This Chapter will discuss the methodology used to automate the search for lower latency models by tweaking pruning parameters. Section **??** explains how the pipeline was implemented, the discrete parts and how they all fit together. Section **??** discusses the specifics of how we tested the system, the network model used and pruning algorithms selected.
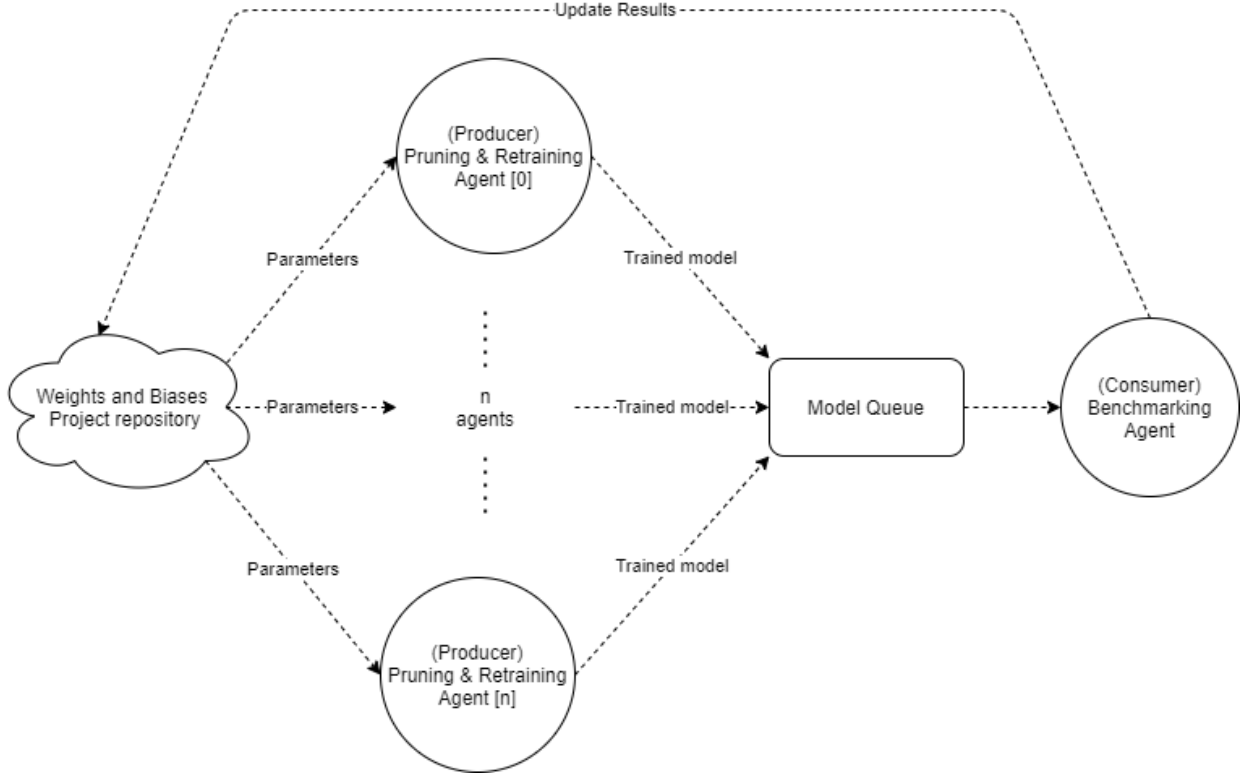

Figure 1: Diagram showing communication between discrete parts of the system.

0.1 Automated pruning pipeline

We constructed a pipeline to prune, retrain, benchmark, and record the data from each model, this pipeline consists of 4 separate elements; the systems to prune and retrain, a message queue (for this we used Redis), a benchmarking system, and finally the cloud service used to store the data. 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 Weights and Biases (WandB) system stores and provides the data used to compute each set of sweep parameters passed to the 'Producer' agents on request.

When pruning begins, the producer agent uses initially random pruning parameters, the pro-

ducer 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 WandB. The parameter importance and correlation with the target metric is re-computed on each iteration of the pipeline using results that are logged to WandB each time a benchmark is performed.

The runtime of a benchmark for a single model on the NCS is usually at most 5 seconds, retraining the network however can take between 20 - 120 mins depending on the network size and number of epochs, the one-shot pruning method utilised in this experiment usually takes less than 5 seconds. Because the training process is so slow 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. To make use of this pipeline 2 files must be provided by the user, a distiller schedule with a definition for which weights will be pruned, and a WandB configuration file which defines the type and ranges of the parameters it will seek to optimise.

0.1.1 Defining parameters to prune

```
1   pruners:
2       layer_1_conv_pruner:
3           class: 'L1RankedStructureParameterPruner'
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
```
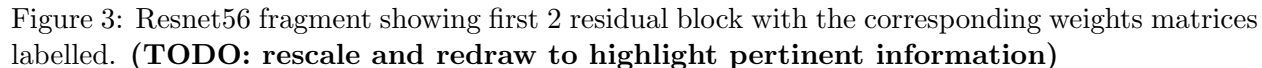
Figure 2: Example distiller schedule file, showing the pruning algorithm selected, and that algorithms parameters

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_fine_tuning_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 **??** provides instructions to Distiller to use the 'L1RankedStructureParameterPruner' algorithm (as described in Section **??**) 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 3: Resnet56 fragment showing first 2 residual block with the corresponding weights matrices labelled. **(TODO: rescale and redraw to highlight pertinent information)**

Each grouping of weights in the network has labels (see figure **??**), distiller uses these labels to identify which weight matrices are being referenced by the compression schedule. Lines 7 and 8 in the schedule in Figure **??** reference the weights we wish to prune from the model in Figure **??**.

0.1.2 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 4: WandB sweep configuration file

To explore the space of pruning parameter values the hyperparameter optimisation framework exposed by 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 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

The sweep 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 number. The pipeline script that receives the arguments from WandB contains a mapping from the WandB arguments to a corresponding value in a distiller compression schedule. This is accomplished by parsing a base schedule file and identifying which values will be changed, then a

new schedule is written with the parameters from WandB, this new schedule is then provided to Distiller as the compression schedule to use.
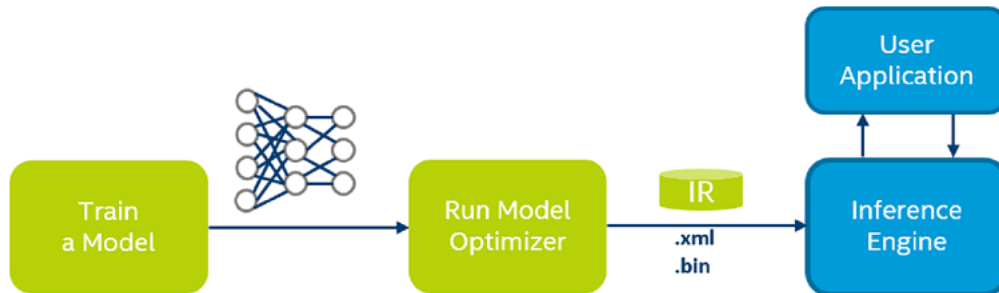
0.1.3   Benchmarking

Figure 5: 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 example table of HW 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 optimiser, the IR is then read by the Inference Engine and loaded into VPU memory. Once the model is ready 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).

0.2 Experiment Design

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 [**liPruningFiltersEfficient2017**]. 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 $\Sigma |\mathcal{F}_{i,j}|$, sorting them, and pruning the filters starting with the smallest sum values. Each filter that gets removed causes the corresponding feature map to be removed, along with its corresponding kernel in the next convolutional layer, see Figure **??**.

Li et al [**liPruningFiltersEfficient2017**] defines the procedure for pruning m filters from the ith 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 $s_j = \Sigma_{l=1}^{n_i} \Sigma |\mathcal{K}_l|$.

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 ith 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,
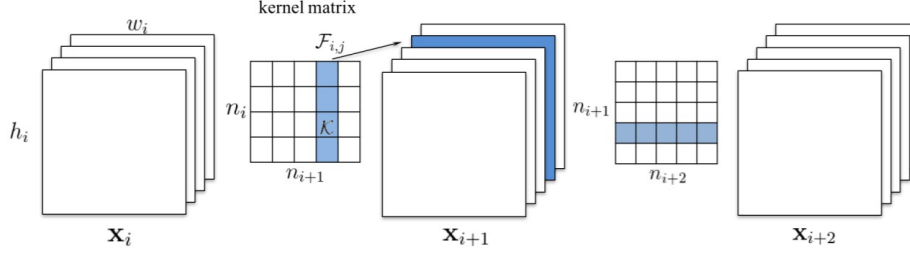
Figure 6: Pruning a filter results in removal of its corresponding feature map and related kernels in the next layer. [**liPruningFiltersEfficient2017**]**Include annotations for the feature map and kernel**

0.2.2 Model Selection

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' (Figure **??** shows a residual block) which, from the perspective of pruning, adds additional interdependencies between layers.

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 the necessity of using compression techniques such as pruning increases as networks get larger, these residual blocks are very common in very large networks today.

The pre-tuned pruning schedule publicly available from Distiller has been hand built by an expert in the field, providing a solid baseline for comparison. It is not a trivial task to improve on the pre-existing hand built schedules manually without extensive understanding of layer sensitivities.
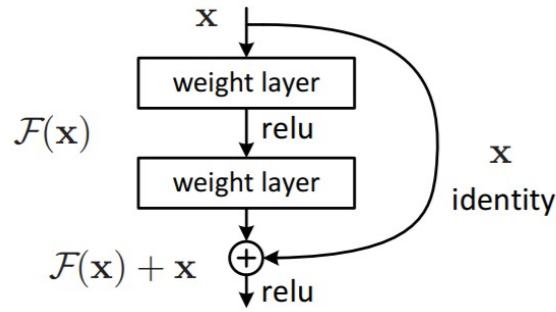
Figure 7: A residual block, note the identity feature map skips the weight layers, this is also known as a 'skip connection'.

Discuss further how residual blocks effect pruning. ResNets were originally proposed to help address a training accuracy degradation problem that can occur in very deep networks, degradation (or accuracy saturation) occurs when adding more layers to a suitably deep model leads to higher training error [**heDeepResidualLearning2016**]. The accuracy degradation problem with very deep CNNs is common enough that many new deep networks in research and production make use of them today.

0.2.3 Optimisation Process

In general machine learning training is computationally expensive, execution times can commonly be hours or even days, when the cost of retraining is so high it is easy to see why we would want to exchange some extra work up front for finding a good next search point rather than simply opting for the more commonly used hill climbing algorithms that rely more on the local gradient. One of the key ideas here is to use all information available from all previous evaluations of the search space, allowing us to model the plausibility of future search points, Bayesian optimisation processes are designed to construct this model by evaluating this prior knowledge about known properties, optimisation processes are considered to be some of the most effective optimisation methods in terms of the number of evaluations required [**brochuTutorialBayesianOptimization2010**].

The specific flavour of Bayesian optimisation process used in this dissertation is based on the method described by Snoek et al. [**snoekPracticalBayesianOptimization2012**] where statistical models are used to find search points heuristically using a Monte Carlo estimate of the expected improvement.

Due to the stochastic nature of the pruning methods we have utilised there is a considerable volume of noise in our data because when we perform optimisation we tune all pruners in parallel, each pruner will have a different degree of impact on the target metric. This results in a very noisy search surface (See Figure ??), this noise can be challenging to overcome and is the primary reason we selected this optimisation approach.
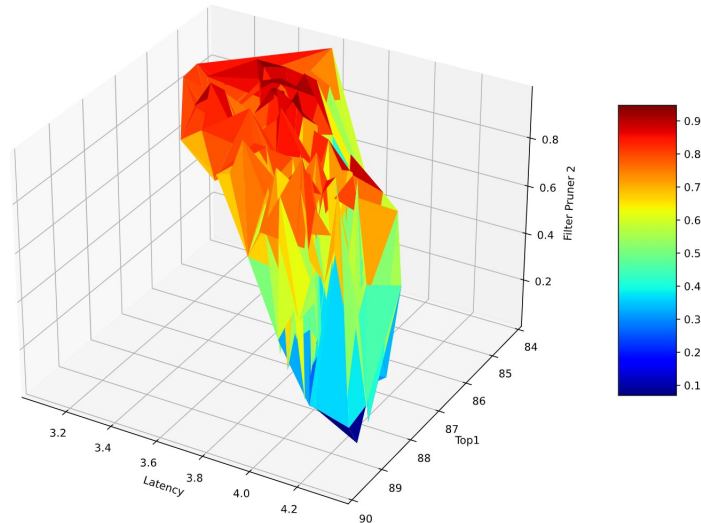

Figure 8: A single pruner search surface showing Latency and Top1. The very spikes would be challenging for many hill climbing algorithms, due to the depth of local maxima.

0.2.4 Experiment metrics and parameters

We conducted three experiments using a Resnet56 model with pretrained weights for the CIFAR10 dataset. This section describes the parameters and metrics that will be either set or observed during the experiments.

Parameters:

- **Desired Sparsity** — Specified on a per-pruner basis, this lets us specify to the pruning algorithm what proportion of the weights to try and prune (See Section ?? for more details).

- **Epochs** — The number of epochs used for retraining, one epoch refers to a single cycle of training through the entire training dataset.

- **Learning Rate** — Used to scale how much the retraining process will adjust the weights during each weight update.

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, this is more indicative of real world requirements.

- **Throughput** — Shows the number of images per second that can be processed by the NCS (Frames Per Second).

- **Top1** — The % accuracy of the first class predicted by the model.

- **Top5** — The % accuracy of the first 5 classes the model predicted for a given image.

0.2.5 Schedule

Table **??** shows how the weights are grouped and labelled for pruning in the selected Resnet56 model, the 4 labelled pruners and their corresponding weights were used in all resnet56 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 probability that pruned neural networks that lose all predictive ability (in other words the network will predict a single class 100% of the time). 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 (denoted by 'conv1' inside the weight name), because the convolutions following this will also have the kernels removed following the removed feature maps (See Section **??**).

Label	Weights
filter_pruner_layer_1	• 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	• 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	• module.layer3.1.conv1.weight
filter_pruner_layer_3.2	• 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 labels to resnet56 weights

0.2.6 Passing pruned/trained networks to benchmark

Discuss reading/writing yaml files, outputting .onnx files, redis to pass messages between agents

0.2.7 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 no pruning, and second an 'off-the-shelf' version of ResNet56 with pruning parameters hand-picked by the researcher responsible for development of Distiller (thus it is highly likely to be used as a starting point for new users to distiller) [**liPruningFiltersEfficient2017**].

Model	Top1	Top5	Throughput (FPS)	Latency (ms)	Total Latency (ms)
Baseline - no pruning	92.58	99.78	294.08	4.375	13.19
Off the shelf - no retraining	11.19	51.02	303.98	3.947	12.89
Off the shelf - retrained	87.72	99.47	305.27	3.88	12.95

0.2.8 Experiment 1: Rapid pruning, no retraining

Targeting the weights described in table **??** (the full schedule is listed in appendix **??**) we repeatedly pruned networks without performing any training to regain accuracy, we set the target metric to minimize Latency, the number of epochs to 0, and the learning rate to 0.1. The goal at this stage was to observe any reduction in latency, with the added benefit of allowing us to gather a large volume of data very quickly. Figure **??** shows the WandB configuration file used for this part of the experiment, this configuration seeks to optimise the desired sparsity settings for each of the 4 pruners.

```
program: pipeline.py
method: bayes
metric:
  goal: minimize
  name: Latency
parameters:
  filter_pruner_layer_1:
    min: 0.0
    max: 0.99
  filter_pruner_layer_2:
    min: 0.0
    max: 0.99
  filter_pruner_layer_3.1:
    min: 0.0
    max: 0.99
  filter_pruner_layer_3.2:
    min: 0.0
    max: 0.99
```

Figure 9: Targeting Latency sweep config

0.2.9 Experiment 2: Target latency, with retraining

Using the same WandB configuration file as in the first experiment (Figure ??), we again target minimizing latency but this time retrain for a fixed 70 epochs, and set the learning rate to 0.1. The purpose was to observe how well the network would recover accuracy when we focus purely on Latency.

0.2.10 Experiment 3: Target Top1, with retraining

During the third experimental stage we tweaked the configuration file (see Figure ??) to specify a new target metric: maximise Top1 . Similarly to Experiment 2 we kept the number of epochs fixed at 70 and the learning rate at 0.1. During this experiment we were interested in observing how the optimisation processes effected Top1, and if we could improve on our best Top1 score from the second experiment.

```
program: pipeline.py
method: bayes
metric:
  goal: maximize
  name: Top1
parameters:
  filter_pruner_layer_1:
    min: 0.0
    max: 0.99
  filter_pruner_layer_2:
    min: 0.0
    max: 0.99
  filter_pruner_layer_3.1:
    min: 0.0
    max: 0.99
  filter_pruner_layer_3.2:
    min: 0.0
    max: 0.99
```

Figure 10: Targeting Latency sweep config

0.2.11 Experiment 4: Improving on the fastest latency settings

For this final experiment we set the pruner parameters to the configuration with which we observed the lowest latency, and we sought to maximise Top1 by searching over the learning rate and number of epochs.

Figure 11: This configuration fixes the previously found lowest latency set of parameters, we search instead for an improvement in epochs and learning rate.