

0.1 Overview

- *Questions to be addressed*
- *Metrics to be measured - why*

This section will discuss the methodology used to search for lower latency models by tweaking pruning parameters.

0.2 Conceptual Process

- *Sensitivity analysis - filter/channel selection and layer interdependencies*
- *Filter pruning implementation - Theory*
- *Channel pruning implementation - Theory*
- *Retraining pruned model*

0.2.1 Sensitivity Analysis

0.2.2 Filter Pruning

This methodology selected the algorithm dubbed ‘L1RankedStructureParameterPruner’ by Distiller, this is based on the implementation described by Li et al in Pruning Filters for Efficient Convnets [liPruningFiltersEfficient2017]. This algorithm removes the filters that have the smallest impact on accuracy drop, the effect of filter pruning on network feature maps **Definition needed** is described in section TBD.

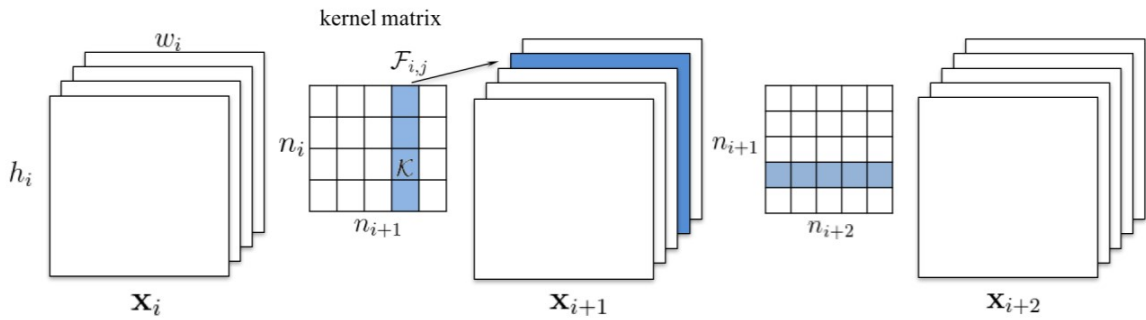


Figure 1: Pruning a filter results in removal of its corresponding feature map and related kernels in the next layer. [liPruningFiltersEfficient2017]

0.3 Filter and channel selection

Link back to selected model - concrete examples of process described in previous section

- *Filter selection (visual representation of filters)*
- *Channel selection (visual representation of channels)*
- *Discussion of pruning consequences (and recovery) - $\dot{\epsilon}$ top1/top5 before retraining and after*

0.4 Engineering/implementation details

- *High level overview of physical system - justify need for multiple training agents*
- *Pruning & retraining setup - Distiller (Pruning & training)*
- *Benchmarking setup - openvino + benchmark (getting latency/throughput)*
- *Data processing - wandb + data visualisation steps*

0.4.1 High level overview of system

Figure ?? shows how each system interacts in the pipeline, pruning is handled by the agent/s marked ‘Producer’, benchmarking is handled by the ‘Consumer’ agent, and the wandb system serves the next set of sweep parameters to each of the ‘Producer’ agents.

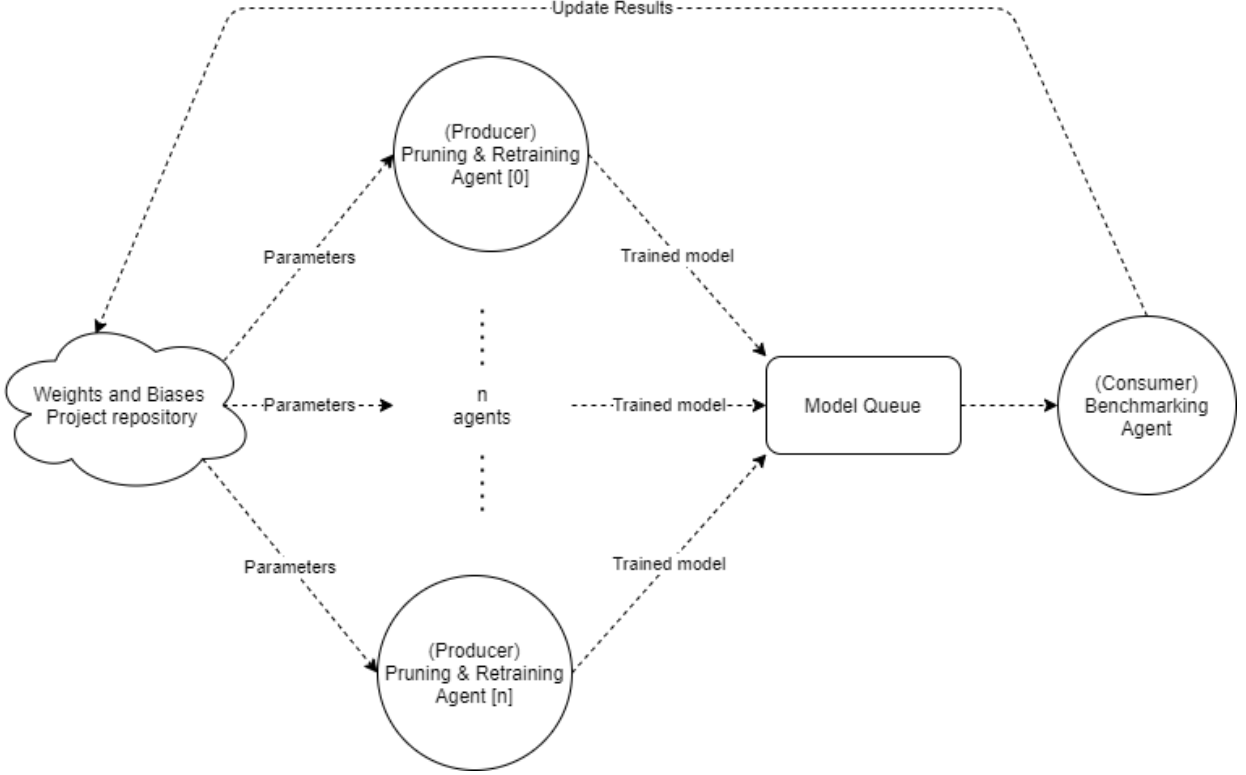


Figure 2: Diagram showing agent communication

When pruning begins, the producer agent requests the (initially random) pruning parameters from the Weights and Biases Project server, the producer then applies the pruning algorithm and begins retraining the model. Upon completion of retraining the model is exported into ONNX format and added to a queue for the consumer (the benchmarking agent) to benchmark and record the results, these results are then logged to weights and biases. As described in **(TBD)** the parameter importance and correlation with the target metric is re-computed each time results are logged this can help determine in what direction to tune the parameter settings to minimise (or maximise) the target metric.

The runtime of a full benchmark for one model on the NCS is usually at most 5 seconds, pruning and retraining the network however can take between 20 - 120 mins depending on the network size

and number of epochs. To improve the efficiency of the training we separated the benchmarking system (consumer) from the pruning and retraining systems (producer), this made it easy to add new pruning and benchmarking agents to a single experiment or run multiple experiments in parallel.

0.4.2 Defining parameters to prune

```

1      pruners:
2          layer_1_conv_pruner:
3              class: 'LiRankedStructureParameterPruner'
4              group_type: Filters
5              desired_sparsity: 0.9
6              weights: [
7                  module.layer1.0.conv1.weight,
8                  module.layer1.1.conv1.weight
9              ]
10     lr_schedulers:
11         exp_finetuning_lr:
12             class: ExponentialLR
13             gamma: 0.95
14
15     policies:
16         - pruner:
17             instance_name: layer_1_conv_pruner
18             epochs: [0]
19
20         - lr_scheduler:
21             instance_name: exp_finetuning_lr
22             starting_epoch: 10
23             ending_epoch: 300
24             frequency: 1

```

Figure 3: Example distiller schedule file, showing the pruning algorithm selected, and that algorithms parameters

Distiller uses a ‘compression schedule’ file to define the behaviour of the compression algorithms used, Figure ?? shows a simple example compression schedule, with a definition for a single ‘pruner’ instance (line 2 - `layer_1_conv_pruner`), a single ‘lr_scheduler’ instance (line 11 - `exp_finetuning_lr`), and their respective policies (explained below).

The pruning schedule is composed of lists of sections that define ‘pruners’, ‘lr-schedulers’, and

‘policies’. A ‘pruner’ defines a pruning algorithm and the layers on which that pruning algorithm will be applied, ‘LR-schedulers’ define the **learning-rate decay**(**Definition required**) algorithm. Finally each policy references an instance of a pruner or LR-scheduler, and controls when the respective algorithm will be applied, such as the start and end epoch, and the frequency of application.

The example compression schedule shown in Figure ?? provides instructions to Distiller to use the ‘L1RankedStructureParameterPruner’ algorithm (section **TBD**) to prune the weights in each of the convolutions visible inside the ‘weights’ array, in this case ‘group-type’ specifies filter pruning and ‘target_sparsity’ indicates how many tensors will try to remove (0.9 indicates the algorithm will attempt to remove 90% of the tensors), note that target sparsity should not be confused with an actual change in sparsity - it should be noted that filter and channel pruning will always result in a dense layer with an actual sparsity of 0.

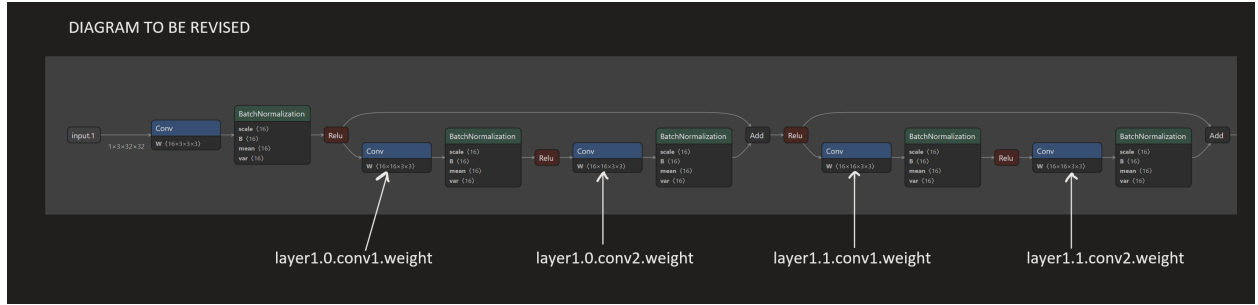


Figure 4: Resnet56 example showing first 4 convolutional layers with labels for the weights. (TODO: rescale and redraw to highlight pertinent information)

Each layer in the network is labelled (see figure ??), distiller uses these labels to identify which layers are being referenced by the compression schedule.

0.4.3 WandB API

Figure 5: WandB sweep configuration file

To explore the space of pruning parameter values the hyperparameter optimisation framework exposed by WandB called ‘Sweeps’ was leveraged. This involves writing a python script that can run the entire pipeline (pruning, training & benchmarking) and record the results, to accomplish this each sweep needs a configuration file (see Figure ??), table ?? shows a description of each key in the wandb configuration file with a summary of appropriate arguments.

Key	Description	Value
program	Script to be run	Path to script
method	Search strategy	grid, random, or bayse
metric	The metric to optimise	Name and direction of metric to optimise
parameters	The parameter bounds to search	Name and min/max or array of fixed values

Table 1: Configuration setting keys, descriptions and values

This configuration file tells wandb the names of the parameters to pass as arguments to the pipeline script with their expected value ranges, such as a list of strings or a min and max float/integer. The pipeline script that receives the arguments from wandb contains a mapping from the wandb arguments to the corresponding value in the distiller compression schedule. The pipeline then uses the values provided by wandb and writes out a new schedule that will be fed into distiller.

0.4.4 Benchmarking

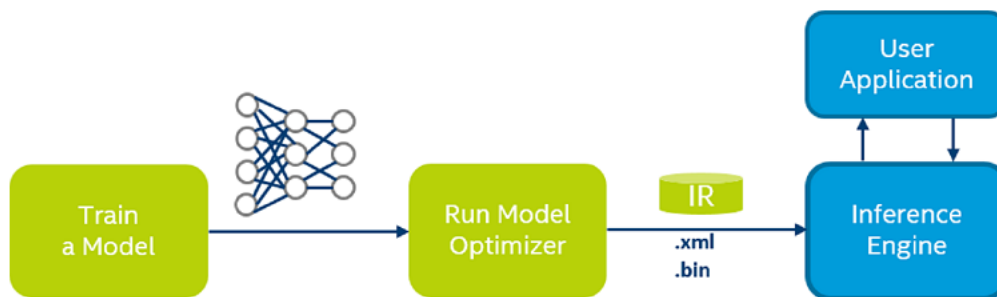


Figure 6: Workflow for deploying trained model onto NCS [ModelOptimizerDeveloper]

To pass the pruned and trained model to the Neural Compute stick OpenVino was used, it is a toolkit providing a high level **inference engine(Definition needed)** API, this facilitates the process of optimising the model for specialised hardware (in this case the NCS), and loading the optimised model into the hardware. OpenVino itself has a benchmarking tool that we leveraged to access detailed latency and throughput metrics; from end to end latency all the way down to the latency of each instruction used for inference on the VPU **link to table of operations and latency in appendix**. Before starting the benchmark we convert the ONNX model into an Intermediate Representation (IR) format by running to through the model optimizer, the IR can then be processed by the Inference Engine. Once the model is loaded to the VPU we load the images that will be used for benchmarking into the VPU memory. We observe three measurements for every model, the mean end-to-end latency (from loading an image into the model until getting a result), the actual inference latency or the latency to pass the data through the neural network excluding loading images into memory, and finally we also measure the throughput (the number of images(frames) that can be processed per second or FPS).

0.5 Experiment setup

- Wrapper on Distiller, reading schedule & parameterise elements
- WandB implementation, defining parameters to optimise
- communication between producer & consumer (redis - pub/sub)

- running benchmark and logging results

For the purpose of this experiment we chose to use the `L1RankedStructureParameterPruner` algorithm with filter pruning, .

We conducted three experiments using the Resnet56 model trained on the CIFAR10 dataset. These three experiemnts each used a different target metric: Latency, Top1, and a hybrid metric (see section (TBD)).

0.5.1 Schedules

Table ?? shows the labels and groupings of weights used for Filter pruning in the selected Resnet56 model. Note that only the first

Label	Weights
filter_pruner_layer_1	<ul style="list-style-type: none"> • module.layer1.0.conv1.weight • module.layer1.1.conv1.weight • module.layer1.2.conv1.weight • module.layer1.3.conv1.weight • module.layer1.4.conv1.weight • module.layer1.5.conv1.weight • module.layer1.6.conv1.weight • module.layer1.7.conv1.weight • module.layer1.8.conv1.weight
filter_pruner_layer_2	<ul style="list-style-type: none"> • module.layer2.1.conv1.weight • module.layer2.2.conv1.weight • module.layer2.3.conv1.weight • module.layer2.4.conv1.weight • module.layer2.6.conv1.weight • module.layer2.7.conv1.weight
filter_pruner_layer_3.1	<ul style="list-style-type: none"> • module.layer3.1.conv1.weight
filter_pruner_layer_3.2	<ul style="list-style-type: none"> • module.layer3.2.conv1.weight • module.layer3.3.conv1.weight • module.layer3.5.conv1.weight • module.layer3.6.conv1.weight • module.layer3.7.conv1.weight • module.layer3.8.conv1.weight

Table 2: Mapping of pruners to filter weights

0.5.2 Latency Target Metric

This experiment targeted pure inference latency, no information regarding accuracy was encoded in the optimisation metric.