Friedrich-Alexander-Universität
Technische Fakultät

Deep Learning Model Optimization
Quantization and Pruning for Enhanced Efficiency and Performance

Abdullah, Muhammad
Kakumanu, Ram Saran

DNN Model Optimization

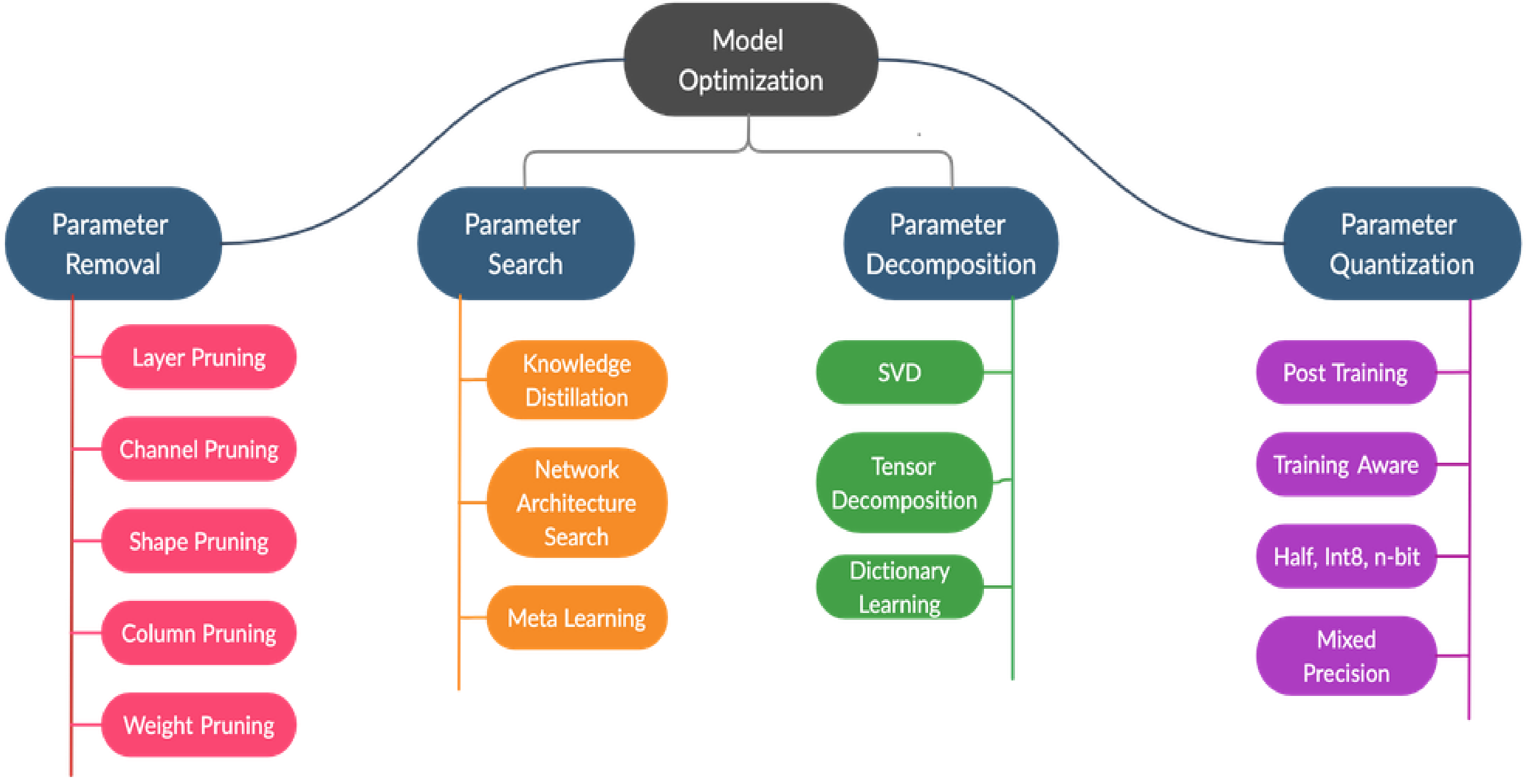
Deep Learning Quantization: Enhancing Efficiency and Performance

01 Model Optimization: Quantization, Pruning and Approximate Computing

02 CIFAR datasets and ResNet Models

03 Quantization Experiments and Results

04 Pruning Experiments and Results



https://www.edge-ai-vision.com/2020/09/dnn-model-optimization-series-part-i-whats-the-drill/

DNN Quantization

Experimental Procedure to Understand the Effect of Quantization

01 Base Model Training

02 Dynamic Quantization

03 Post-training Quantization

05 Quantization-aware Training

06 Capture and Compare Results

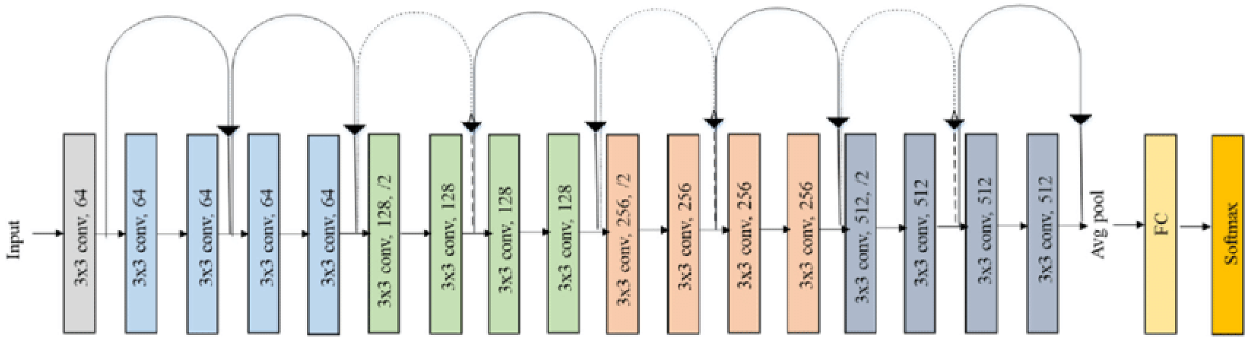
Where bins for conversion of FP32 to Int8 are defined

DNN Quantization

ResNet and CIFAR: The Building of Base Model

What are ResNet model?

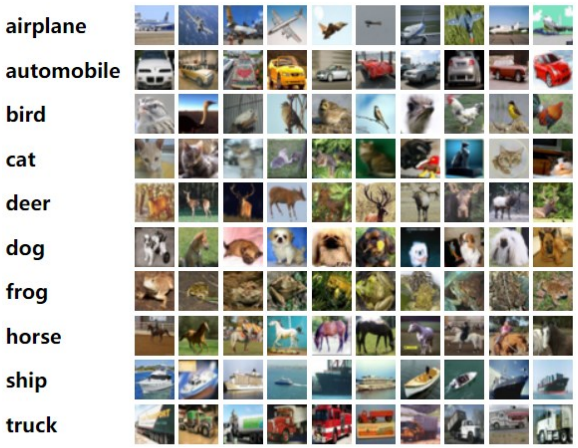
- DNN architecture widely used in computer vision tasks
- Uses concept of residual connections
- Addresses vanishing gradient problem



https://www.researchgate.net/figure/Original-ResNet-18-Architecture_fig1_336642248

What are CIFAR dataset?

- Benchmark DL models
- 60,000 32x32 color images
  - CIFAR10: 10 classes, with 6,000 images per class
  - CIFAR100: 100 classes, with 600 images per class



https://www.cs.toronto.edu/~kriz/cifar.html

DNN Quantization

Various Strategies

Dynamic Quantization

- Float32 x “Scalar Factor” = Rounded to nearest “Int8” → Dynamically at runtime
- Weights are known → Quantized before using
- Activations → Quantized on the fly (Before using in Activation Layers)
- Scaling factor adjusted based on input data
- Least performant quantization technique

```
import torch.quantization
quantized_model = torch.quantization.quantize_dynamic(model,
                        {torch.nn.Linear}, dtype=torch.qint8)
```

DNN Quantization

Various Strategies

Post Training Quantization

- Int8 Memory Access +
- Fine-tuning step between Model Completion and Inference
- Feed data batches → Distributions of different Activations → “Determines the bins”
- Right technique for medium-to-large-sized models

```
model_fp32 = CustomModelclass()
model_fp32.eval()

model_fp32.qconfig = torch.quantization.get_default_qconfig('fbgemm')

# fusing different layers into one
model_fp32_fused = torch.quantization.fuse_modules(
    model_fp32, [['conv', 'relu']])

# inserting observer module
model_fp32_prepared = torch.quantization.prepare(model_fp32_fused)

# quantization algorithm calibration using some data
model_fp32_prepared(input_fp32_model)

model_int8 = torch.quantization.convert(model_fp32_prepared)
output = model_int8(input_fp32)
```

DNN Quantization

Various Strategies

Quantization Aware Training

- Int8 mimic FP32 → "FakeQuantile"
- Training → FP32 only
- Best performance when compared to the other two methods
- Increased training time

```
model_fp32.train()
model_fp32.qconfig = torch.quantization.get_default_qat_qconfig('fbgemm')

model_fp32_fused = torch.quantization.fuse_modules(model_fp32,
                                                  [['conv', 'bn', 'relu']])
model_fp32_prepared = torch.quantization.prepare_qat(model_fp32_fused)

# calibration
training_loop(model_fp32_prepared)

model_fp32_prepared.eval()
model_int8 = torch.quantization.convert(model_fp32_prepared)
```

DNN Quantization

Evaluation Metrics

Accuracy

- Measure of correctness
- Ratio of correct predictions
- Indicates model performance

Inference Time

- Time taken to make predictions
- Measures runtime efficiency
- Need for real-time applications

Training Time

- Duration required to train a model
- Determines practicality of model
- Parameters influence performance

Model Size

- Amount of memory
- Occupied by the parameters
- Measured in bytes or megabytes

Power Consumption

- Amount of electrical consumption during operation
- Determines energy efficiency of model

Quantization Error

- Discrepancy from original model
- Arises due to the loss of precision
- Minimized to maintain accuracy

DNN Quantization

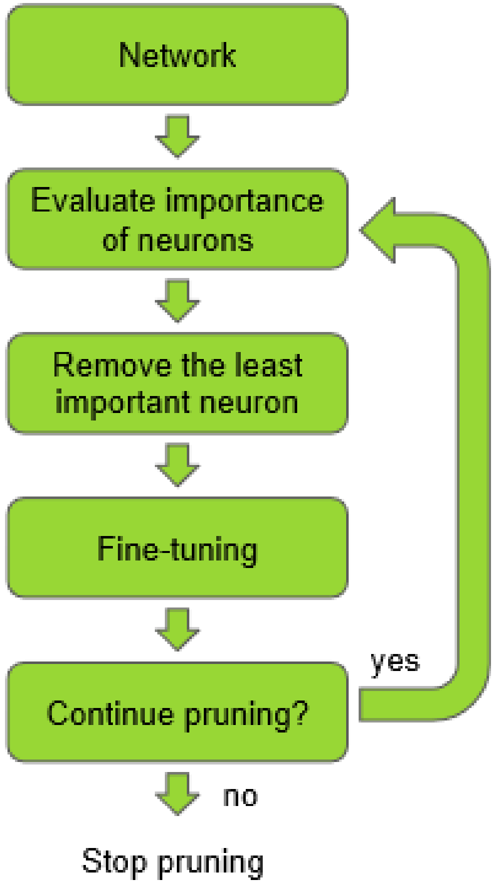
Results: ResNet18 – CIFAR10 vs. CIFAR100

ResNet18 - Metrics	Base Model		Dynamic Quantization		Post Training Quantization		Quantization Aware Training	
	CIFAR10	CIFAR100	CIFAR10	CIFAR100	CIFAR10	CIFAR100	CIFAR10	CIFAR100
Accuracy	**74.00%**	42.79%	**69.35%**	35.36%	**70.89%**	37.63%	**74.63%**	42.79%
Model Size (in MB)	**42.729**	42.905	**10.787**	10.834	**10.704**	10.749	**42.852**	43.032
Training Time (in s)	1952.085	1866.672	1952.08	1866.672	1952.08	1866.672	3286.48	3307.339
Inference Time (in ms)	**22.447**	22.161	**10.933**	27.890	**10.546**	10.173	**10.486**	10.011
Quantization Error	NA	NA	0.790	0.976	0.791	0.975	0.253	0.572

- Trade-off between “model size and inference time” with accuracy
- As complexity of model/data increases – Quantization affects the Accuracy

DNN Pruning

Introduction: Sparsity in DNN



https://arxiv.org/pdf/1611.06440.pdf

DNN Pruning

Terminology

Local Pruning

- Prune each layer/specified layer by a certain pruning ratio

Global Pruning

- Prune weights globally. Some layers can have very high pruning ratio and some can have very low

One-Shot Pruning

- Prune weights according pruning ratio all at once

Iterative Pruning

- Prune progressively in iterations until reaching given pruning ration

Unstructured Pruning

- Remove connections without any pattern or constraints (Sparsity)

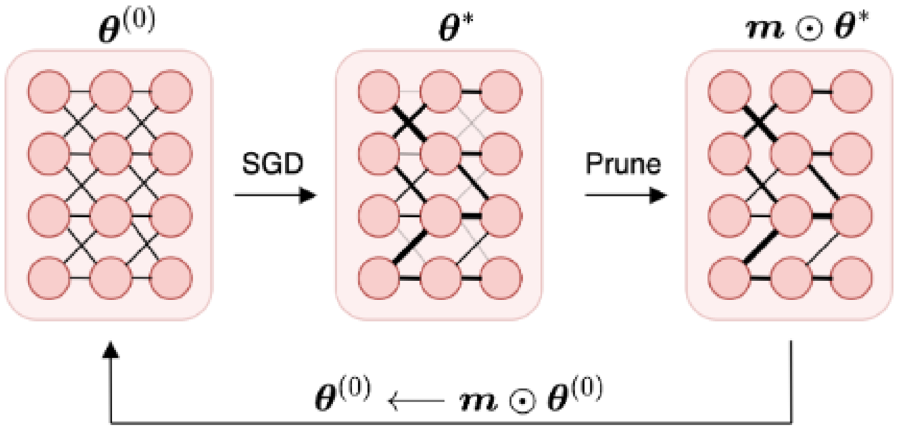
Structured Pruning

- Remove entire structures or groups of connections while maintaining a certain pattern or structure.

DNN Pruning

Lottery Ticket Hypothesis[2]

- Sparse architectures after pruning are difficult to train from the start
- Dense, randomly initialized networks contain subnetworks which can achieve similar performance
- Copy weights of subnetworks from original weights (winning lottery ticket) after pruning
- Winning tickets learn faster than original network
- Prune FC layers (MLP) and conv layers (Vgg, Resnet)



Source: https://www.researchgate.net/publication/368753812_Random_Teachers_are_Good_Teachers

DNN Pruning

Pruning in Pytorch[1]

```python
from torchvision.models import LeNet
from torch.nn.utils import prune

# Load Network
model = LeNet()

# Select layer you want to Prune
module = model.conv1

# Check current parameters for this layer
print(list(module.named_parameters()))

# Check buffer parameters for this layer
print(list(module.named_buffers()))

# Prune the layer by randomly making 30% weights zero
prune.random_unstructured(module, name="weight", amount=0.3)

# Pruning mask is stored in buffers names as 'weight_mask'
print(list(module.named_buffers()))

# A forward prehook is created
print(module._forward_pre_hooks)

# This is new pruned weights
print(module.weight)
```

[('weight', Parameter containing:

tensor([…],

device='cuda:0’, requires_grad=True)),

('bias', Parameter containing:

tensor([…],

device='cuda:0’, requires_grad=True))]

DNN Pruning

Pruning in Pytorch[1]

```python
from torchvision.models import LeNet
from torch.nn.utils import prune

# Load Network
model = LeNet()

# Select layer you want to Prune
module = model.conv1

# Check current parameters for this layer
print(list(module.named_parameters()))

# Check buffer parameters for this layer
print(list(module.named_buffers()))

# Prune the layer by randomly making 30% weights zero
prune.random_unstructured(module, name="weight", amount=0.3)

# Pruning mask is stored in buffers names as 'weight_mask'
print(list(module.named_buffers()))

# A forward prehook is created
print(module._forward_pre_hooks)

# This is new pruned weights
print(module.weight)
```

[('weight_mask',

tensor([[1, 0, 1],…],

device='cuda:0'))]

DNN Pruning

Pruning in Pytorch[1]

```python
from torchvision.models import LeNet
from torch.nn.utils import prune

# Load Network
model = LeNet()

# Select layer you want to Prune
module = model.conv1

# Check current parameters for this layer
print(list(module.named_parameters()))

# Check buffer parameters for this layer
print(list(module.named_buffers()))

# Prune the layer by randomly making 30% weights zero
prune.random_unstructured(module, name="weight", amount=0.3)

# Pruning mask is stored in buffers names as 'weight_mask'
print(list(module.named_buffers()))

# A forward prehook is created
print(module._forward_pre_hooks)

# This is new pruned weights
print(module.weight)

```

[OrderedDict(

[(0, <torch.nn.utils.prune.RandomUnstructured

object at 0x7f0749753c70>)])]

DNN Pruning

Pruning in Pytorch[1]

```python
from torchvision.models import LeNet
from torch.nn.utils import prune

# Load Network
model = LeNet()

# Select layer you want to Prune
module = model.conv1

# Check current parameters for this layer
print(list(module.named_parameters()))

# Check buffer parameters for this layer
print(list(module.named_buffers()))

# Prune the layer by randomly making 30% weights zero
prune.random_unstructured(module, name="weight", amount=0.3)

# Pruning mask is stored in buffers names as 'weight_mask'
print(list(module.named_buffers()))

# A forward prehook is created
print(module._forward_pre_hooks)

# This is new pruned weights
print(module.weight)

```

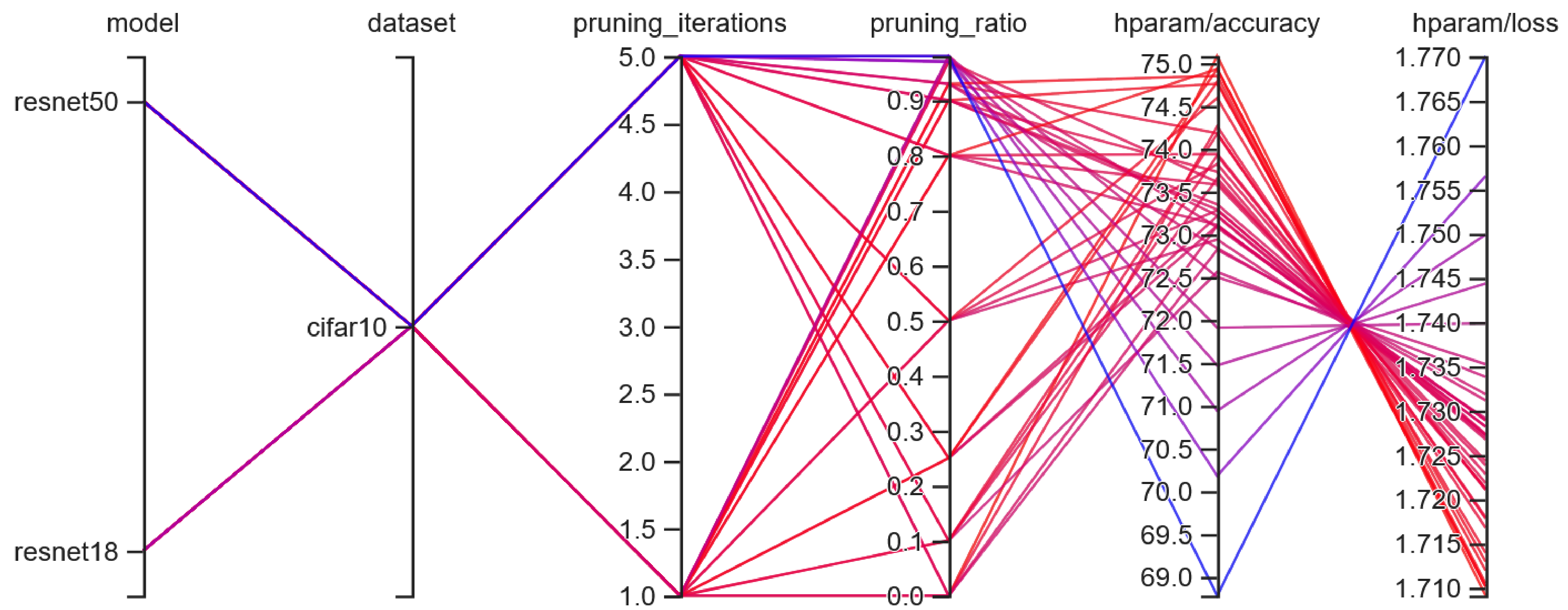
tensor([…],

device='cuda:0', grad_fn=<MulBackward0>)

DNN Pruning

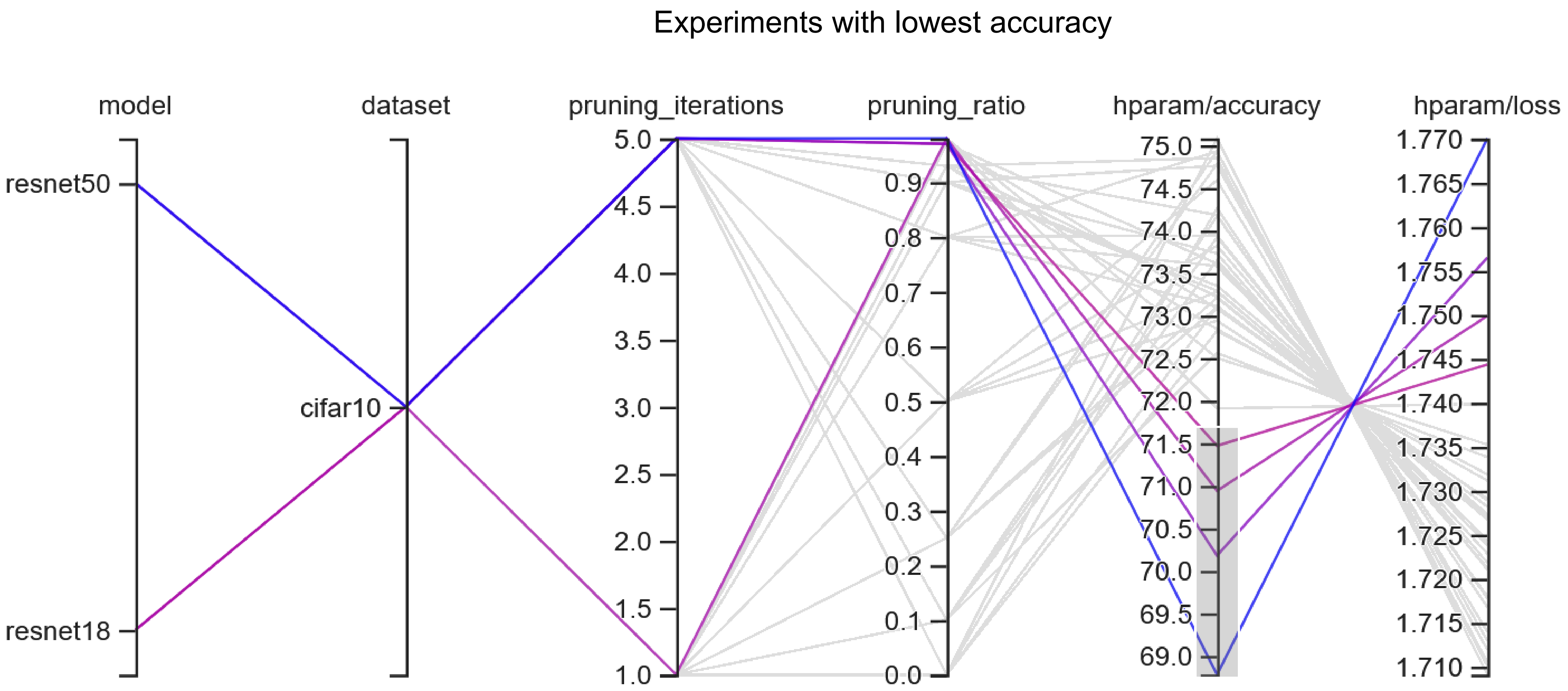
Results and Discussion

Parallel Coordinates View[4]



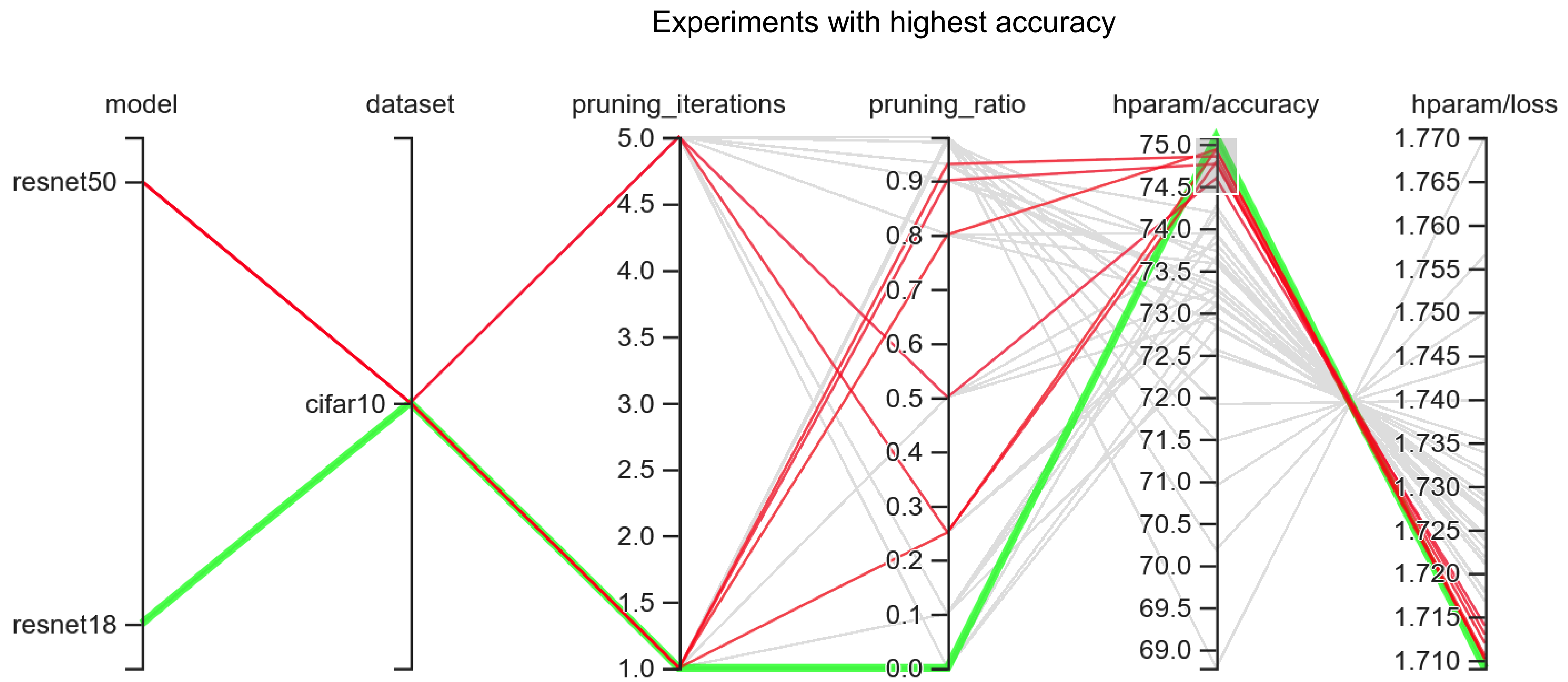
DNN Pruning

Results and Discussion



DNN Pruning

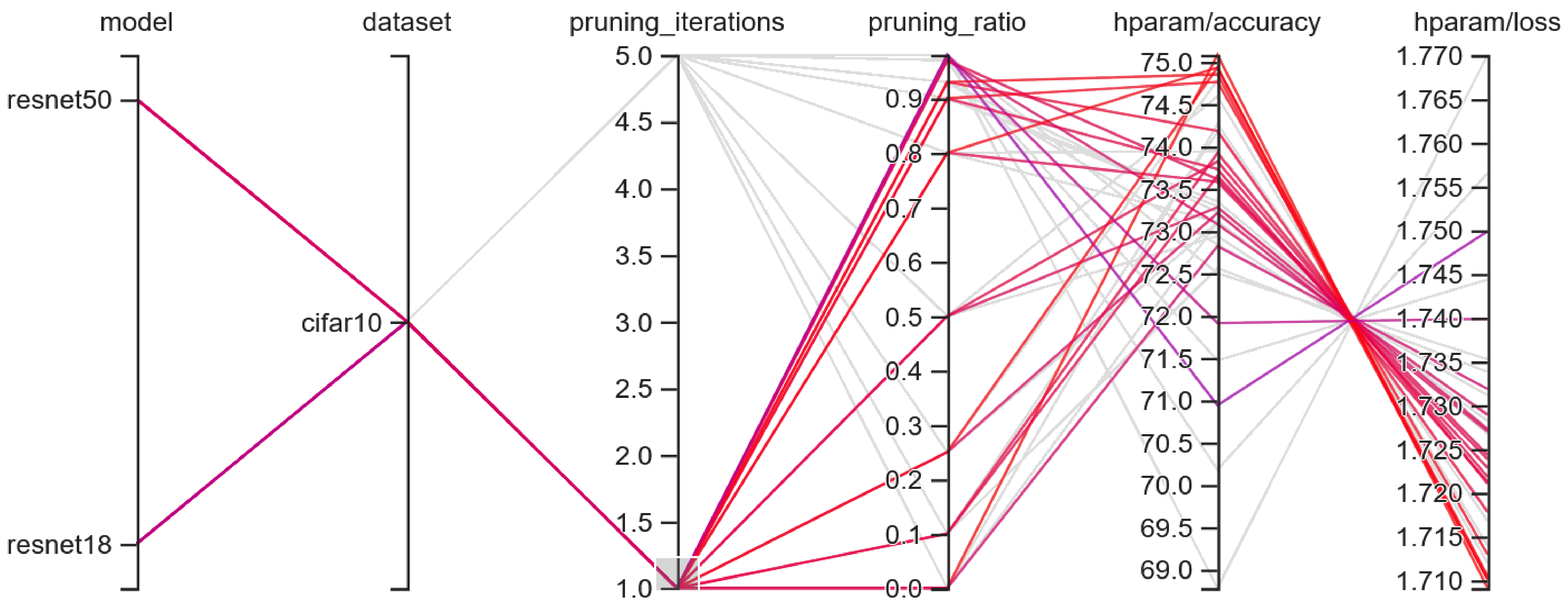
Results and Discussion



DNN Pruning

Results and Discussion

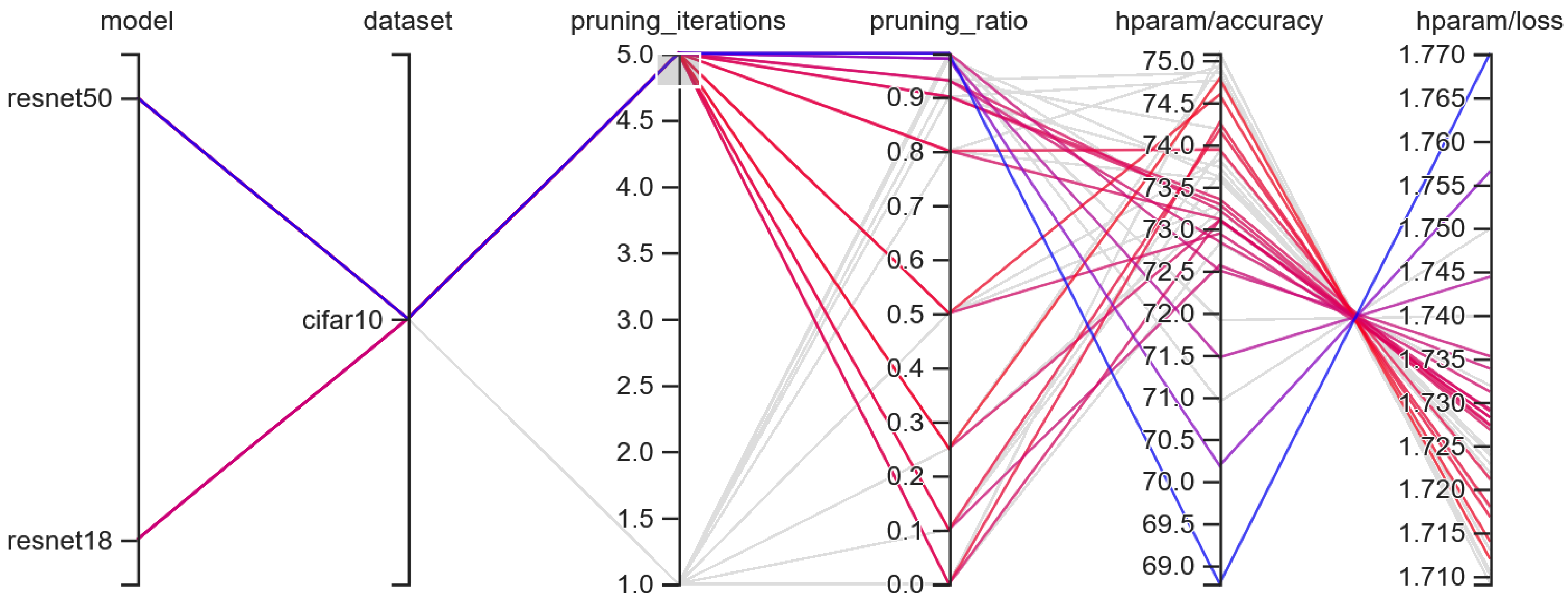
Experiments with single shot pruning



DNN Pruning

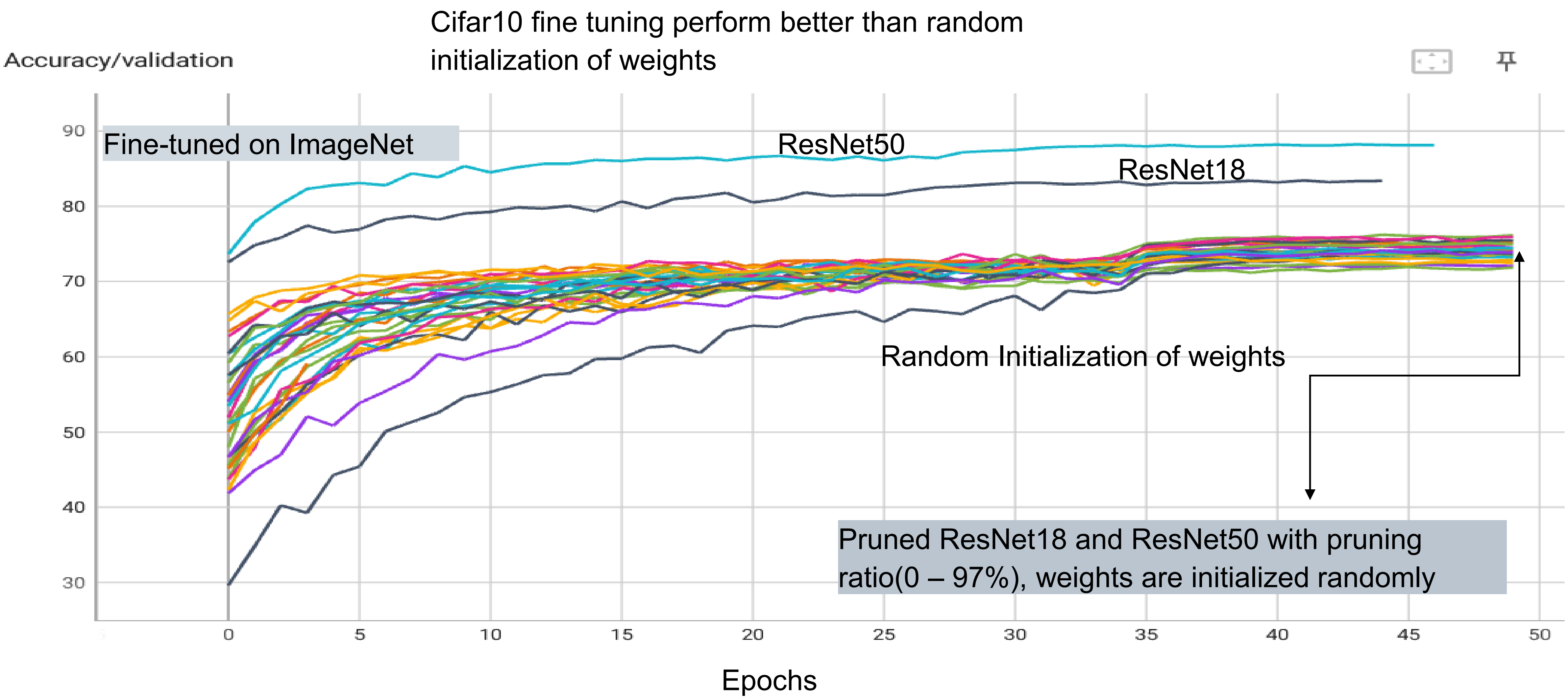
Results and Discussion

Experiments with iterative pruning



DNN Pruning

Results and Discussion



DNN Model Optimization

Conclusion

- Number of parameters in DNNs are exploding (175 billion in GPT3[3])
- As use cases of DNNs are increasing, deploying them on edge devices with real-time performance poses a great challenge.
- Model optimization strategies like Quantization, Pruning, Approximate Computing etc are used to make DNNs deployable on resource-constrained devices.
- There is no one-size-fits-all solution. You have to perform experiments on your Dataset, Network to find best fit.

Appendix: A Note on HPC Parallel Experiments

GNU *parallel* command[5]

```
$ parallel_exp.sh
#!/bin/bash
# Parallel Jobs
N_JOBS=10
ARGS="-P$N_JOBS --header :"

# Uncomment this line for dry run
#ARGS="--dry-run "$ARGS

# Experiment parameters
PROJECT='pruning_lottery_ticket_hypothesis'

MAX_EPOCHS=(50)
PRUNE_ITERS=(1 5)
PRUNE_METHODS=('l1')
PRUNE_RATIOS=(0 0.1 0.25 0.5 0.8 0.9 0.93 0.97 0.98)
REINITIALIZES=('false')
RANDOM_STATES=(1)
DATASETS=('cifar10')
MODELS=('resnet18' 'resnet50')

```

All parameters with different possible values are defined in a bash script first

```

parallel $ARGS \
    sbatch \
        --job-name=$PROJECT \
        $(echo --export=dataset={dataset},\
                    model={model},\
                    epochs={max_epochs},\
                    pruning_iterations={prune_iter},\
                    pruning_method={prune_method},\
                    pruning_ratio={prune_ratio},\
                    seed={random_state},\
                    weight_reinit={reinitialize} | tr -d '[:space:]')\
        run-job.sh \
            ::: max_epochs "${MAX_EPOCHS[@]}" \
            ::: prune_iter "${PRUNE_ITERS[@]}" \
            ::: prune_method "${PRUNE_METHODS[@]}" \
            ::: prune_ratio "${PRUNE_RATIOS[@]}" \
            ::: random_state "${RANDOM_STATES[@]}" \
            ::: reinitialize "${REINITIALIZES[@]}" \
            ::: dataset "${DATASETS[@]}" \
            ::: model "${MODELS[@]}" \
```

parallel command is used to submit jobs using SLURM[6] based ***sbatch*** command. ::: separator is used by parallel command to iterate over variables.

DNN Model Optimization

References

1. https://pytorch.org/tutorials/intermediate/pruning_tutorial.html
2. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks (https://arxiv.org/abs/1803.03635)
3. https://en.wikipedia.org/wiki/GPT-3
4. https://en.wikipedia.org/wiki/Parallel_coordinates
5. https://www.gnu.org/software/parallel/parallel_tutorial.html
6. https://slurm.schedmd.com/documentation.html

Notable Mentions for Programming help

– https://github.com/facebookresearch/open_lth

– https://github.com/jankrepl/mildlyoverfitted/tree/master/github_adventures/lottery

Thank You For Your Attention!