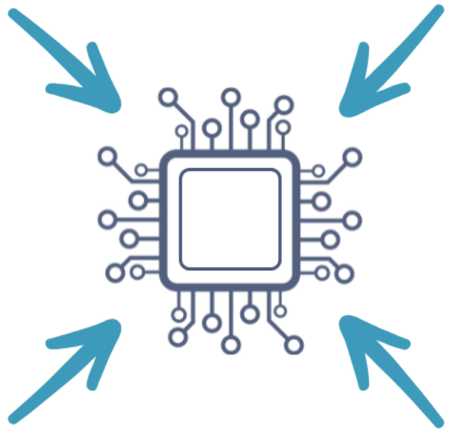
Applying dynamic quantization

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Sergiy Tkachuk
Director, GenAI Productivity

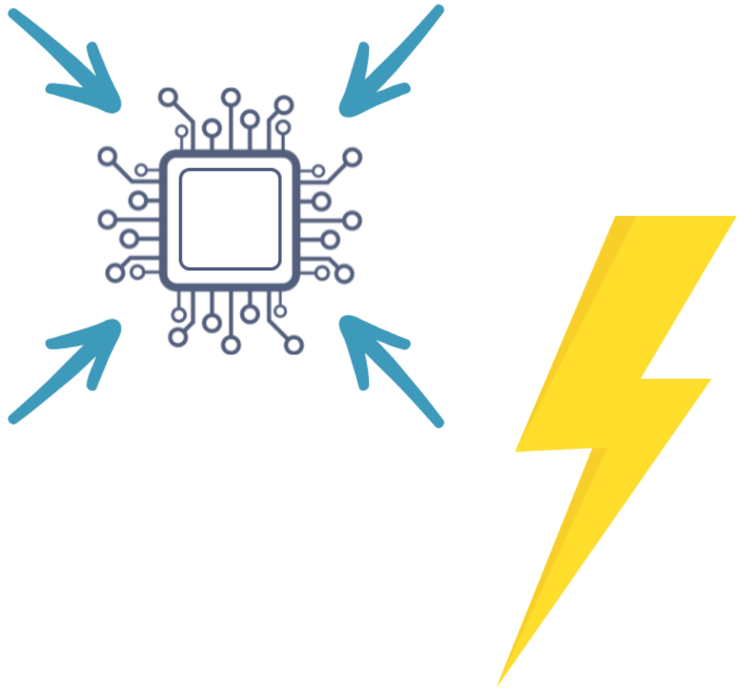
Why use quantization?

- Memory reduction



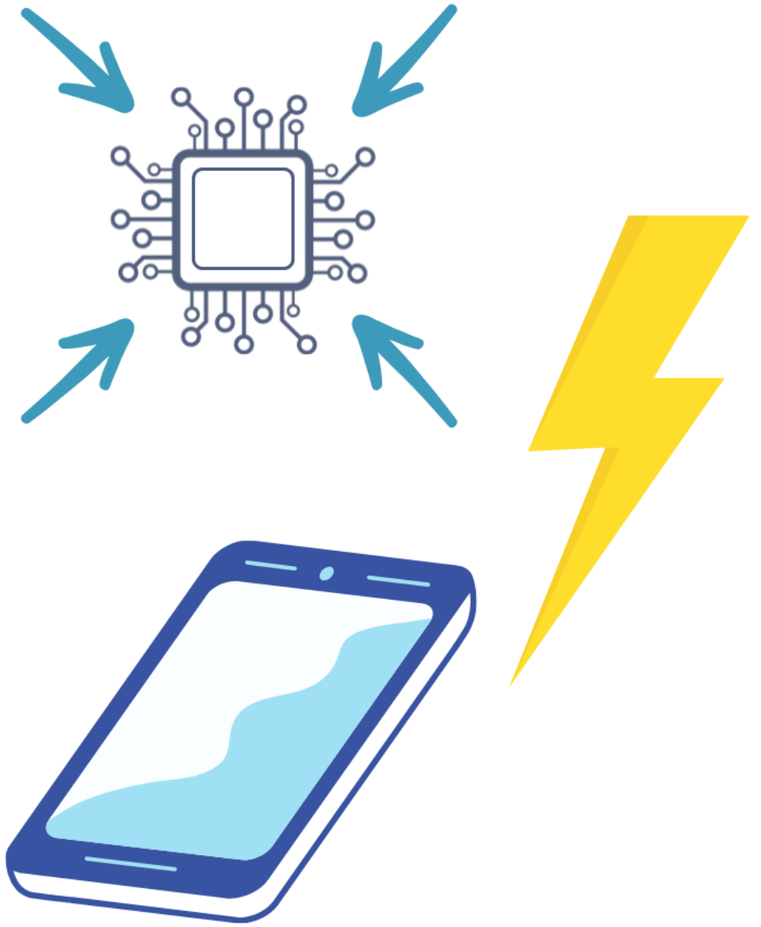
Why use quantization?

- Memory reduction
- CPU acceleration



Why use quantization?

- Memory reduction
- CPU acceleration
- Mobile inference

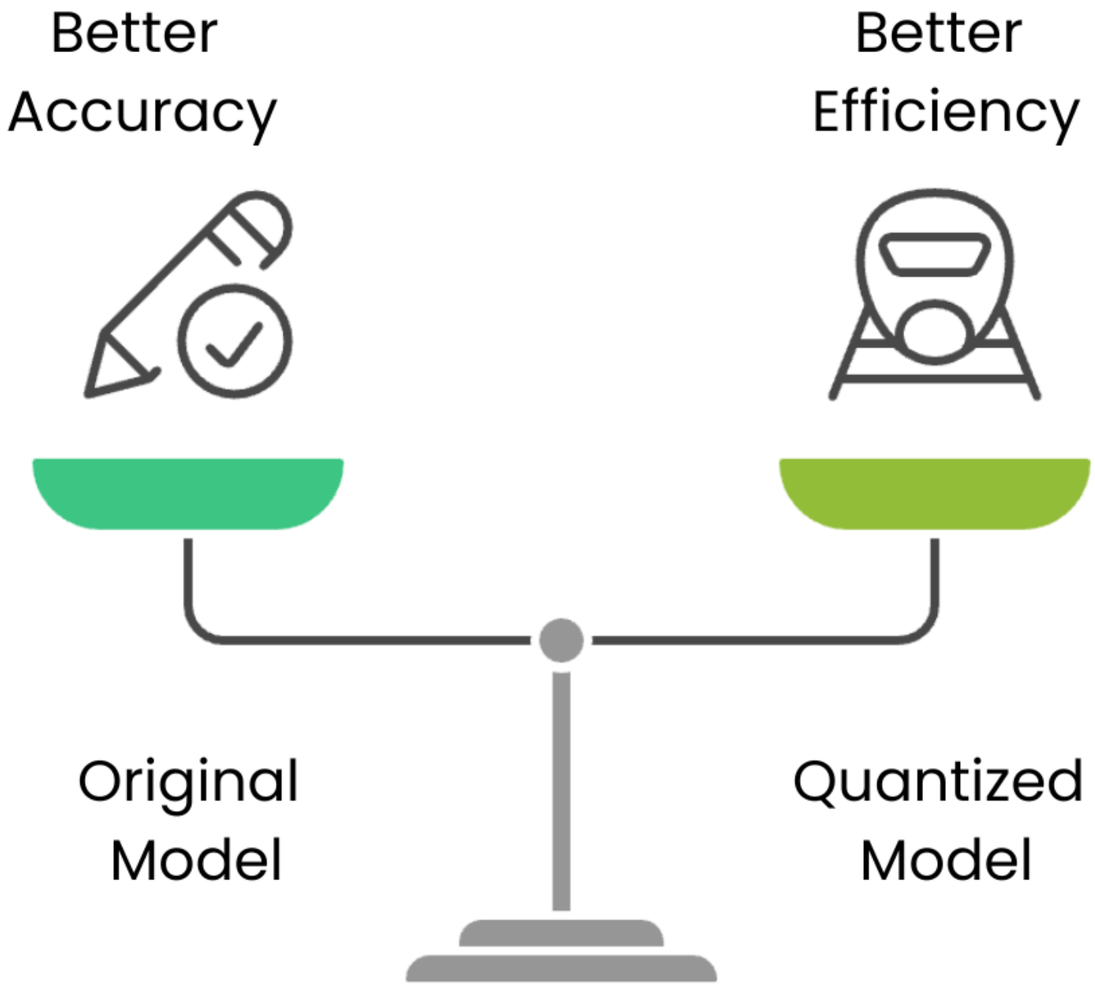


What is dynamic quantization?

- Reduce precision of weights and operations
- Improves inference speed
- Ideal for deployment on resource-constrained devices

```
import torch
from torch.quantization
import quantize_dynamic

model_quantized = quantize_dynamic(
    model,
    {torch.nn.Linear},
    dtype=torch.qint8
)
```

Evaluating quantization impact



Performance comparison

- ⚡ Compare inference speed and memory footprint
- □ Determine acceptable accuracy trade-offs
- Decide on quantization suitability based on deployment needs

Comparing performance

```
import time

def measure_time(model, data_loader):
      model.eval()  # Set model to evaluation mode
    start_time = time.time()
    for inputs in data_loader:
        _ = model(inputs)
    end_time = time.time()
    return end_time - start_time

original_time = measure_time(model, test_loader)
quant_time = measure_time(model_quant, test_loader)

print(f"Original Model Time: {original_time:.2f}s")
print(f"Quantized Model Time: {quant_time:.2f}s")
```

Let's practice!

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Implementing model pruning techniques

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

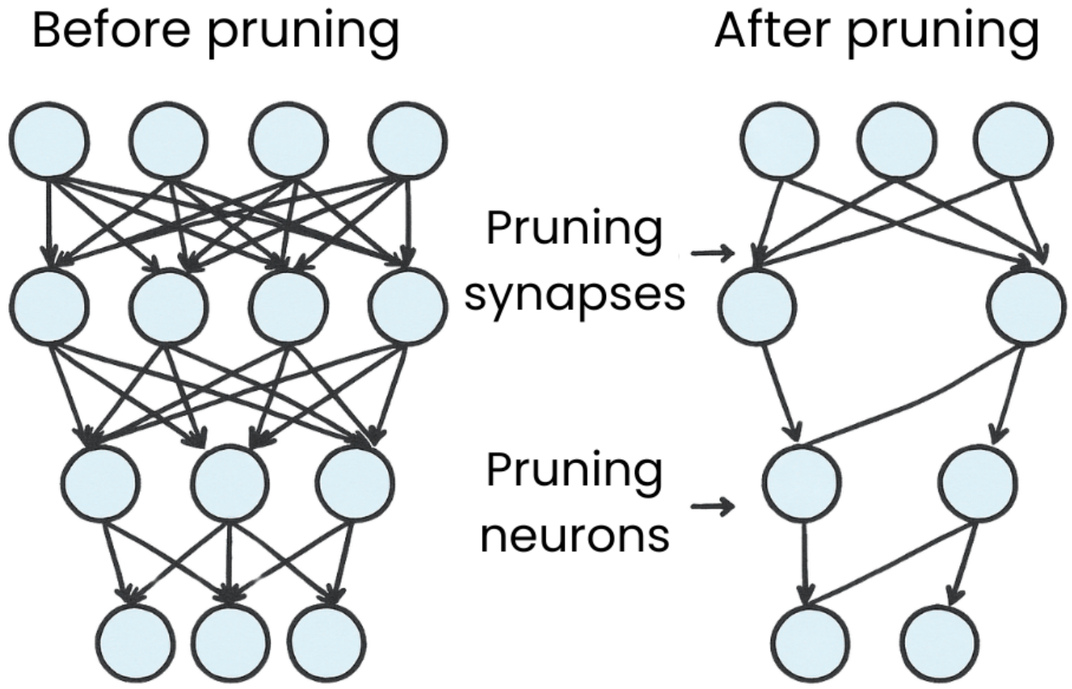
Sergiy Tkachuk
Director, GenAI Productivity

When to use pruning?

- 📱 Helpful when deploying models to edge or embedded systems
- 🔗 Can be combined with quantization for compound efficiency gains
- ⚡ Use when reducing latency or model size is a priority

What is model pruning?

- Removes less important connections in neural networks
- Leads to sparse models that are efficient to store and compute
- A common approach is L1 unstructured pruning



What is model pruning?

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc, name="weight",
                      amount=0.4)

print(model.fc.weight.data)
```

```
tensor([[ 0.25, -0.13,  0.05,  0.70],
        [-0.88,  0.31, -0.02,  0.44]]) # Before pruning


tensor([[ 0.25, -0.13,  0.00,  0.70],
        [ 0.00,  0.31,  0.00,  0.44]]) # After pruning (40% weights set to 0)
```

Understanding pruning masks

- Pruning adds a binary mask to each targeted weight tensor.

- Mask = 1 --> weight kept.
- Mask = 0 --> weight set to 0 at forward time.

- Weights are still stored in memory until the mask is removed.

Making pruning permanent

- By default, pruned weights remain part of the original tensor
- To finalize pruning, remove reparametrizations
- Converts sparse layer into standard layer with zeroed weights

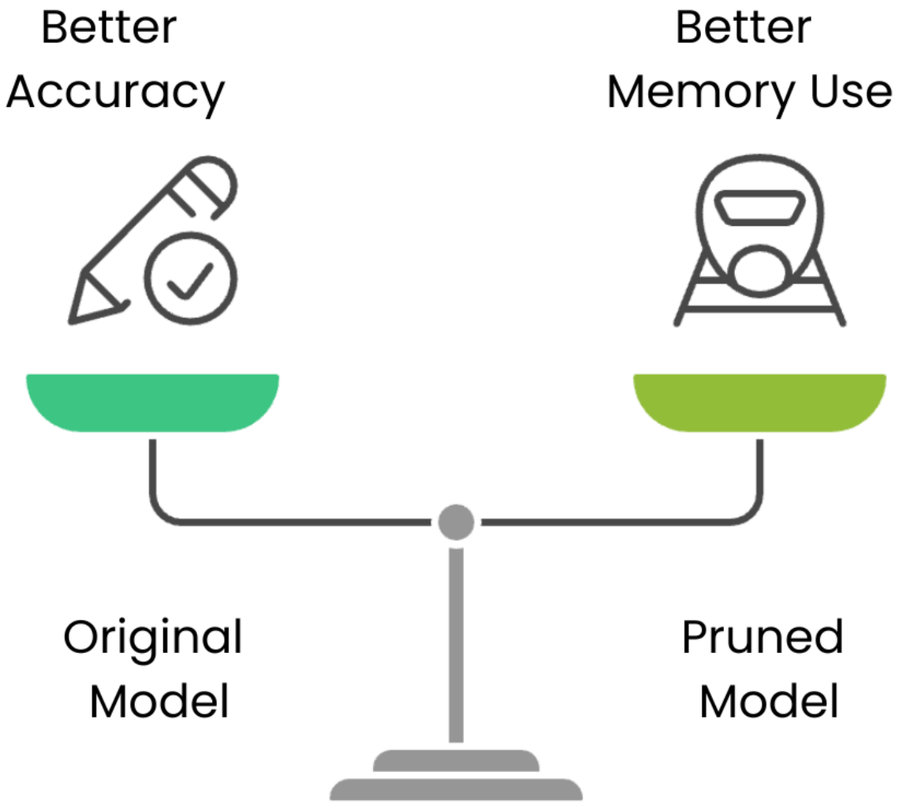
```
Sequential(
  (fc): Linear(
    in_features=128, out_features=64,
    bias=True
    (weight): PrunedParam()
  )
) # Before prune.remove
```

```
import torch.nn.utils.prune as prune

prune.remove(model.fc, 'weight')

# Print model structure
print(model)
```

```
Sequential(
  (fc): Linear(in_features=128,
               out_features=64,
               bias=True)
) # After prune.remove
```

Evaluating pruning impact

- Compare original and pruned model performance
- Expect slight drops in accuracy but major size/memory savings
- Helps assess if the trade-off is acceptable for deployment



Let's practice!

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Exporting models with TorchScript

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Sergiy Tkachuk
Director, GenAI Productivity

What is TorchScript?

- Independent of Python
- Efficient in production
- Examples:
  - Deployment on mobile devices

What is TorchScript?

- Independent of Python
- Efficient in production
- Examples:
  - Deployment on mobile devices
  - High-performance inference in production systems

Converting models to TorchScript

Two methods for conversion:

- `torch.jit.trace` : Uses example inputs to trace execution
- `torch.jit.script` : Compiles the model by analyzing Python code

When to use:

1. Use `trace` for simpler models
2. Use `script` for models with control flow (e.g., loops)

```
import torch
import torch.nn as nn
class SimpleModel(nn.Module):
    def forward(self, x):
        return x * 2
model = SimpleModel()
scripted_model = torch.jit.script(model)
```

Saving and loading TorchScript models

- Saving the model:
  - `torch.jit.save` : Save the scripted model to a file
- Loading the model:
  - `torch.jit.load` : Load the model back for inference

```
# Save the model
torch.jit.save(scripted_mod,"model.pt")

# Load the model
loaded_model=torch.jit.load("model.pt")
```

Performing inference with TorchScript

- Steps:
  - Load the TorchScript model
  - Pass inputs to the model for predictions
  - Outputs are identical to PyTorch predictions

```
# Perform inference
input_arr = [1.0, 2.0, 3.0]
input_tensor = torch.tensor(input_arr)
output = loaded_model(input_tensor)
print(output)
```

Example Input:

- Input Tensor: `[1.0, 2.0, 3.0]`

Example Output:

- Output Tensor: `[2.0, 4.0, 6.0]`

TorchScript in a nutshell

- `torch.jit.trace` : Works for static models
- `torch.jit.script` : Handles dynamic control flow
- `torch.jit.save` : Saves the scripted model
- `torch.jit.load` : Reloads for inference

Let's practice!

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Recap: Scalable AI Models with PyTorch Lightning

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Sergiy Tkachuk
Director, GenAI Productivity

Building scalable models with PyTorch Lightning

- Neural network structuring
- Effective training logic
- LightningModule implementation

```
import lightning.pytorch as pl
import torch.nn as nn

class ClassificationModel(pl.LightningModule):
    def __init__(self, input_dim,
                 hidden_dim, num_class):
          # Initialize parent class
        super().__init__()
        # First layer
        self.layer1 = nn.Linear(input_dim,
                                hidden_dim)
        # Activation function
        self.relu = nn.ReLU()
        # Output layer
        self.layer2 = nn.Linear(hidden_dim,
                                num_class)
```

Advanced techniques in PyTorch Lightning

- Data management and preprocessing
- Validation and testing methods
- Lightning callbacks optimization

```
from lightning.pytorch import Trainer
from lightning.pytorch.callbacks import EarlyStopping, ModelCheckpoint

checkpoint = ModelCheckpoint(
    monitor='val_accuracy',
    save_top_k=2,
    mode='max')
early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=5,
    mode='max')
```

Optimizing models for scalability

- Model quantization strategies
- Model pruning for efficiency
- TorchScript for deployment

```
import torch
import torch.nn as nn

class SimpleModel(nn.Module):
    def forward(self, x):
        return x * 2

scripted_model = torch.jit.script(SimpleModel())

torch.jit.save(scripted_mod,"model.pt") # Save the model
loaded_model=torch.jit.load("model.pt") # Load the model
```

Congratulations!

SCALABLE AI MODELS WITH PYTORCH LIGHTNING