Model Pruning and Multi-Point Precision Optimization - Group 13

Jinal Vyas | Pratik Agarwal

Quantization and Pruning to Improve Inference Performance on ResNet-18

Model: ResNet-18 (PyTorch Pretrained)

Dataset: ImageNet

Techniques: Post-Training Quantization (PTQ) + Unstructured Pruning

Target Hardware: CPU, NVIDIA T4

Motivation

Deep neural networks like ResNet-18 are computationally expensive

Goal: Optimize inference performance and memory usage

Explore quantization + pruning techniques for efficiency

Focus on INT8, FP16, and mixed-precision models

Literature Review: Quantization in Deep Neural Networks

Key Concepts:

- **Quantization** reduces the precision of weights and activations (e.g., FP32 → INT8 or FP16).
- It helps **compress models** and **accelerate inference**, especially on hardware accelerators.

Key Papers:

- **Jacob et al., 2018** (Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference)
  - Proposed PyTorch's standard **static post-training quantization** method.
- **Micikevicius et al., 2018** (Mixed precision training)
  - Showed that **unsafe** operations (softmax, batchnorm, reductions) must remain in FP32.
  - Introduced the idea of **loss scaling** (for training) but also justified operator-wise precision choice for inference stability.

Takeaway:
Post-Training Quantization (PTQ) is a **lightweight, efficient method** but can cause small accuracy degradation without careful calibration.

Literature Review: Pruning in Deep Neural Networks

Key Concepts:

- Pruning removes less important weights/connections, making networks sparser.
- It reduces model size, memory footprint, and sometimes inference latency.

Key Papers:

- Han et al., 2015 (Deep Compression)
  - Proposed iterative magnitude pruning followed by retraining.
- Frankle & Carbin, 2019 (Lottery Ticket Hypothesis)
  - Showed existence of small subnetworks ("winning tickets") that train effectively.

Takeaway:
Pruning leverages the over-parameterization of deep networks to find compact subnetworks without major loss in accuracy.

Literature Review: Why Focus on ResNet-18 + ImageNet?

ResNet-18:

- Moderate size (~11M parameters).
- Strong baseline accuracy (~69.8% Top-1 on ImageNet).
- Well-understood bottleneck architecture (good for layerwise analysis).

ImageNet Dataset:

- Large-scale, high-diversity dataset (1.2M images, 1000 classes).
- Serves as **industry standard** benchmark for evaluating compression techniques.

Justification:
ResNet-18 + ImageNet is a **realistic, production-grade** setting to benchmark compression techniques like quantization and pruning.

Objectives

- Apply post-training quantization (PTQ) to ResNet-18
- Apply magnitude-based unstructured pruning to conv/fc layers
- Compare:
  - Model size
  - Inference latency
  - Top-5 accuracy
- Evaluate on both CPU and GPU

Architecture Overview

Flow: Pretrained → PTQ → Pruning → Evaluation

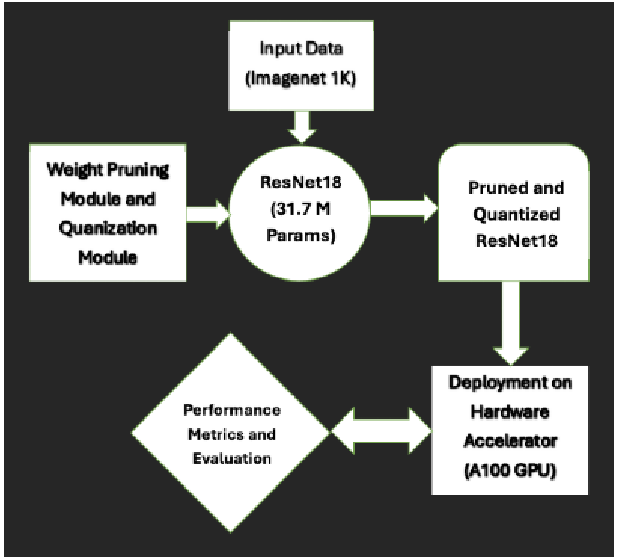


Figure 1: Pipeline for applying quantization and pruning

Post-Training Quantization

What is PTQ?

PTQ compresses a fully trained model by converting its weights and activations from FP32 to lower-precision formats (like INT8 or FP16) *after* training is completed.

No model re-training is required. Only a small calibration dataset is needed.

Steps Involved:

Train full model (ResNet-18 on ImageNet).

Prepare model for quantization (module fusion, quantization config setup).

Calibrate: Run a few batches through the model to collect activation ranges.

Convert model to a quantized version (weights/activations become INT8).

Methodology - PTQ Workflow

Load Pretrained Model
↓
Set Model to eval()
↓
Fuse Layers (Conv+BN+ReLU)
↓
Define Quantization Config
(get_default_qconfig('fbgemm'))
↓
Prepare Model (Insert Observers)
↓
Calibrate with Representative Data
(Forward Pass Only, No Gradients)
↓
Convert Model (Quantized Ops)
↓
Evaluate Accuracy and Latency

```
# Load Pretrained Model
model = load_pretrained_model()

# Set Model to Evaluation Mode
model.eval()

# Fuse Layers (e.g., Conv + BN + ReLU)
fused_model = fuse_layers(model)

# Define Quantization Configuration
fused_model.qconfig = get_default_qconfig('fbgemm')

# Prepare Model (Insert Observers)
prepared_model = prepare_model(fused_model)

# Calibrate Model with Representative Data
for batch in calibration_data:
    prepared_model(batch)

# Convert Model (Apply Quantization)
quantized_model = convert_model(prepared_model)

# Evaluate Quantized Model
evaluate_model(quantized_model)
```

Methodology - ResNet18: Before and After Quantization

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu1): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (add_relu_FF): FloatFunctional(
        (activation_post_process): Identity()
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu1): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (add_relu_FF): FloatFunctional(
        (activation_post_process): Identity()
      )
...
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
  (quant): QuantStub()
  (dequant): DeQuantStub()
)
```

```
ResNet(
  (conv1): QuantizedConvReLU2d(3, 64, kernel_size=(7, 7), stride=(2, 2), scale=0.0030553361866623163, zero_point=0, padding=(3, 3))
  (bn1): Identity()
  (relu): Identity()
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): QuantizedConvReLU2d(64, 64, kernel_size=(3, 3), stride=(1, 1), scale=0.002219037851318717, zero_point=0, padding=(1, 1))
      (bn1): Identity()
      (relu1): Identity()
      (conv2): QuantizedConv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), scale=0.013336232863366604, zero_point=127, padding=(1, 1))
      (bn2): Identity()
      (add_relu_FF): QFunctional(
        scale=0.007735365536063909 5, zero_point=0
        (activation_post_process): Identity()
      )
    )
    (1): BasicBlock(
      (conv1): QuantizedConvReLU2d(64, 64, kernel_size=(3, 3), stride=(1, 1), scale=0.004394975490868092, zero_point=0, padding=(1, 1))
      (bn1): Identity()
      (relu1): Identity()
      (conv2): QuantizedConv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), scale=0.01596945710480213, zero_point=144, padding=(1, 1))
      (bn2): Identity()
      (add_relu_FF): QFunctional(
        scale=0.011057315394282341, zero_point=0
...
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): QuantizedLinear(in_features=512, out_features=1000, scale=0.08900965005159378, zero_point=84, qscheme=torch.per_channel_affine)
  (quant): Quantize(scale=tensor([0.0038]), zero_point=tensor([0]), dtype=torch.quint8)
  (dequant): DeQuantize()
)
```

Pruning Strategy (Post-Quantization)

Approach: Apply global unstructured pruning *after* quantization

Why this order?

Targets final inference-ready weights

Aligns with deployment-stage model optimization

Prune-Then-Quantize: Pruning distorts the weight distribution, which can negatively affect quantization calibration.

Quantize-Then-Prune: Harder to implement but may yield better performance on hardware.

Framework and hardware support are major influencing factors

Challenges in Pruning Quantized Models

- Quantized layers store weights in compressed format (weight + scale + zero-point)
- PyTorch's standard pruning tools expect FP32 tensors
- Direct pruning of torch.nn.quantized.Conv2d not supported
- Deep integration of pruning masks with quantized parameters is non-trivial.

Custom Pruning for Quantized Models

- PyTorch pruning modules fail on quantized layers
- Developed custom pruning for:
  - INT8 weights (using scale + zero-point)
  - Mixed precision (Fully Connected Layer -> FP32 head, Rest All -> INT8)
  - Used model.half() for FP16
- Preserves weight format post quantization

```
for each layer in model:
    if layer is prunable (Conv or Linear):
        weights ← layer weights
        abs_weights ← absolute(weights)
        flatten_weights ← flatten(abs_weights)

        k ← pruning_fraction × number_of_weights
        threshold ← kth smallest value in flatten_weights

        mask ← abs_weights > threshold
        pruned_weights ← weights × mask

        layer.weights ← pruned_weights
```

Results

Table 1: Evaluation Metrics for FP32 Model

Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	14.36 ms	46.83 MB	89%
GPU	6.82 ms	46.83 MB	89.21%

Table 2: Evaluation Metrics for INT8 Model - Custom Quantization and Pruning

Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	782 ms	11.83 MB	76.45%
GPU	3.33 ms	11.81 MB	77.89%

Table 3: Evaluation Metrics for FP16 Model - Custom Quantization and Pruning

Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	16.55 ms	23.43 MB	81.98%
GPU	4.86 ms	23.22 MB	81.11%

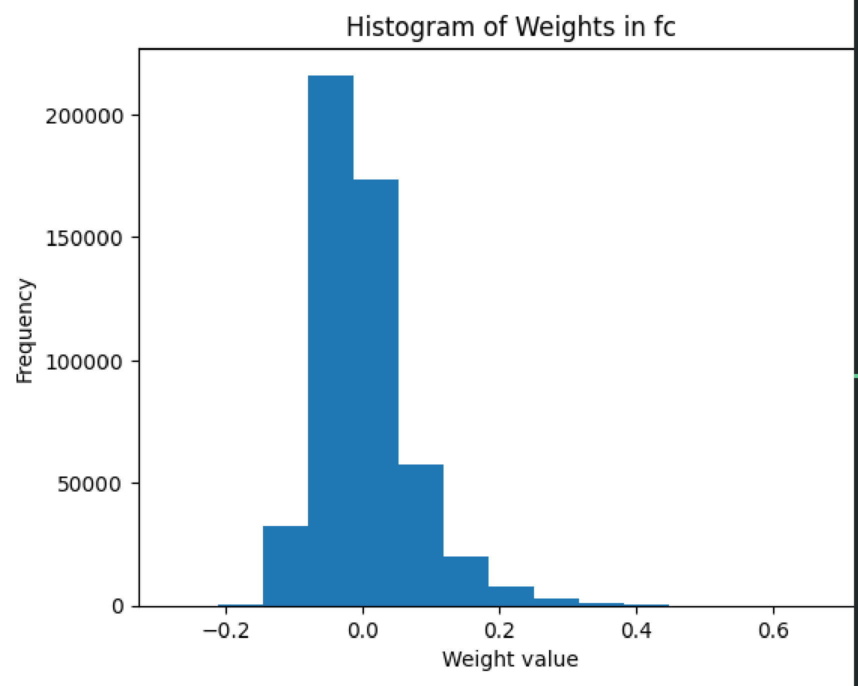
Table 4: Evaluation Metrics for Mixed Precision Model (INT8 + FP32 FC Layer) - Custom Quantization and Pruning

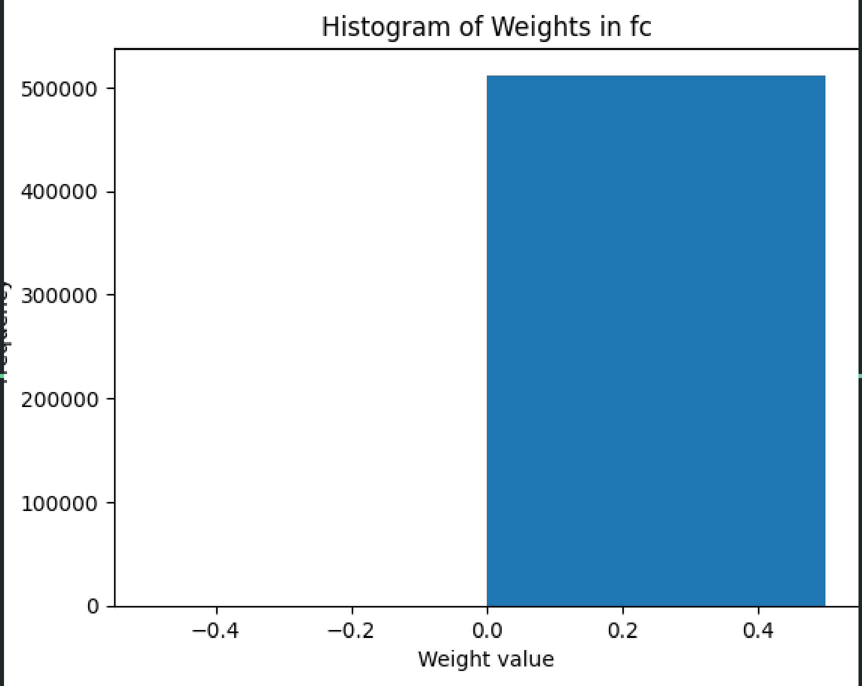
Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	80.82 ms	14.29 MB	79.234%
GPU	3.84	14.3	79.9%

Table 5: Evaluation Metrics for Just Pruned Model (Using PyTorch Function)

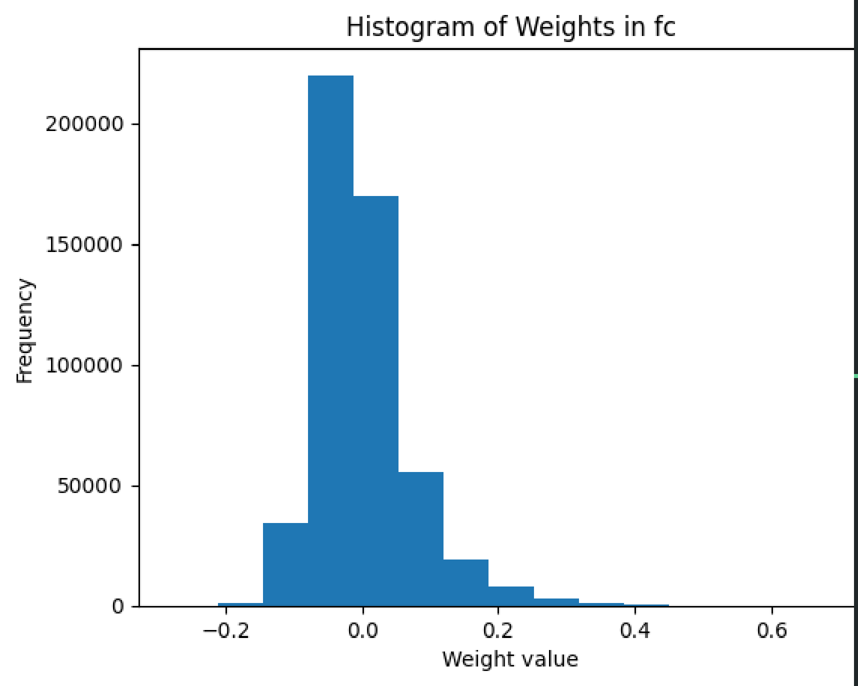
Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	12.11 ms	44.96 MB	89.54%
GPU	3.33 ms	45.1 MB	89.11%

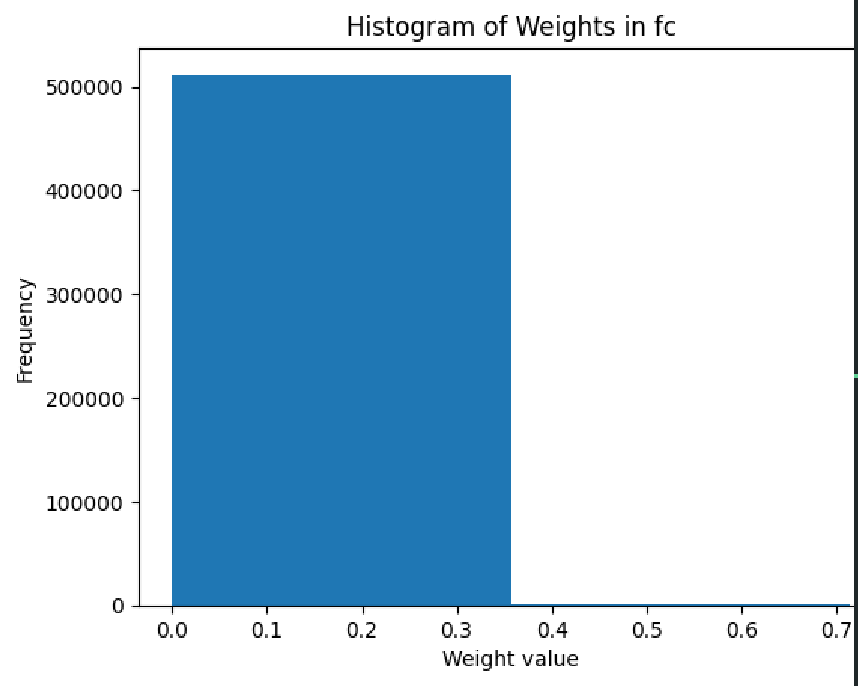
INT8 Fully Connected Layer Weight Distribution Before and After Pruning



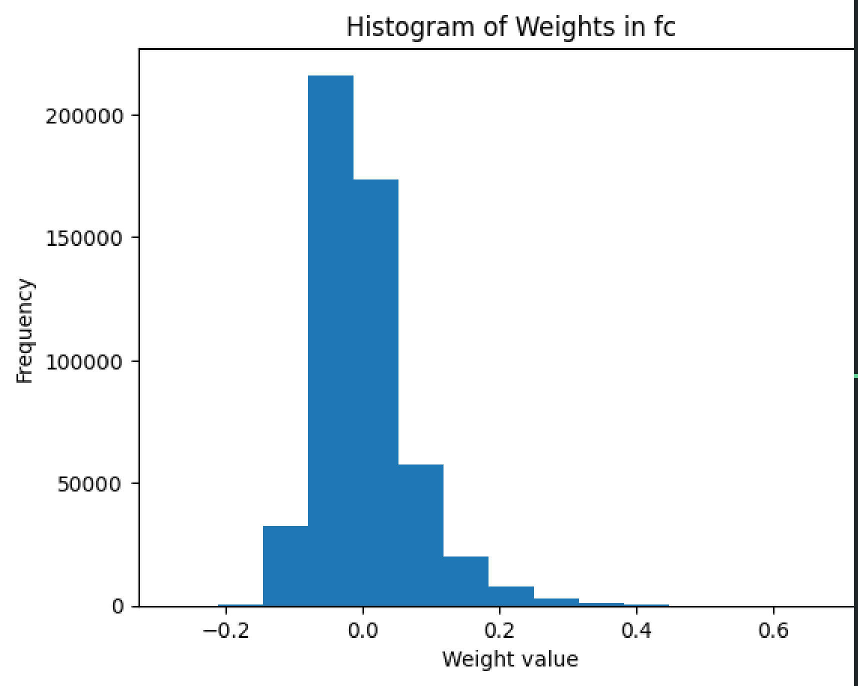


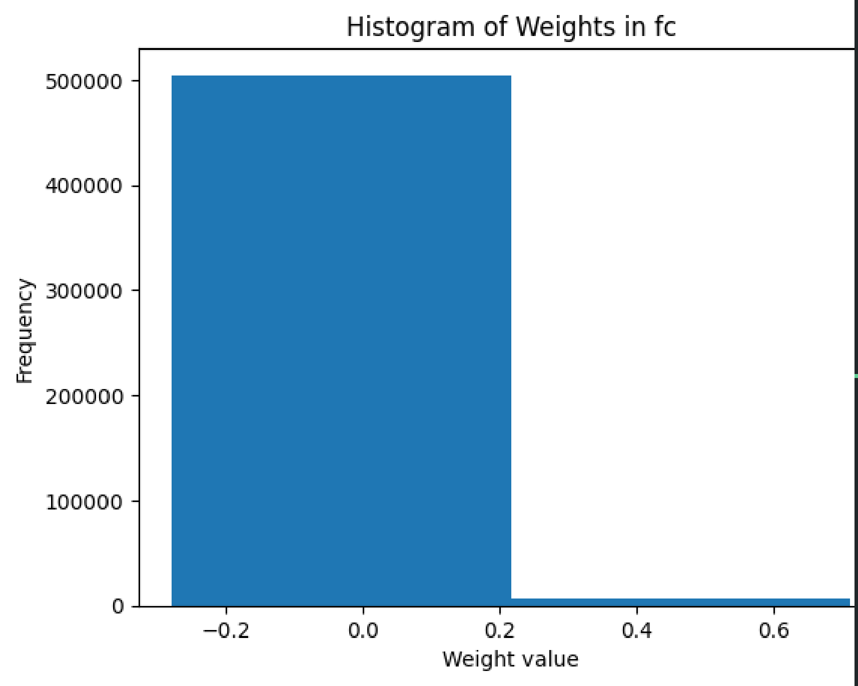
FC Layer Weight Distribution in FP16 Model: Pre- and Post-Pruning





Weight Distribution in Mixed Precision Model (FP32 FC + INT8 Rest)





Key Insights

- Quantization drastically reduces model size
- Pruning induces sparsity, slightly lowers accuracy
- Mixed precision balances accuracy and efficiency
- INT8 + Pruning: smallest model, best latency

Conclusion

- Quantization + pruning > pruning alone
- Slight accuracy trade-off worth the gains
- PTQ + unstructured pruning effective
- Accuracy drop can be recovered with fine-tuning

Future Work

EXPLORE QAT + PRUNING

INTEGRATE WITH SPARSE INFERENCE ENGINES

APPLY TO LARGER MODELS (E.G., RESNET-50)

FINETUNE QUANTIZED + PRUNED MODEL TO IMPROVE ACCURACY

References

Han et al., Deep CompressionJacob et al., Quantization and Training of Neural Networks for Efficient Inference

Jacob et al., Quantization and Training of Neural Networks for Efficient

Krishnamoorthi, Quantization in PyTorch

Gale et al., The State of Sparsity in Deep Neural Networks

Migacz, 8-bit Inference with TensorRTKrishnamoorthi, Quantization in PyTorch

Thank you! Any Questions?