Vitis AI Optimizer User Guide

UG1333 (v1.3) February 3, 2021

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
02/03/2021 Version 1.3	
Preparing Training Dataset	Added new 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.
VAI Pruner License	Added new topic

Table of Contents

Chapter 1

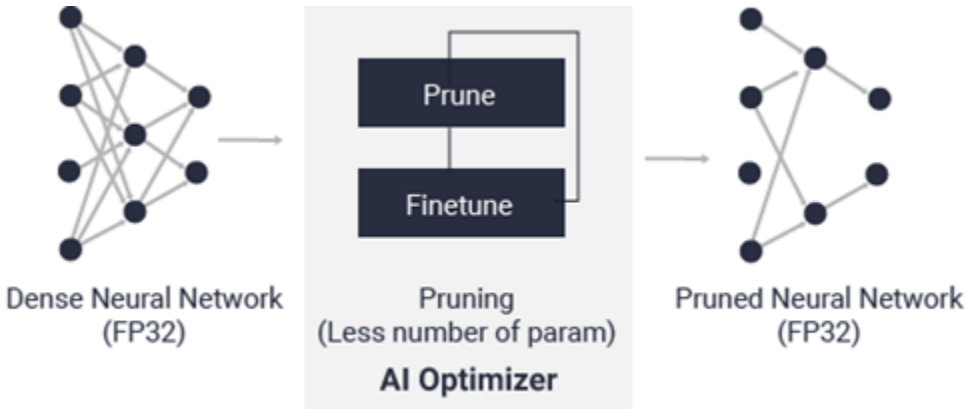
Overview and Installation

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.

Vitis AI optimizer provides the ability to optimize neural network models. Currently, Vitis AI optimizer includes only one tool called pruner. Vitis AI pruner (VAI pruner) prunes redundant connections in neural networks and reduces the overall required operations. The pruned models produced by VAI pruner can be further quantized by VAI quantizer and deployed to an FPGA. For more information on VAI quantizer and deployment, see the *Vitis AI User Guide* in the *Vitis AI User Documentation* (UG1431).

Figure 1: **VAI Optimizer**



The VAI pruner supports four deep learning frameworks: TensorFlow, PyTorch, Caffe, and Darknet. The corresponding tool names are vai_p_tensorflow, vai_p_pytorch, vai_p_caffe, and vai_p_darknet, where the "p" in the middle stands for pruning.

Vitis AI Optimizer requires a commercial license to run. Contact xilinx_ai_optimizer@xilinx.com to access 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. 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 VAI Pruner
  - Chapter 4: Example Networks

Installation

The following are the two ways to obtain the Vitis AI Optimizer:

- **Docker Image:** Vitis AI provides a docker environment for the Optimizer. In the docker image, there are three optimizer related conda environments: vitis-ai-optimizer_tensorflow, vitis-ai-optimizer_caffe, and vitis-ai-optimizer_darknet. All requirements are ready in these environments. CUDA and cuDNN versions in the docker are CUDA 10.0 and cuDNN 7.6.5. After getting a license, you can run the VAI pruner directly in the docker.

  ***Note*:** The optimizer for PyTorch is not in the docker image and can only be installed using the conda package.

- **Conda Packages:** Conda packages are also available for Ubuntu 18.04. Contact xilinx_ai_optimizer@xilinx.com to access the Vitis AI Optimizer installation package and license. Follow the installation steps to install the pre-requirements and Vitis AI Optimizer.

Hardware Requirements

Nvidia GPU card with CUDA Compute Capability >= 3.5 is required. It is recommended to use Tesla P100 or Tesla V100.

Software Requirements

***Note*:** This section is only required for installing the Conda package. For Docker image, skip this section. The dependencies are already there.

- **GPU-related Software:** Install GPU related software according to the operating system. For Ubuntu 16.04, install CUDA 9.0, cuDNND 7 and Driver 384 or above. For Ubuntu 18.04, install CUDA 10.0, cuDNN 7 and Driver 410 or above.
- **NVIDIA GPU Drivers:**

  Install GPU driver by apt-get or directly install the CUDA package with driver. For example:

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

- **CUDA Toolkit:**

  Get the CUDA package associated with the Ubuntu version from https://developer.nvidia.com/cuda-toolkit-archive and directly install the NVIDIA CUDA runfile package.

- **cuDNN SDK:** Get cuDNN from https://developer.nvidia.com/cudnn and append its installation directory to the `$LD_LIBRARY_PATH` environmental variable.

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

- **CUPTI:** CUPTI is required by `vai_p_tensorflow` and is installed together with CUDA. You must add the CUPTI directory to the `$LD_LIBRARY_PATH` environment variable. For example:

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

- **NCCL:** NCCL is required by `vai_p_caffe`. Download NCCL from its homepage (https://developer.nvidia.com/nccl/nccl-legacy-downloads) and install.

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

VAI Pruner

Note: To install the VAI pruner, you must install the Conda package first. For the Docker image, skip this section. VAI pruner already exists in the coresponding Conda environments.

To install VAI pruner, first install Conda and then install the Conda package for your framework.

Install Conda

For more information, see the Conda installation guide.

vai_optimizer_tensorflow

vai_p_tensorflow is based on TensorFlow 1.15. Install this vai_optimizer_tensorflow package to get vai_p_tensorflow.

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

vai_optimizer_pytorch

vai_optimizer_pytorch is a Python library and you can use it by calling its APIs.

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

vai_optimizer_caffe

vai_p_caffe binary is included in this vai_optimizer_caffe conda package.

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

vai_optimizer_darknet

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

VAI Pruner License

There are two types of license: floating license and node-locked license. The VAI Pruner finds licenses using an environment variable `XILINXD_LICENSE_FILE`. For floating license server, you need to specify the path in the form port@hostname. For example, `export XILINXD_LICENSE_FILE=2001@xcolicsvr1`. For node-locked license file, you need to specify a particular license file or directory where all the `.lic` files 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, you can specify them at the same time, each separated by a colon:

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

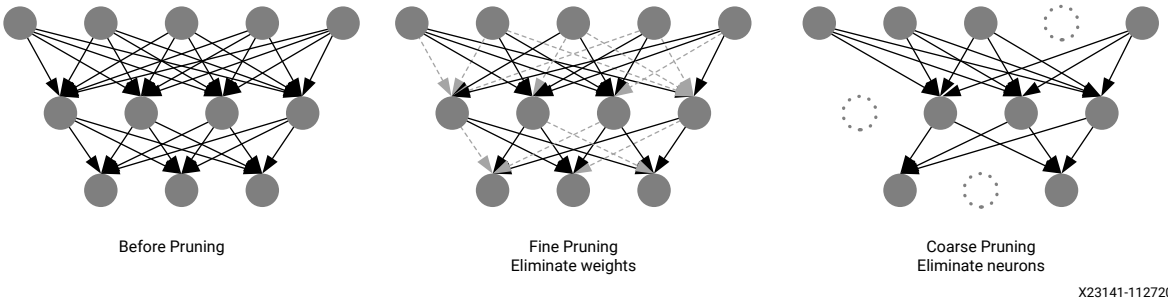
For node-locked license, it can also be installed by copying to `$HOME/.Xilinx directory`.

Pruning

Pruning Overview

Most neural networks are typically over-parameterized, with significant redundancy to achieve a certain accuracy. “Pruning” is the process of eliminating redundant weights while keeping the accuracy loss as low as possible.

Figure 2: **Pruning Methods**



The simplest form of pruning is called “fine-grained pruning” and results in sparse weight matrices. The Vitis AI pruner employs the “coarse-grained pruning” method, which eliminates neurons that do not contribute significantly to the network’s accuracy. For convolutional layers, “coarse-grained pruning” prunes the entire 3D kernel and so is also called channel pruning.

Pruning always reduces the accuracy of the original model. Retraining (finetuning) adjusts the remaining weights to recover accuracy.

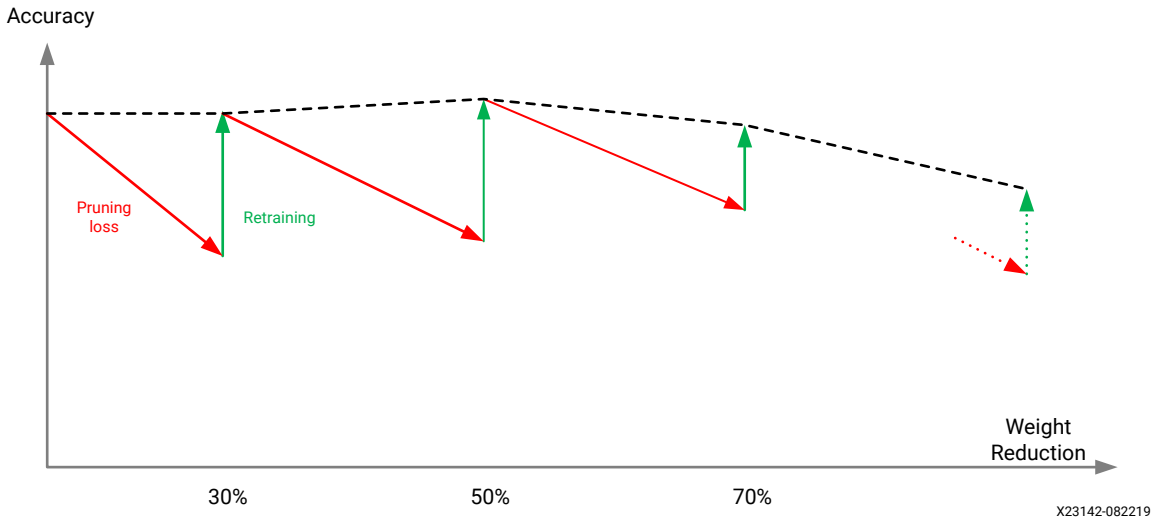
Iterative Pruning

The Vitis AI pruner is designed to reduce the number of model parameters while minimizing the accuracy loss. This is done in an iterative process as shown in the following figure. Pruning results in accuracy loss and 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 previous iterations is used to prune again. This process is usually repeated several times until a desired sparse model is obtained. A model cannot be pruned to a smaller size at once. Once too many parameters are removed from the model, the performance of the model is reduced and it is challenging to restore the model.

IMPORTANT! *The reduction parameter is gradually increased in every iteration, to help better recover accuracy during the finetune stage.*

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

Figure 3: **Iterative Process of Pruning**



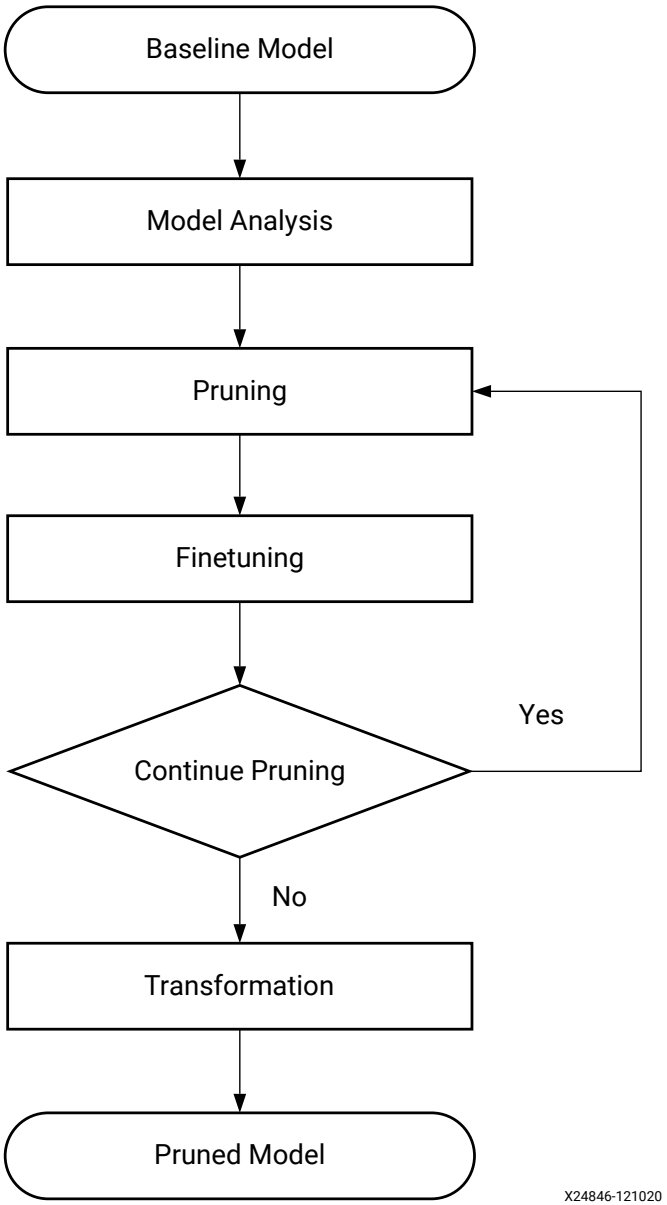
The four primary tasks in the Vitis AI pruner are as follows:

1. Analysis (ana): Perform a sensitivity analysis on the model to determine the optimal pruning strategy.
2. Pruning (prune): Reduce the number of computations in the input model.
3. Fine-tuning (finetune): Retrain the pruned model to recover accuracy.
4. Transformation (transform): 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. Finetune the pruned model.
4. Repeat steps 2 and 3 several times.
5. Transform the pruned sparse model to a final dense model.

Figure 4: **Pruning Workflow**



Guidelines for Better Pruning Results

The following is a list of suggestions for better pruning results, higher pruning rate, and smaller accuracy loss.

1. Use as much data as possible to perform a model analysis. Ideally, you should use all the data in the validation dataset, which 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.
2. During the finetuning 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.
3. The data used in fine-tuning should be the same as the data used to train the baseline.
4. If the accuracy does not improve sufficiently after several finetuning experiments, try reducing the pruning rate and then re-run pruning and finetuning.

Working with VAI Pruner

TensorFlow Version - vai_p_tensorflow

Exporting an Inference Graph

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 TensorFlow Estimator API to train and evaluate your network, your `model_fn` must return an instance of tf.estimator. At the same time, you also need to 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)
```

The 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))
```

Performing Model Analysis

Before conducting model pruning, you need to analyze the model first. The main purpose of this process is to find a suitable pruning strategy to prune the model.

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

- A function named `model_fn()` that returns a Python dict 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
```

Suppose you use the first way to write the script, the following snippet shows how to call `vai_p_tensorflow` to perform model analysis.

```
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 is more than one score in the network, you should 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 command `ana` has ended, 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 it a value between 0 and 1. The larger the value, the sparser the model is after pruning.

When the prune command finishes, the `vai_p_tensorflow` outputs FLOPs of network before and after pruning.

Finetuning the Pruned Model

The performance of the pruned model has a certain degree of decline and you need to fine-tune it to improve its performance. Finetuning 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.

When pruning and fine-tuning is done, an iteration of pruning is completed. In general, to achieve higher pruning rate without significant loss of performance, the model needs to be pruned several times. After every iteration of "prune-finetune", you need to make two 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 `--sparsity` flag to prune more in the next iteration.

Generating Dense Checkpoints

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

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

It should be noted that transformation is only required after all iterations of pruning are completed. Do not run the transform command between each iteration of pruning.

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. For more details about freezing, see Using the Saved Format.

Freezing the Graph

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. This file can be used 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 1: **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.
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.

Table 1: **vai_p_tensorflow Arguments** *(cont'd)*

Argument	Type	Action	Default	Description
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.

PyTorch Version - vai_p_pytorch

The pruning tool on PyTorch is a Python package rather than an executable program. Use the pruning APIs to prune the model.

Preparing a Baseline Model

For simplicity, ResNet18 from torchvision is used here.

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

Creating a Pruner

A pruner can be created by providing the model to be pruned and its input shape and input dtype. Note that shape is the size of the input image and does not contain batch size.

```
from pytorch_nndct import Pruner
from pytorch_nndct import InputSpec

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

For models with multiple inputs, you can use a list of `InputSpec` to initialize a pruner.

Model Analysis

To run model analysis, you need to define a function that can be used to evaluate the model. The first argument of this function must be the model to be evaluated.

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

  # switch to evaluate mode
  model.eval()

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

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

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

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

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

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

  return top1.avg, top5.avg

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

Then, call `ana()` method with the function defined above as the first argument.

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

Here, the `'args'` is the tuple of arguments starting from the second argument required by `'ana_eval_fn'`.

Pruning the Model

Call `prune()` method to get a pruned model. The ratio is the proportion of FLOPs expected to be reduced.

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

Finetuning the Pruned Model

The process of fine-tuning is the same as training a baseline model. The difference is that the weights of the baseline model are randomly initialized and the weights of the pruned model are inherited from the baseline model.

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

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

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

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

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

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

  # switch to train mode
  model.train()

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

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

    # compute output
    output = model(images)
```

```
    loss = criterion(output, target)

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

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

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

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

Next, run the training loop. Here the parameter 'model' in `train()` function is the returned object from `prune()` method.

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

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

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

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

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

Note: In the last two lines of code we save two checkpoint files. 'model.save()' saves sparse weights with the same shapes as the baseline model and the removed channel is set to 0. 'model.state_dict()' returns dense weights with the pruned shapes. The first checkpoint is used as input for the next round of pruning, and the second checkpoint is used for the final deployment. That is, if there is another round of pruning, then use the first checkpoint, and if this is the last round of pruning, then use the second checkpoint.

Iterative Pruning

Load the sparse checkpoint and increase pruning ratio. Here, the pruning ratio is increased from 0.1 to 0.2.

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

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

When you get the new pruned model, you can start fine-tuning again.

vai_p_pytorch APIs

pytorch_nndct.InputSpec

Specifies the dtype and shape of every input to a module.

Arguments

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

- **shape:** Shape tuple, expected shape of the input.
- **dtype:** Expected torch.dtype of the input.

pytorch_nndct.Pruner

Implements channel pruning at the module level.

Arguments

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

Create a new Pruner object.

- **module:** A `torch.nn.Module` object to be pruned.
- **input_specs:** The inputs of the module: a `InputSpec` object or list of `InputSpec`.

Methods

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

  Performs model analysis.

  - **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 used for model analysis. If not set, the default GPU will be used.

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

  Prune the network by given ratio or threshold. Return a `PruningModule` object works like a normal `torch.nn.Module` with addtional pruning info.

  - **ratio:** The expected percentage of FLOPs reduction. This is just a hint value, the actual FLOPs drop not necessarily strictly to this value after pruning.
  - **threshold:** Relative proportion of model performance loss that can be tolerated.
  - **excludes:** Modules that need to prevent from pruning.
  - **output_script:** Filepath that saves the generated script used for rebuilding model.

- `summary(pruned_model)`

  Get the pruning summary of the pruned model.

  - **pruned_model:** A pruned module returned by `prune()` method.

pytorch_nndct.pruning.core.PruningModule

Represents a pruned module returned by `pytorch_nndct.Pruner.prune()`.

Attributes

- **module:** A `torch.nn.Module` that represents the actual pruned module.
- **pruning_info:** A dictionary containing pruning details of each layer.

Methods

```
save(path)
```

Saves sparse state to given path.

- **path:** Checkpoint path to save.

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

Returns a dictionary containing a whole state of the module. Refer to related Pytorch Document

```
padded_state_dict()
```

Returns a dictionary containing a sparse state of the module. The shape of the state is the same as the original baseline model with the pruned channels filled with zeros.

Caffe Version - vai_p_caffe

Creating a Configuration File

Most `vai_p_caffe` tasks require a configuration file as an input argument. A typical configuration file is shown below:

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

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

rate: 0.1

pruner {
  method: REGULAR
}
```

The definition for the terms used are:

- **workspace:** Directory for saving temporary and output files.
- **gpu:** Use the given GPU devices IDS separated by ',' for acceleration.
- **test_iter:** The number of iterations to use in a test phase. A larger value improves the analysis results but increases the run time. The maximum value of this parameter is determined by the size of the validation dataset/batch_size, i.e, all data in the validation dataset will be used for testing.
- **acc_name:** The accuracy measure used to determine the "goodness" of the model.
- **model:** The model definition protocol buffer text file. If there are two separate model definition files used in training and testing, merge them into a single file.
- **weights:** The model weights to be pruned.
- **solver:** The solver definition protocol buffer text file used for finetuning.
- **rate:** The weight reduction parameter sets the amount by which the number of computations is reduced relative to the baseline model. For example, with a setting of "0.1," the tool attempts to reduce the number of multiply-add operations by 10% relative to the baseline model.
- **method:** Pruning method is used. Currently, REGULAR is the only valid value.

Performing Model Analysis

This is the first stage of the pruning process. This task attempts to find a suitable pruning strategy. Create a suitable configuration file named `config.prototxt`, as described in the previous section, and execute the following command:

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

Figure 5: **Model Analysis**

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

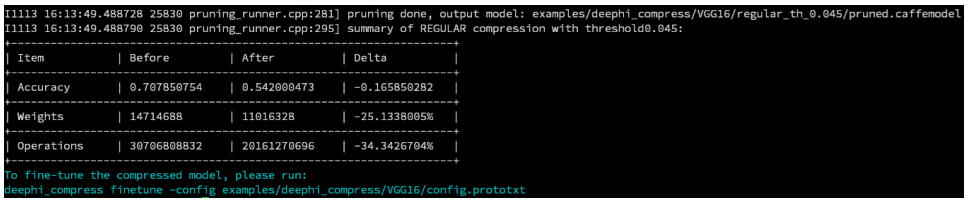
Starting Pruning Loop

Pruning can begin after the analysis task completed. The prune command uses the same configuration file:

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

vai_p_caffe prunes the model using the rate parameter specified in the configuration file. Upon completion, the tool generates a report that includes the accuracy, the number of weights, and the required number of operations before and after pruning. The following figure shows a sample report.

Figure 6: **Pruning Report**



A file named `final.prototxt` which describes the pruned network is generated in the workspace.

Finetuning the Pruned Model

Run the following command to recover the accuracy loss from pruning:

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

Finetuning a pruned model is essentially the same as training the model from scratch. The solver parameters such as initial learning rate, learning rate decay type, etc. may be different. A pruning iteration is composed of the prune and finetune tasks executed sequentially. In general, to achieve a greater weight reduction without significant accuracy loss, several pruning iterations must be performed.

The configuration file needs to be modified after every pruning iteration:

1. Increase the *rate* parameter relative to the baseline model.
2. Modify the *weights* parameter to the best model obtained in the previous finetuning step.

A modified configuration file is shown below:

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

model: "examples/decent_p/float.prototxt"

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

solver: "examples/decent_p/solver.prototxt"

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

Generating the Final Model

After a few pruning iterations, a model with fewer weights is generated. The following transformation step is required to finalize the model:

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

If you fail to specify the name of the output file, a default file named `transformed.caffemodel` is generated. The corresponding model file is the `final.prototxt` generated by the prune command.

To get the FLOPs of a model, you can use the stat command:

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

IMPORTANT! *The transformation should only be executed after all pruning iterations have been completed.*

vai_p_caffe Usage

The following arguments are available when running vai_p_caffe:

Table 2: **vai_p_caffe Arguments**

Argument	Attribute	Default	Description
ana			
config	required	“”	The configuration file path.
prune			
config	required	“”	The configuration file path.
finetune			
config	required	“”	The configuration file path.
transform			
model	required	“”	Baseline model definition protocol buffer text file
weights	required	“”	Model weights file path.
output	optional	“”	The output transformed weights.

Table 3: **vai_p_caffe Configuration File Parameters**

Argument	Type	Attribute	Default	Description
workspace	string	required	None	Directory for saving output files.
gpu	string	optional	“0”	GPU device IDs used for compression and fine-tuning, separated by ‘,’.
test_iter	int	optional	100	The number of iterations to run in test phase.

Table 3: **vai_p_caffe Configuration File Parameters** *(cont'd)*

Argument	Type	Attribute	Default	Description
acc_name	string	required	None	The accuracy measure of interest. This parameter is the layer_top of the layer used to evaluate network performance. If the network has multiple evaluation metrics, choose the one that you think is most important. For classification tasks, this parameter may be top-1 or top-5 accuracy; for detection tasks, this parameter is generally mAP; for segmentation tasks, typically the layer for calculating mIOU is set here.
model	string	required	None	The model definition protocol buffer text file. If there are two different model definition files for training and testing, it is recommended to merge them into a single file.
weights	string	required	None	The trained weights to compress.
solver	string	required	None	The solver definition protocol buffer text file.
rate	float	optional	None	The expected model pruning ratio.
method	enum	optional	REGULAR	Pruning method to be used. Currently REGULAR is the only valid value.
ssd_ap_version	string	optional	None	The ap_version setting for SSD network compression. Must be one of 11point, MaxIntegral and Integral.
exclude	repeated	optional	None	Used to exclude some layers from pruning. You can use this parameter to prevent specified convolutional layers from being pruned.
kernel_batch	int	optional	2	The number of output channels is a multiple of this value after pruning.

Darknet Version - vai_p_darknet

Creating a Configuration File

A typical main cfg file for YOLOv3 pruning is as follows. In this example, you will prune a YoloV3 model trained on VOC dataset and prepare the VOC data in the standard Darknet way. Refer to the YOLO website for details.

Because of the YOLOv3 network structure, the convolution layer before the YOLO layer cannot be pruned, which means that if you use standard YOLOv3 cfg file, layer 81, 93, and 105 should be added to “ignore_layer”. A full list of main cfg options can be found in the vai_p_darknet Usage section.

RECOMMENDED: *Do not prune the convolution layer before 81, 93, and 105. The ana step will be very slow if layers 80, 92,104 are not ignored.*

```
# a cfg example to prune YoloV3
[pruning]
workspace=pruning
datacfg=pruning/voc.data
modelcfg=pruning/yolov3-voc.cfg
prunedcfg=pruning/yolov3-voc-prune.cfg
```

```
ana_out_file=pruning/ana.out
prune_out_weights=pruning/weights.prune
criteria=0
kernel_batch=2
ignore_layer=80,81,92,93,104,105
yolov3=1
threshold=0.005
```

Preparing Training Dataset

Finetuning with the original training dataset is required to get a satisfactory pruning result. Refer to the Yolo page for how to prepare datasets for darknet. Taking Pascal VOC Dataset as an example, find or create a data cfg file `"voc.data"` such as the following.

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

The "train" text file specifies the training images.

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

At the same time, label files should be located in corespoonding "labels" folder. The directory hierarchy looks like this.

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

Performing Model Analysis

A model should be analyzed before conducting a model pruning. The main purpose of this process is to find the optimal pruning strategy to prune the model later. Run the following command to start the analysis:

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

Depending on the size of model and validation dataset, the `ana` command may take a couple of hours. Running ana with multi-gpu can greatly accelerate the running speed. The following command runs with four GPUs.

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

Starting Pruning Loop

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

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

The pruning tool prunes the model according to the value of threshold and generates an output as follows. After loading the model, the tool shows the pruning rate for each layer and the output weights file. Finally, it automatically evaluates the mAP of the pruned model. You can see that the mAP after pruning is 0.494531, much lower than the original one. Finetuning, which is the next step, is used to recover the accuracy. If you want to follow the iterative pruning with a small step size, change the threshold option in the main cfg to a smaller value and run again.

```
Start pruning ...
pruning slot: 0,
prune main layer 0
  kernel_batch:2, rate:0.3000, keep_num:24
pruning slot: 1,3,
prune main layer 1
  kernel_batch:2, rate:0.3000, keep_num:46
  prune slave layer 3
  prune related layer 2
  kernel_batch:2, rate:0.3000, keep_num:24
...
pruning slot: 102,
prune main layer 102
  kernel_batch:2, rate:0.5000, keep_num:128
  prune related layer 101
  kernel_batch:2, rate:0.5000, keep_num:64
Saving weights to pruning/weights.prune
calculate map
Process    100 on GPU 0
Process    200 on GPU 0
...
Process   4800 on GPU 0
Process   4900 on GPU 0
AP for class 0 = 0.501017
AP for class 1 = 0.711958
...
AP for class 18 = 0.621339
AP for class 19 = 0.472648
mAP : 0.494531
Total Detection Time: 158.943884 Seconds
```

Finetuning the Pruned Model

The performance of the pruned model has a certain degree of decline and must be finetuned to improve the performance.

Run the following command to start finetuning.

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

Multi-GPU finetuning is generally faster.

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

The command outputs the basic information at first. Modify the training parameters in `pruning/yolov3-voc-prune.cfg`, if necessary.

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

When pruning and fine-tuning are done, one iteration of pruning is completed. In general, to achieve higher pruning rate without significant loss of performance, the model needs several iterations. A typical workflow is as follows:

1. Setting a small threshold in the configuration file.
2. Start the pruning of the model.
3. Fine-tuning the pruned model.
4. Increasing the threshold.
5. Back to step 2.

After every iteration of pruning, you need to make two changes before you run next iteration: The first one is to increase the threshold in the main cfg file and the second one is to change the output file to avoid overwriting the old results.

Here is a sample for the main cfg file modification:

```
[pruning]
workspace=pruning
datacfg=pruning/voc.data
modelcfg=pruning/yolov3-voc.cfg
prunedcfg=pruning/yolov3-voc-prune.cfg
ana_out_file=pruning/ana.out
# change prune_out_weights to avoid overwriting old results
prune_out_weights=pruning/weights.prune.round2
criteria=0
```

```
kernel_batch=2
ignore_layer=80,81,92,93,104,105
yolov3=1
# change threshold from 0.005 to 0.01
threshold=0.01
```

After the modifications, use the new pruning main cfg file to run the prune command again and you can start a new iteration of pruning.

Model Transformation

After a few iterations of pruning, you get a model that is much smaller than its original size. Finally, you need to transform the pruned model, which has many zero channels, to a normal model.

Run the following command to transform model.

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

***Note*:** The transformation is only required after all iterations of pruning have been completed and you do not need to run the transform command between each iteration of pruning.

If you want to know the compression ratio or how many operations are in the pruned network, use the `stat` command.

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

The command outputs the following.

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

Model Evaluation

Use the standard way to test the transformed model. Change modelcfg in main cfg to transformed model.

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

Run the following command to generate detection output and further evaluate mAP using the tools from different datasets.

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

Model Conversion and Deployment

Because our hardware only supports the deployment of Caffe and TensorFlow models, it is necessary to convert a Darknet model to a Caffe model. An open source conversion tool is included in Caffe. Convert Darknet model to Caffe and then follow *Vitis AI User Guide* in the *Vitis AI User Documentation* (UG1431) to deploy to an FPGA.

vai_p_darknet Usage

The pruning tool includes seven commands shown in Table 1: vai_p_tensorflow Arguments. The `ana`, `prune`, `finetune`, and `transform` are main commands corresponding to the pruning process. The `stat`, `map`, and `valid` commands are auxiliary tools. The main cfg file is required for all the commands. A full list of options in the main cfg file is shown in Table 2: vai_p_caffe Arguments. The last column "Used in" in Table 2: vai_p_caffe Arguments shows this option is used in which commands. Use the first character to represent the commands. For example, the "workspace" option is used in all the commands, so "`APFTSMV`" is there. "is_yolov3" is used only in the `ana` and `prune` commands, so "`AP_____`" is in that column.

Table 4: **Seven Commands in vai_p_darknet**

Command	Description and Usage

ana	Analyze the model to find the pruning strategy. This command analyzes the input model ("modelcfg" in main cfg + weights in command line), generate a prunedcfg file and output ana results. The default output location is the "workspace" specified in cfg and the default file names are `{$prefix}-prune.cfg` and `ana.out`. Output file names can be changed by specifying "prunedcfg" and "ana_out_file" in cfg. See Table 2: vai_p_caffe Arguments for all the items in main cfg. Multiple GPUs can be used to accelerate the ana process by setting the GPUs indices.
	
	`Usage:`
	`./vai_p_darknet pruner ana cfg weights [-gpus gpu_ids]`
	`Example:`
	`./vai_p_darknet pruner ana pruning/cfg pruning/yolov3.weights -gpus 0,1,2,3`

prune	Prune the input model. This command prunes the input model ("modelcfg" in main cfg + weights in command line) according to the ana result by `ana` command and the settings in main cfg. The parameters include "criteria", "kernel_batch", "ignore_layer", "threshold" and "is_yolov3". See Table 2: vai_p_caffe Arguments for setting details.
	Generally, the `prunedcfg` file has been created in the ana step. If "prunedcfg" is not specified in main cfg, this command will generate one automatically to the default path. The output pruned weights file is defined by "prune_out_weights" in main cfg. If not defined, the default output is "weights.prune" in workspace.
	
	`Usage:`
	`./vai_p_darknet pruner prune cfg weights`
	`Example:`
	`./vai_p_darknet pruner prune pruning/cfg pruning/yolov3.weights -gpus 0,1,2,3`

Table 4: **Seven Commands in vai_p_darknet** *(cont'd)*

Command	Description and Usage

finetune	Fine-tune the pruned model to improve the performance of the model. This command finetunes the pruned model. It reads model description specified by "prunedcfg" in main cfg. For weights file, weights file in command line is the first priority. If not specified, the "prune_out_weights" in main cfg is used. This finetune command follows the standard Darknet training process and save model snapshots to "backup" directory by default.
	`Usage:`
	`./vai_p_darknet pruner finetune cfg [weights] [-gpus gpu_ids]`
	`Example:`
	`./vai_p_darknet pruner finetune pruning/cfg pruning/weights.prune -gpus 0,1,2,3`

transform	Transform the pruned model to a regular model. After channel pruning, there are many zeros in the pruned model ("prunedcfg" in main cfg + weights in command line). This transform command removes useless zeros and change the pruned model to a normal one. The output model cfg and model weights are specified by "transform_out_cfg" and "transform_out_weights" in main cfg. If not specified, "model-transform.cfg" and "weights.transform" are the default file names.
	`Usage:`
	`./vai_p_darknet pruner transform cfg weights`
	`Example:`
	`./vai_p_darknet pruner transform pruning/cfg backup/*_final.weights -gpus 0,1,2,3`

stat	Count how many floating-point operations are required for the model. Only a modelcfg is required for this command. In the command line, you can use main cfg with a "modelcfg" item like the other commands. Or you can directly use modelcfg in the command line.
	`Usage`
	`./vai_p_darknet pruner stat cfg`
	`Example:`
	`./vai_p_darknet pruner stat pruning/cfg`
	`./vai_p_darknet pruner stat pruning/yolov3.cfg`

map	Test model ("modelcfg" in main cfg + weights in command line) mAP using the built-in method. This mAP is used in our tool and it is not run in the standard way. Use "valid" command to generate detection results and calculate mAP using the standard python script.
	`Usage`
	`./vai_p_darknet pruner map cfg weights`
	`Example:`
	`./vai_p_darknet pruner map pruning/cfg backup/*_final.weights -gpus 0,1,2,3`

valid	Predict using specified model and output standard detection results. The results can be further used to evaluate mAP, using the tools provided by different datasets. This command is the same as "darknet detector valid" in the open source Darknet.
	`Usage`
	`./vai_p_darknet pruner valid cfg weights [-out outfile]`
	`Example:`
	`./vai_p_darknet pruner valid pruning/cfg backup/*_final.weights -gpus 0,1,2,3`

Table 5: **Full list of Options in Main cfg File**

Option	Description	Default Value	Active In
workspace	Workspace for pruning	"./pruning"	APFTSMV
datacfg	Data cfg file, same to standard darknet	"./pruning/voc.data"	A_F__MV

Table 5: **Full list of Options in Main cfg File** *(cont'd)*

Option	Description	Default Value	Active In
modelcfg	Model cfg file, describe the model structure	""	AP__SMV
prunedcfg	Model cfg file for pruned model, add some flags to the original modelcfg	""	APFT___
ana_out_file	ana output file	"./pruning/ana.out"	AP_____
criteria	Criteria for sorting during prune. 0: sort by L1 norm; 1: sort by L2 norm	0	AP_____
kernel_batch	Minimum number of channels to prune at a time	2	AP_____
ignore_layer	Layers id, separate by “,”, that should not be pruned	""	AP_____
is_yolov3	Flag for YOLOv3	1	AP_____
eval_images	Number of images are used to evaluate mAP in our built-in function. -1 means using the whole validation dataset	-1	A____M_
threshold	Threshold for pruning. Under the threshold, calculate the pruning rate of each layer. See example for more information.	0.005	_P_____
prune_out_file	Output file for pruned weights.	""	_PF____
snapshot	Number of iterations during finetuning after which a model can be saved.	4000	__F____
transform_out_cfg	Model cfg file for transformed model.	model-transform.cfg	___T___
transform_out_weights	Weights file for transformed model.	weights.transform	___T___

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 how we can use `vai_p_tensorflow` to prune it.

TensorFlow Low Level API

Download and Convert 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` supply function calls `get_init_data taking train_batch` and `test_batch` as arguments and returns an image, label tensors, and initializer operations for train data and test data respectively, which will now run in training and evaluating.

The `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
```

Build 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 MNIST image data as arguments and return a logits tensor. Function `model_fn` read an input data pipeline and returns a dictionary of evaluation metrics operations.

Model Building, Training and Evaluating

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 graph and train and evaluate model. It is a framework 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 specifying if the `test_only` argument is true.

Train 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, you get a trained checkpoint: `models/train/model.ckpt`.

Export 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}
```

Run 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
```

Prune 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
```

Finetune 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 "sparse training" mode, that is, the weights of pruned channels will be kept to 0 and will not be 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.

Finetune the pruned model is similar to train model from scratch 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, you need to 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 models/pruned.

Estimator

Build the CNN MNIST Classifier

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

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

# Imports
import numpy as np
import tensorflow as tf

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

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

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

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

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

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

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

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

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels,
logits=logits)

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

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

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

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

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

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

The `cnn_model_fn` function conforms to the interface expected by the Estimator API of TensorFlow. It takes MNIST feature data, labels and mode as arguments; create convolution and activation layers, and returns predictions, loss, and a training operation.

`train_input_fn` and `eval_input_fn` are functions that provide data to the network during training and evaluation respectively.

Train Baseline Model

To train the model by creating an Estimator and calling `train()` on it, create a file named `train.py` and add following codes:

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

# Imports
import numpy as np
import tensorflow as tf

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

def main(unused_argv):
  mnist_classifier = model_fn()

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

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

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

Run `train.py`.

```
$ python train.py
```

As the model trains, an output similar to the following is displayed:

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

You can get an accuracy of 96.84% on our test data set.

Export an Inference GraphDef File

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

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

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

FLAGS = tf.app.flags.FLAGS

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

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

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

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

Run Model Analysis

Now that you have prepared a trained checkpoint and a GraphDef file, you can start pruning.

Write some shell scripts to call `vai_p_tensorflow` functions.

```
WORKSPACE=./models

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

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

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

You have previously defined an operation of tf.metrics.accuracy named “accuracy” to calculate the accuracy of your model in `est_cnn.py`:

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

Use this operation to evaluate the performance of your model by setting --target=”accuracy”.

Prune the Model

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

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

Finetune 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
from est_cnn import cnn_model_fn, train_input_fn

# Imports
import numpy as np
import tensorflow as tf

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

FLAGS = tf.app.flags.FLAGS

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

def main(unused_argv):
  tf.set_pruning_mode()
  ws = tf.estimator.WarmStartSettings(
      ckpt_to_initialize_from=FLAGS.checkpoint_path)
  mnist_classifier = tf.estimator.Estimator(
```

```
      model_fn=cnn_model_fn, model_dir="./models/ft/", warm_start_from=ws)

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

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

Use tf.estimator.WarmStartSettings to load pruned checkpoint and finetune from it.

Run `ft.py` to finetune the pruned model:

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

The output log looks like the following:

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

As a final step, transform and freeze the finetuned model to get a dense model.

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

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

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

Finally, you have a frozen GraphDef file named `mninst.pb`.

VGG-16

This sample demonstrates how to run vai_p_tensorflow on real-world models. VGG (https://arxiv.org/abs/1409.1557) is a network for large-scale image recognition. This sample uses a pre-trained VGG-16 model from TensorFlow-Slim image classification model library.

Download the TensorFlow-Slim repository and its pre-trained VGG16 model:

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

Use the following instructions to prepare the ImageNet dataset:

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

Prepare graph evaluation script, named `vgg16_eval.py`:

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

import math
import tensorflow as tf

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

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

slim = tf.contrib.slim

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

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

  tf_global_step = slim.get_or_create_global_step()

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

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

  ##############################################################
  # Create a dataset provider that loads data from the dataset #
  ##############################################################
  provider = slim.dataset_data_provider.DatasetDataProvider(
      dataset,
      shuffle=False,
      common_queue_capacity=2 * batch_size,
      common_queue_min=batch_size)
  [image, label] = provider.get(['image', 'label'])
  label -= labels_offset

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

  eval_image_size = network_fn.default_image_size

  image = image_preprocessing_fn(image, eval_image_size, eval_image_size)

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

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

  variables_to_restore = slim.get_variables_to_restore()

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

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

Modify `models/research/slim/export_inference_graph.py` to get an output of a readable text file instead of binary file.

```
+ from google.protobuf import text_format

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

Export an inference graph:

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

Run model analysis:

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

In `vgg_16_eval.py`, a variable named batch_size with initial value of 100 was defined. There are 50,000 examples in the validation set of ImageNet, so the max_num_steps is set to 500 to ensure that all the examples in the validation set are tested in evaluation.

IMPORTANT! *The nodes with vgg_16/fc prefix affect the number of output labels of the network, so you need to exclude these codes to prevent shape mismatch with the dataset.*

Run model pruning:

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

Open `models/research/slim/train_image_classifier.py` and insert the following line in the beginning of the `main()` function:

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

Fine-tune the pruned model:

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

Get a dense checkpoint and freeze graph:

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

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

As the process of quantization are almost the same, only Resnet_v1_50 quantization is included in this user guide. For more information, see here.

PyTorch Examples

ResNet18

Write code for model training.

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

from torchvision.models.resnet import resnet18
```

```
from pytorch_nndct import Pruner
from pytorch_nndct import InputSpec

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

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

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

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

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

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

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

class ProgressMeter(object):

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

  def display(self, batch):
    entries = [self.prefix + self.batch_fmtstr.format(batch)]
```

```
    entries += [str(meter) for meter in self.meters]
    print('\t'.join(entries))

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

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

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

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

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

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

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

  # switch to train mode
  model.train()

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

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

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

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

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

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

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

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

  # switch to evaluate mode
  model.eval()

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

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

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

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

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

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

  return top1.avg, top5.avg
```

Define a function used for model analysis.

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

Create a resnet18 model and add pruning APIs to perform pruning.

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

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

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

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

  train_loader = torch.utils.data.DataLoader(
      train_dataset,
      batch_size=batch_size,
      shuffle=True,
      num_workers=workers,
      pin_memory=True)

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

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

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

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

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

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

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

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

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

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

Download pretrained ResNet18 model:

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

Prepare ImageNet dataset, see `http://image-net.org/download-images`.

Run the first round of pruning with model analysis:

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

Starting from the second round, pruning no longer requires a model analysis, you just need to increase the pruning ratio and use the sparse checkpoint saved from previous round as the pretrained weights.

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

Caffe Examples

VGG-16

Baseline Model

VGG is a network for large-scale image recognition. Refer to https://arxiv.org/abs/1409.1556 for the architecture of the VGG16.

Create a Configuration File

Create a file named `config.prototxt`:

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

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

rate: 0.1

pruner {
  method: REGULAR
}
```

Perform Model Analysis

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

Prune the Model

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

Finetune the Pruned Model

You can use the following solver settings to perform fine-tuning:

```
net: "vgg16/train_val.prototxt"
test_iter: 1250
test_interval: 1000
test_initialization: true
display: 100
average_loss: 100
base_lr: 0.004
lr_policy: "poly"
power: 1
gamma: 0.1
max_iter: 500000
momentum: 0.9
weight_decay: 0.0001
snapshot: 1000
snapshot_prefix: "vgg16/snapshot/res"
solver_mode: GPU
iter_size: 1
```

Use the following command to start finetuning:

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

Estimated time required: about 70 hours for 30 epochs using training set of ImageNet (ILSVRC2012) (1.2 million images, 4 x NVIDIA Tesla V100).

Get Final Output

After a few pruning iterations, a pruned model with only 33% of required operations relative to the baseline is obtained.

To finalize the model, run the following:

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

Pruning Results

- **Dataset:** ImageNet (ILSVRC2012)
- **Input Size:** 224 x 224
- **GPU Platform :** 4 x NVIDIA Tesla V100
- **FLOPs:** 30G
- **#Parameters:** 24M

Table 6: **Pruning Results of XFPN**

Round	FLOPs	Parameters	Top-1/Top-5 Accuracy
0	100%	100%	0.7096/0.8984
1	50%	57.3%	0.7020/0.8970
2	9.7%	35.8%	0.6912/0.8913

SSD (Single Shot Multibox Detector)

Baseline Model

SSD (https://arxiv.org/abs/1512.02325) is a deep neural network for detecting objects in images. This example uses the VGG16 as the backbone of the model.

Create a Configuration File

Create a file named `config.prototxt`:

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

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

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

rate: 0.15

pruner {
  method: REGULAR
```

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

Due to the nature of the SSD network, the number of filters in some convolution layers must be fixed and these layers need to be excluded from pruning. In the sample above, the top names of the layers to be excluded are listed within the "exclude" section. In general, if a convolution layer is directly calculated with the label, it cannot be pruned. For example, if the output of a convolution layer needs to be calculated with the label to get top-5 accuracy, then it must be excluded. Because the number of classes of label is fixed, it is necessary to ensure that the dimensions of the output of this convolution layer match the label.

Perform Model Analysis

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

Prune the Model

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

Finetune the Pruned Model

The following solver settings can be used as initial parameters for fine-tuning:

```
net: "float.prototxt"
test_iter: 229
test_interval: 500
base_lr: 0.001
display: 10
max_iter: 120000
lr_policy: "multistep"
gamma: 0.1
momentum: 0.9
weight_decay: 0.0005
```

```
snapshot: 500
snapshot_prefix: "SSD_"
solver_mode: GPU
device_id: 4
debug_info: false
snapshot_after_train: true
test_initialization: false
average_loss: 10
stepvalue: 80000
stepvalue: 100000
stepvalue: 120000
iter_size: 1
type: "SGD"
eval_type: "detection"
ap_version: "11point"
```

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

Estimated time required: about 50 hours for 650 epochs using Cityscapes training set (2975 images, 4 x NVIDIA Tesla V100).

Get Final Output

To get the finalized model, run the following:

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

Pruning Results

- **Dataset:** Cityscapes (four classes)
- **Input Size:** 500 x 500
- **GPU Platform :** 4 x NVIDIA Tesla V100
- **FLOPs:** 173G
- **#Parameters:** 24M

Table 7: **Pruning Results of SSD**

Round	FLOPs	Parameters	mAP
0	100%	100%	0.571
1	50%	29%	0.587
2	9.7%	9.7%	0.559

XFPN

Baseline Model

XFPN is a network for segmentation task. It is mainly composed of GoogleNet_v1 and FPN. The last few layers of the network are defined as follows:

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

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

    }
}
layer {
    name: "segmentation_eval_classIOU"
    type: "SegmentPixelIOU"
    bottom: "result"
    bottom: "label"
```

```
    top: "segmentation_eval"
    include {
        phase: TEST
    }
}
```

Create a Configuration File

Create a file named `config.prototxt`:

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

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

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

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

rate: 0.1

pruner {
    method: REGULAR

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

Perform Model Analysis

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

Prune the Model

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

Finetune the Pruned Model

The following solver settings can be used as initial parameters for fine-tuning:

```
net: "./workspace/segmentation/trainval.prototxt"
test_iter: 125
test_interval: 5000
test_initialization: true
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "multistep"
```

```
gamma: 0.1
stepvalue: 75000
stepvalue: 85000
display: 10
max_iter: 200000
snapshot: 5000
snapshot_prefix: "./workspace/segmentation/snapshots/"
solver_mode: GPU
iter_size: 1
average_loss: 20
eval_type: "segmentation"
classiou_class_num: 19
```

Use the following command to start finetuning:

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

Estimated time required: about 40 hours for 270 epochs using Cityscapes training set (2975 images, 4 x NVIDIA Tesla V100).

Get the Final Output

To get the finalized model, run:

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

Pruning Results

- **Dataset:** Cityscapes
- **Input Size:** 2048 x 1024
- **GPU Platform :** 4 x NVIDIA Tesla V100
- **FLOPs:** 136G

Table 8: **Pruning Results of XFPN**

Round	FLOPs	mIOU
0	100%	71.25
1	90%	70.88
2	83%	69.94

Darknet Examples

Generally, Darknet is used to prune YOLOov2 or YOLOv3. See Darknet Version - vai_p_darknet for a YOLOv3 example, and use it to prune a YOLOv3 network.

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)

Please Read: Important Legal Notices

Copyright