set of 10,000 examples of the handwritten digits. Each example is a 28 x 28-pixel monochrome image.

This sample shows the use of low-level APIs and tf.estimator.Estimator to build a simple convolution neural network classifier and to use `vai_p_tensorflow` to prune it.

Downloading and Converting the Dataset

Create a file called `data_utils.py` and add the following code:

```
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)
```

The `dataset_utils` function calls `get_init_data` taking `train_batch` and `test_batch` as arguments and returns an image, label tensors, and initializer operations to train and test data respectively, which will now run in training and evaluation.

`data_utils.py` is imported as a module to provide input data pipeline. You can also run it in shell to download the MNIST dataset and convert it into TFRecord format using the following command:

```
$ python data_utils.py
```

This generates the following:

```
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
```

Building the CNN MNIST Classifier

Create a file called `low_level_cnn.py`, and add the following code:

```
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
```

The `net_fn` function defines the network architecture. It takes the MNIST image data as argument and return a logits tensor. The `model_fn` function reads an input data pipeline and returns a dictionary of evaluation metrics operations.

Building, Training, and Evaluating the Model

Create a file called `train_eval_utils.py`, and add the following code:

```
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 is a class which can build graphs, and train and evaluate models. It is a framework created by combining the data utils, net definition, and metrics. To train and evaluate a model, instantiate a ConvNet class, then call the class method `build` to build, train, or evaluate a graph by setting the `test_only` argument as True.

Training the Model

To train the model, create a file named `train.py` and add following code:

```
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()
```

Run `train.py` in shell:

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

The running output log looks like this:

```
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: 246.51300048828125
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
```

After a few minutes, a trained checkpoint is generated: `models/train/model.ckpt`.

Exporting an Inference GraphDef File

Create a file named `export_inf_graph.py` and add the following code:

```
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()
```

Run `export_inf_graph.py`.

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

Running Model Analysis

Now that you have prepared a trained checkpoint and a GraphDef file, you can start the pruning process. Run the following shell scripts to call the `vai_p_tensorflow` functions.

```
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}\"
```

The output log is as shown below:

```
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
```

Pruning the Model

You can prune the model now and write some shell scripts to call the `vai_p_tensorflow` functions.

```
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
```

Fine-tuning the Pruned Model

Create a file named `ft.py` and add the following code:

```
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()
```

Note: You must call `tf.set_pruning_mode()` before creating the model. The API is used to enable the sparse training mode, that is, the weights of pruned channels are set to zero and not updated during training. If you fine-tune a pruned model without calling this function, the pruned channels will be updated and finally you will get a normal non-sparse model.

Fine-tune the pruned model and run `ft.py`:

```
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
```

The output log looks like:

```
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
```

As a final step, transform and freeze the fine-tuned model to get a dense model.

```
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}
```

Finally, you should have a frozen GraphDef file named `mninst.pb` in the `models/pruned` location.

TensorFlow2 Examples

See the Vitis AI repository on GitHub.

PyTorch Examples

See https://github.com/Xilinx/Vitis-AI/tree/2.5/examples/Vitis-AI-Optimizer/vai_p_pytorch.

AMD XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Vitis AI User Guide* (UG1414)
2. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/15/2022 Version 2.5	
Chapter 2: Pruning	Updated Once-for-All (OFA) and Supported Frameworks and Features.
Chapter 3: Working with the Vitis AI Pruner	Updated PyTorch Version - vai_p_pytorch.
Chapter 4: Example Networks	Updated TensorFlow2 Examples and PyTorch Examples. Removed Caffe and Darknet examples.
01/20/2022 Version 2.0	
Chapter 2: Pruning	Added Once-for-All (OFA).
Chapter 3: Working with the Vitis AI Pruner	Added TensorFlow (2.x) Version – vai_p_tensorflow2. Added Once-for-All (OFA) for PyTorch. Added new APIs to vai_p_pytorch APIs.
10/29/2021 Version 1.4.1	
Chapter 4: Example Networks	Updated VGG16 and ResNet18.
07/22/2021 Version 1.4	
Chapter 3: Working with the Vitis AI Pruner	Updated chapter. Added Keras Model.
Chapter 4: Example Networks	Added ResNet50.
02/03/2021 Version 1.3	
Darknet Version	Added Preparing Training Dataset section.
12/17/2020 Version 1.3	
Entire document	Minor changes.
PyTorch Version - vai_p_pytorch	Added new section.
Chapter 4: Example Networks	Added PyTorch Examples.
07/07/2020 Version 1.2	
Entire document	Minor changes.
03/23/2020 Version 1.1	
Entire document	Minor changes.
Vitis AI Pruner License	Added new topic

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright