

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 Experiment descisions (section to be renamed)

0.2.1 Filter Pruning algorithm

We selected the one-shot pruning algorithm dubbed ‘L1RankedStructureParameterPruner’ by Distiller, this is based on the algorithm described by Li et al. in Pruning Filters for Efficient Convnets [1]. We prune the filters that are expected to have the smallest impact on the accuracy of the network, this is determined by computing the sum of the absolute weights in each filter $\sum |\mathcal{F}_{i,j}|$, sorting them, and pruning the filters starting with the smallest sum values. Each filter that gets removed results in the corresponding feature map to be removed, along with its corresponding kernel in the next convolutional layer, see Figure 1.

Li et al [1] defines the procedure for pruning m filters from the i th convolutional layer as follows: Let n_i denote the number of input channels.

1. For each filter $\mathcal{F}_{i,j}$, calculate the sum of its absolute kernel weights $s_j = \sum_{l=1}^{n_i} |\mathcal{K}_l|$.
2. Sort the filters by s_j .
3. Prune m filters with the smallest sum values and their corresponding feature maps. The kernels in the next convolutional layer corresponding to the pruned feature maps are also removed.
4. A new kernel matrix is created for both the i th and $i + 1$ th layers, and the remaining kernel weights are copied to the new model.

Upon completion of pruning the filters we now retrain the network to regain lost accuracy, in general pruning the more resilient layers once and retraining can result in much of the lost accuracy to be regained. Once pruning is completed we compensate for the performance degradation by retraining the network,

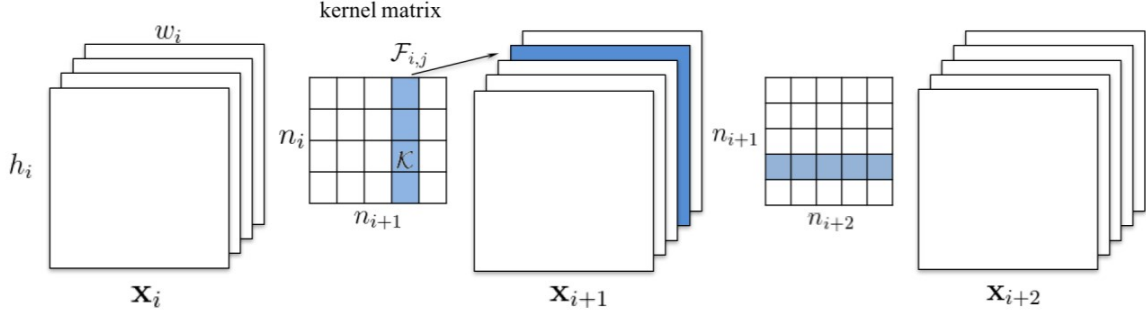


Figure 1: Pruning a filter results in removal of its corresponding feature map and related kernels in the next layer. [1]

0.2.2 Model 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) - $\bar{\epsilon}$ top1/top5 before retraining and after*

Pruning CNNs like AlexNet, or VGGNet is fairly straightforward, we can prune filters in any layer without worrying about damaging the fundamental structure of the network, however this is not the case with ResNets (short for Residual Networks), a very popular type of CNN that makes use of what is known as a ‘residual block’ which, from the perspective of pruning, adds additional interdependencies between layers.

ResNets were originally proposed to help address a training accuracy degradation problem that can occur in very deep networks, degradation (accuracy saturation) occurs when adding more layers to a suitably deep model leads to higher training error [2]. Today ResNets are commonly used for computer vision tasks, we selected a resnet neural network for this experiment because networks that are shallow enough not to benefit from residual blocks will have far fewer pruning parameters to manage, the practicality of automating these parameters increases with the size of a network.

The accuracy degradation problem with very deep CNNs is common enough that many new networks in research and production make use of what are known as residual blocks today.

We selected ResNet56 as the target network because it is one of the few networks with prebuilt ‘off-the-shelf’ schedules that also uses residual blocks, performing this experiment on networks using residual blocks is important because in general the necessity of using compression techniques such as pruning increases as networks get larger, residual blocks are important for larger networks for the reasons described above. The pre-tuned pruning schedule publicly available from Distiller has been hand built by an expert in the field, so it provides a good point of comparison. It is not a trivial task to improve on the preexisting hand built schedules manually without extensive understanding of layer sensitivities.

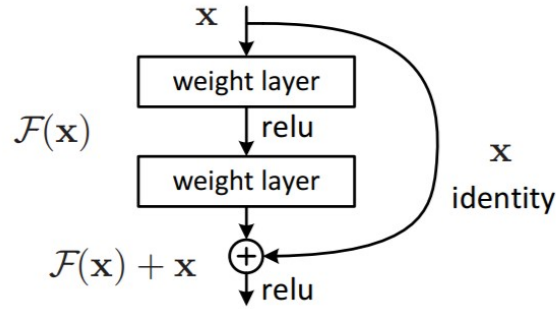


Figure 2: A residual block, note the identity feature map skips the weight layers, these are also known as ‘skip connections’.

0.2.3 Model Retraining

TBD

0.3 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*

0.3.1 High level overview of system

Figure 3 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.

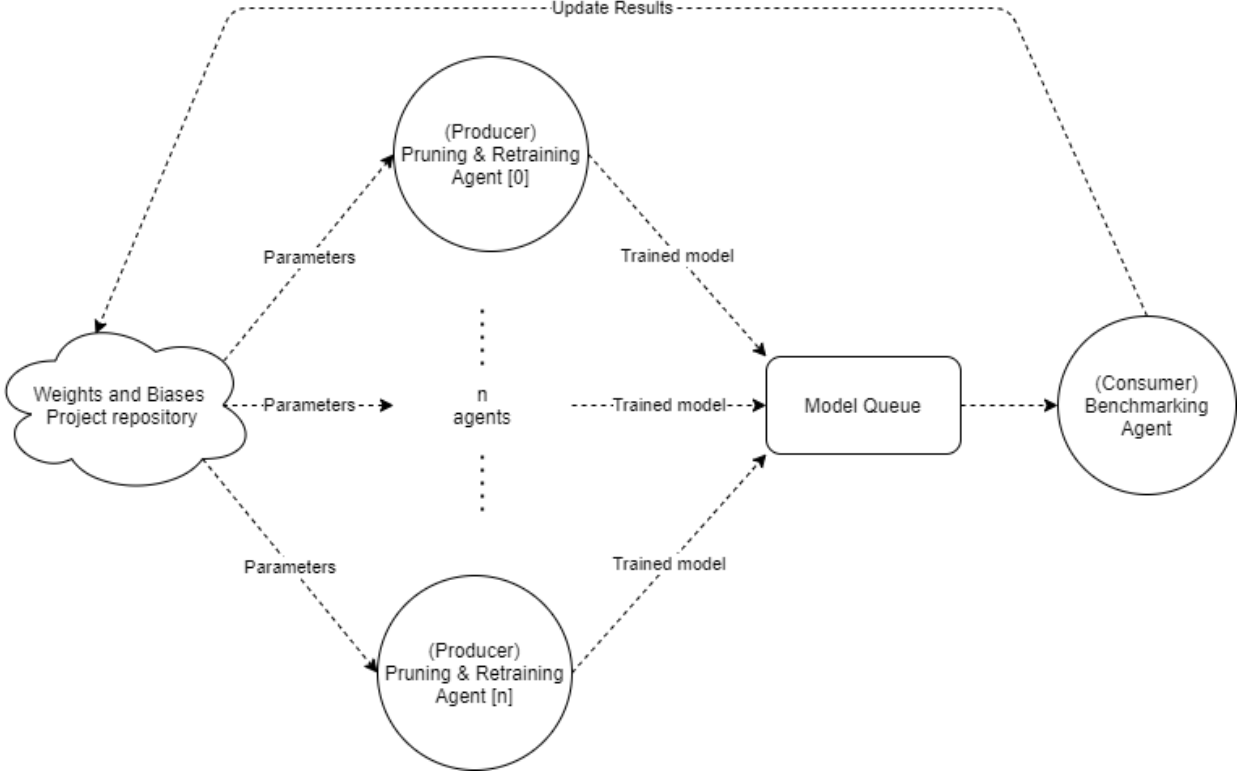


Figure 3: Diagram showing agent communication

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 additional pruning and retraining agents to a single experiment or run multiple experiments in parallel.

0.3.2 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 4: 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 4 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 describe ‘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 4 provides instructions to Distiller to use the ‘L1RankedStructureParameterPruner’ algorithm (as described in Section 0.2.1) to prune the weights in each of the convolutions described by the ‘weights’ array, in this case ‘group_type’ specifies filter pruning and ‘desired_sparsity’ indicates how many tensors it will aim to remove (0.9 indicates the algorithm will attempt to remove 90% of the tensors), desired sparsity should not be confused with an actual change in sparsity - note that filter and channel pruning will always result in a dense layer with an actual sparsity of 0 because this is a form of coarse-grained pruning (see section ??).

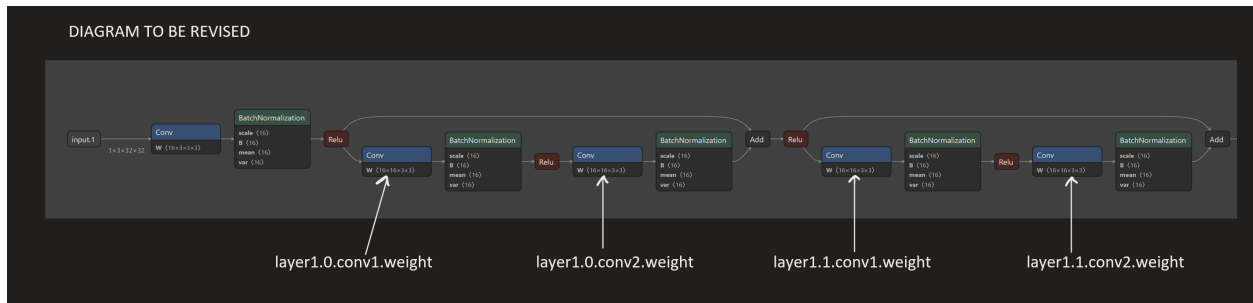


Figure 5: Resnet56 fragment showing first 2 residual block with the corresponding weights matrices labelled. (TODO: rescale and redraw to highlight pertinent information)

Each layer in the network is implicitly labelled (see figure 5), distiller uses these labels to identify which weight matrices are being referenced by the compression schedule.

0.3.3 WandB API

```

program: pipeline.py
method: bayes
metric:
    goal: minimize
    name: Latency
parameters:
    layer_1_conv_pruner_desired_sparsity:
        min: 0.01
        max: 0.99
    layer_1_conv_pruner_group_type:
        values: [Channels, Filters]

```

Figure 6: 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 6), table 1 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.

0.3.4 Benchmarking

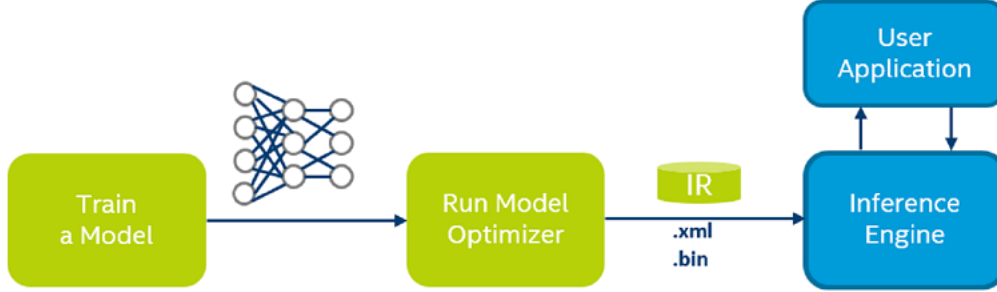


Figure 7: Workflow for deploying trained model onto NCS [3]

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 it through the model optimizer, the IR can then be read by the Inference Engine and loaded into VPU memory. Once the model is loaded into VPU we load the images that will be used for benchmarking into the VPU memory. We observe three measurements for every model, the end-to-end latency (from loading an image into the model until getting a result), the sum of latency for each instruction executed by the VPU once the image is loaded into memory, and finally we also measure the throughput (the number of images (frames) that can be processed per second or FPS).

0.4 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 experiments each used a different target metric: Latency, Top1, and a hybrid metric (see section (TBD)).

Observed metrics:

- **Latency** — Computed by calculating the sum of CPU time for hardware operations inside the NCS after the model and images have been loaded into memory.
- **Total_Latency** — Measures the full latency to perform inference on an image once a model is optimised and loaded into the NCS, including loading the image into the stick memory.
- **Throughput** — Shows the number of images per second that can be processed by the NCS.
- **Top1** — The % accuracy of the most likely class the model predicts.
- **Top5** — The % accuracy of the top 5 predicted classes.

0.4.1 Schedules

Table 2 shows how the weights are grouped and labeled for Filter pruning in the selected Resnet56 model, the 3 labelled pruners and their corresponding weights were used in all experiments. Layers with a similar degree of sensitivity to pruning are grouped together, layers that are omitted from the table have a much higher sensitivity to pruning and are not pruned at all, pruning more sensitive layers can result in a significantly higher rate of pruned neural networks that lose all predictive ability. Grouping layers in this way helps us avoid having to use 56 pruning parameters (one for each layer per residual block) and significantly reduces the complexity of the parameter search.

Note that only the first convolution in each residual block is being pruned, because the convolutions following this will also have the kernels removed following the removed feature maps.

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

0.4.2 Baseline data

For the purposes of all experiments we compare our results to two baseline sets of data, first the basic ResNets56 network with pretrained weights for CIFAR10, and second an ‘off-the-shelf’ version of ResNet56 with parameters hand picked by an expert in the field (**qualify this?**) [1].

0.4.3 Experiment 1: Rapid pruning, no retraining

0.4.4 Experiment 2: Target latency, with retraining

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

0.4.5 Experiment 3: Target Top1, with retraining