EE 046211 - Technion - Deep Learning

[Tal Daniel (https://taldatech.github.io)](https://taldatech.github.io)

Tutorial 10 - Resource Efficiency - AMP, Quantization and Pruning

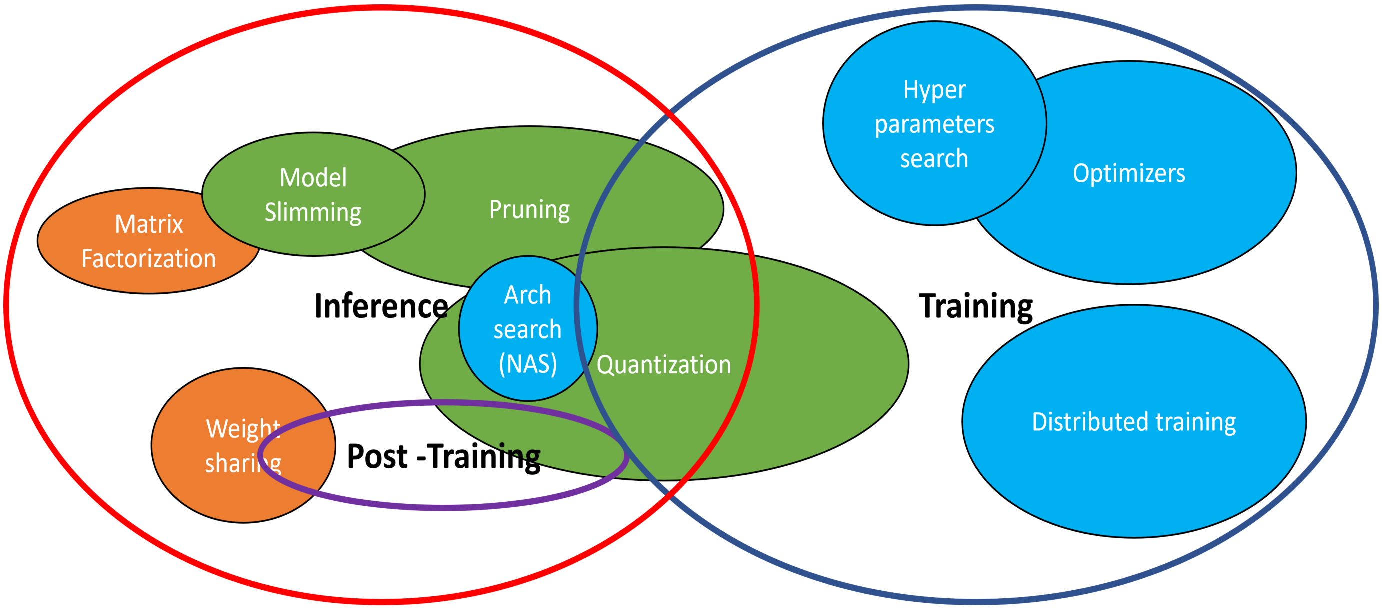
Agenda

In [1]:

```
# imports for the tutorial
import numpy as np
import matplotlib.pyplot as plt
import time
import os
import gc

# pytorch imports
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.quantization
import torch.nn.utils.prune as prune
```

Resource Efficiency Motivation

* Training (large) neural networks is usually a long process that requires sufficient hardware.
* Hardware, such as GPUs, consumes a lot of energy which, for environmental reasons, can be spared.
* For real life applications, we ideally want fast inference times and we also want to deploy our models to, possibly, various devices such as mobile phones that don't have a matching hardware to the machine our models were trained on.
* And of course, these things COST MONEY, and usually lots of it.
* In this tutorial, we will discuss several approaches to incorporate compression during training and during inference.



Energy Required for Mathematical Operations

Operation	MUL	ADD
8-bit Integer	0.2 pJ	0.03 pJ
32-bit Integer	3.1 pJ	0.1 pJ
16-bit Floating Point	1.1 pJ	0.4 pJ
32-bit Floating Point	3.7 pJ	0.9 pJ

How can we utilize this information to make training and inference more efficient?

- pJ - pico-Joules $= 10^{-12}$ Joules, 1 Watt = 1 Joule per second.
- Source (https://ieeexplore.ieee.org/document/6757323)

Train-time Efficiency - Automatic Mixed Precision (AMP)

- Deep Neural Network training has traditionally relied on FP32 (32-bit Floating Point, IEEE single-precision format).
- The (automatic) mixed precision technique - training with FP16 (16-bit Floating Point, half-precision) while maintaining the network accuracy achieved with FP32.
- Enabling mixed precision involves two steps:
  - Porting the model to use the half-precision data type **where appropriate**.
  - Using loss scaling to preserve small gradient values.

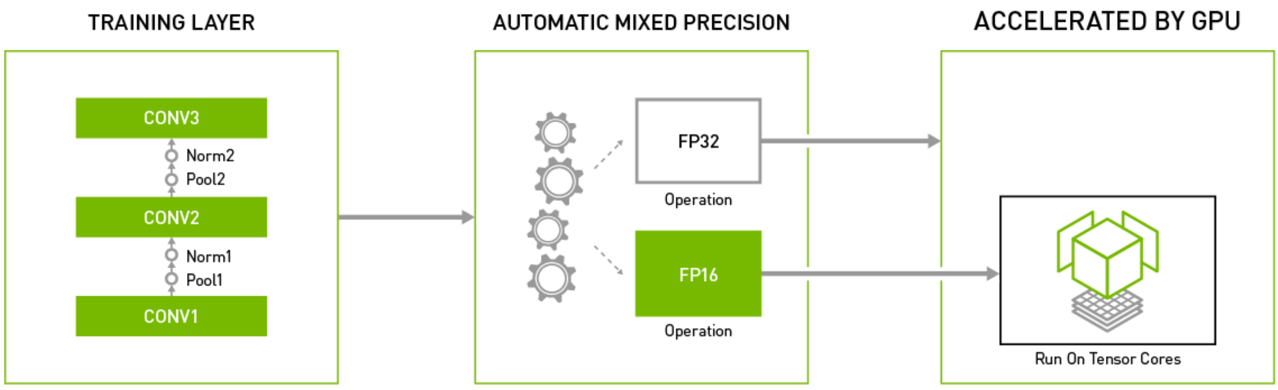


Image Source (https://developer.nvidia.com/automatic-mixed-precision)

The Benefits of AMP

- Speeds up math-intensive operations, such as linear and convolution layers.
- Speeds up memory-limited operations by accessing half the bytes compared to single-precision.
- Reduces memory requirements for training models, enabling larger models or larger minibatches.
- This feature enables automatic conversion of certain GPU operations from FP32 precision to mixed precision, thus improving performance while maintaining accuracy.

Performance of mixed precision training on NVIDIA 8xV100 vs. FP32 training on 8xV100 GPU

- Bars represent the speedup factor of V100 AMP over V100 FP32. The higher the better.

Image Source (https://pytorch.org/blog/accelerating-training-on-nvidia-gpus-with-pytorch-automatic-mixed-precision/)

AMP in PyTorch

- We will follow an [example (https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html)](https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html) by [Michael Carilli (https://github.com/mcarilli)](https://github.com/mcarilli).
- Note: this is only relevant for **GPU-powered machines**.
    - Mixed precision primarily benefits Tensor Core-enabled architectures (Volta, Turing, Ampere, Hopper), where one can obsereve significant (2-3X) speedup on those architectures. On earlier architectures (Kepler, Maxwell, Pascal), you may observe a modest speedup.
- `torch.cuda.amp` provides convenience methods for mixed precision, where some operations use the `torch.float32` (float) datatype and other operations use `torch.float16` (half).
- Some ops, like linear layers and convolutions, are much faster in `float16`, where other ops, like reductions, often require the dynamic range of `float32`.
- Mixed precision tries to match each op to its appropriate datatype, which can reduce your network's runtime and memory footprint.
- AMP - "automatic mixed precision training" uses `torch.cuda.amp.autocast` and `torch.cuda.amp.GradScaler` together, as we will soon explain.

In [2]:
```python
# Timing utilities
start_time = None

def start_timer():
    global start_time
    gc.collect()
    torch.cuda.empty_cache()
    torch.cuda.reset_max_memory_allocated()
    torch.cuda.synchronize()
    start_time = time.time()

def end_timer_and_print(local_msg):
    torch.cuda.synchronize()
    end_time = time.time()
    print("\n" + local_msg)
    print("Total execution time = {:.3f} sec".format(end_time - start_time))
    print("Max memory used by tensors = {} bytes".format(torch.cuda.max_memory_allocated()))
```

In [3]:
```python
# for our demonstrations, we will use a simple MLP network
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
def make_model(in_size, out_size, num_layers):
    layers = []
    for _ in range(num_layers - 1):
        layers.append(torch.nn.Linear(in_size, in_size))
        layers.append(torch.nn.ReLU())
    layers.append(torch.nn.Linear(in_size, out_size))
    return torch.nn.Sequential(*tuple(layers)).to(device)
```

- `batch_size`, `in_size`, `out_size`, and `num_layers` are chosen to be large enough to saturate the GPU with work.
- Typically, mixed precision provides the greatest speedup when the **GPU is saturated**.
- Small networks may be CPU bound, in which case mixed precision won't improve performance.
- Sizes are also chosen such that the participating dimensions of the linear layers are multiples of 8, to permit Tensor Core usage on Tensor Core-capable GPUs.

In [4]:
```python
batch_size = 512 # try, for example, 128, 256, 512.
in_size = 4096
out_size = 4096
num_layers = 3
num_batches = 50
epochs = 3

# creates data in default precision (fp32).
# the same data is used for both default and mixed precision trials below.
# you don't need to manually change inputs' dtype when enabling mixed precision.
data = [torch.randn(batch_size, in_size, device=device) for _ in range(num_batches)]
targets = [torch.randn(batch_size, out_size, device=device) for _ in range(num_batches)]

loss_fn = torch.nn.MSELoss().to(device)
```

Default Precision (FP32)

Without `torch.cuda.amp` , the following simple network executes all ops in default precision (`torch.float32`)

In [11]:
```
net = make_model(in_size, out_size, num_layers)
opt = torch.optim.SGD(net.parameters(), lr=0.001)

start_timer()
for epoch in range(epochs):
    for input, target in zip(data, targets):
        output = net(input)
        loss = loss_fn(output, target)
        opt.zero_grad() # set_to_none=True here can modestly improve performance
        loss.backward()
        opt.step()

end_timer_and_print("Default precision:")
```

```
C:\ProgramData\Anaconda3\envs\deep_learn\lib\site-packages\torch\cuda\memory.py:263: FutureWarning: torc
h.cuda.reset_max_memory_allocated now calls torch.cuda.reset_peak_memory_stats, which resets /all/ peak m
emory stats.
  FutureWarning)

Default precision:
Total execution time = 13.958 sec
Max memory used by tensors = 1493322752 bytes
```

Adding Autocast

- Instances of `torch.cuda.amp.autocast` serve as context managers that allow regions of your script to run in mixed precision.
- In these regions, CUDA ops run in a `dtype` **chosen by** **`autocast`** to improve performance while maintaining accuracy.
- See the [Autocast Op Reference (https://pytorch.org/docs/stable/amp.html#autocast-op-reference)](https://pytorch.org/docs/stable/amp.html#autocast-op-reference) for details on what precision autocast chooses for each op, and under what circumstances.

In [7]:
```
for epoch in range(0): # 0 epochs, this section is for illustration only
    for input, target in zip(data, targets):
        # Runs the forward pass under autocast.
        with torch.cuda.amp.autocast():
            output = net(input)
            # output is float16 because linear layers autocast to float16.
            assert output.dtype is torch.float16

            loss = loss_fn(output, target)
            # loss is float32 because mse_loss layers autocast to float32.
            assert loss.dtype is torch.float32

        # exits autocast before backward().
        # backward passes under autocast are not recommended.
        # backward ops run in the same dtype autocast chose for corresponding forward ops.
        opt.zero_grad() # set_to_none=True here can modestly improve performance
        loss.backward()
        opt.step()
```

Adding GradScaler

- Gradient scaling helps **prevent gradients with small magnitudes from flushing to zero** (“underflowing”) when training with mixed precision.
- `torch.cuda.amp.GradScaler` performs the steps of gradient scaling conveniently.

In [8]:
```
# constructs scaler once, at the beginning of the convergence run, using default args.
# the same GradScaler instance should be used for the entire convergence run.
# if you perform multiple convergence runs in the same script, each run should use
# a dedicated fresh GradScaler instance. GradScaler instances are lightweight.

scaler = torch.cuda.amp.GradScaler()

for epoch in range(0): # 0 epochs, this section is for illustration only
    for input, target in zip(data, targets):
        # use autocast as before
        with torch.cuda.amp.autocast():
            output = net(input)
            loss = loss_fn(output, target)

        opt.zero_grad() # set_to_none=True here can modestly improve performance

        # scales loss. calls backward() on scaled loss to create scaled gradients.
        scaler.scale(loss).backward()

        # scaler.step() first unscales the gradients of the optimizer's assigned params.
        # if these gradients do not contain infs or NaNs, optimizer.step() is then called,
        # otherwise, optimizer.step() is skipped (!).
        scaler.step(opt)

        # Updates the scale for next iteration.
        scaler.update()
```

Putting it All Together: “Automatic Mixed Precision”

- The following also demonstrates `enabled` , an optional convenience argument to `autocast` and `GradScaler` .
- If `False` , `autocast` and `GradScaler` ’s calls become no-ops, which allows switching between default precision and mixed precision without if/else statements.

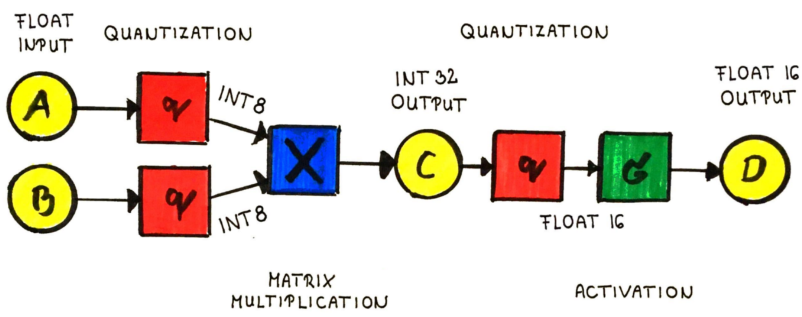
In [13]:
```
use_amp = True

net = make_model(in_size, out_size, num_layers)
opt = torch.optim.SGD(net.parameters(), lr=0.001)
scaler = torch.cuda.amp.GradScaler(enabled=use_amp) # notice the `enabled` parameter

start_timer()
for epoch in range(epochs):
    for input, target in zip(data, targets):
        # notice the `enabled` parameter
        with torch.cuda.amp.autocast(enabled=use_amp):
            output = net(input)
            loss = loss_fn(output, target)

        # set_to_none=True here can modestly improve performance, replace 0 with None (save mem)
        opt.zero_grad(set_to_none=True)
        scaler.scale(loss).backward()
        scaler.step(opt)
        scaler.update()

end_timer_and_print("Mixed precision:")
```

```
C:\ProgramData\Anaconda3\envs\deep_learn\lib\site-packages\torch\cuda\memory.py:263: FutureWarning: torc
h.cuda.reset_max_memory_allocated now calls torch.cuda.reset_peak_memory_stats, which resets /all/ peak m
emory stats.
  FutureWarning)

Mixed precision:
Total execution time = 14.081 sec
Max memory used by tensors = 1585620992 bytes
```

- Here we got similar performance due to the GPU architecture (which is older).
- For more advanced topics on AMP, [see here (https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html)](https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html).

Inference/Train-time Efficiency - Quantization

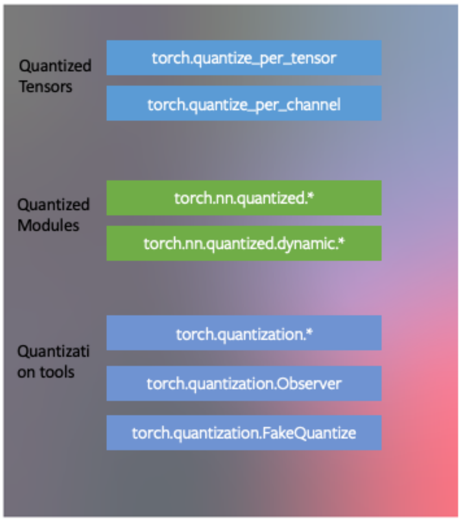
- Quantization refers to techniques for doing both computations and memory accesses with lower precision data, usually `int8` compared to floating point implementations.
- Quantization leverages 8-bit integer (`int8`) instructions to reduce the model size and run the inference faster (reduced latency).
- This enables providing quick inference from a trained model and even fitting it into the resources available on a mobile device.
- Quantization allows for siginificant performance gains!
  - Up to 4x reduction in model size.
  - Up to 2-4x reduction in memory bandwidth.
  - Up to 2-4x faster inference due to savings in memory bandwidth and faster compute with `int8` arithmetic (the exact speed up varies depending on the hardware, the runtime, and the model).
- Quantization doesn't come without additional cost, as it means introducing approximations and the resulting networks have slightly less accuracy.
- These techniques attempt to **minimize the gap between the full floating point accuracy and the quantized accuracy**.



[Image Source (https://towardsdatascience.com/how-to-accelerate-and-compress-neural-networks-with-quantization-edfbbabb6af7)](https://towardsdatascience.com/how-to-accelerate-and-compress-neural-networks-with-quantization-edfbbabb6af7)

Quantization in PyTorch

- Quantization is available in PyTorch in various flavors starting in version 1.3 and there are published quantized models for ResNet, ResNext, MobileNetV2, GoogleNet, InceptionV3 and ShuffleNetV2 in the PyTorch `torchvision >= 0.5` library.
- PyTorch has data types corresponding to quantized tensors, which share many of the features of tensors.
- PyTorch supports quantized modules for common operations as part of the `torch.nn.quantized` and `torch.nn.quantized.dynamic` name-space.
- Quantization is compatible with the rest of PyTorch: quantized models are traceable and scriptable. Quantized and floating point operations can be mixed in a model.
- Mapping of floating point tensors to quantized tensors is customizable with user defined observer/fake-quantization blocks. PyTorch provides default implementations that should work for most use cases.
- Currently the quantized models **can only be run on CPU**. However, it is possible to send the non-quantized parts of the model to a GPU.
  - GPU quantization is a work-in-progress, see [PTQ (Post Training Quantization) (https://pytorch.org/TensorRT/tutorials/ptq.html)](https://pytorch.org/TensorRT/tutorials/ptq.html).



[Image Source (https://pytorch.org/blog/introduction-to-quantization-on-pytorch/)](https://pytorch.org/blog/introduction-to-quantization-on-pytorch/)

The Three Types of Quantization

Dynamic Quantization

- Involves not just converting the weights to `int8` (as in all quantization variants), but also converting the **activations** to `int8` *on the fly*, just before doing the computation (hence "dynamic").
- The computations will be performed using efficient `int8` matrix multiplication and convolution implementations, resulting in faster compute.
  - However, the activations are read and written to memory in floating point format.
- In PyTorch: `torch.quantization.quantize_dynamic` - takes in a **model**, as well as a couple other arguments, and produces a quantized model.

In [ ]:
```
# dynamic quantization usage example
quantized_model = torch.quantization.quantize_dynamic(model, {torch.nn.Linear}, dtype=torch.qint8)
```

- Dynamic Quantization [documentation (https://pytorch.org/docs/stable/quantization.html#torch.quantization.quantize_dynamic)](https://pytorch.org/docs/stable/quantization.html#torch.quantization.quantize_dynamic).
- Examples (project ideas!): [LSTM word model quantization (https://pytorch.org/tutorials/advanced/dynamic_quantization_tutorial.html)](https://pytorch.org/tutorials/advanced/dynamic_quantization_tutorial.html), [pre-trained BERT quantization (https://pytorch.org/tutorials/intermediate/dynamic_quantization_bert_tutorial.html)](https://pytorch.org/tutorials/intermediate/dynamic_quantization_bert_tutorial.html).

Post-Training Static Quantization

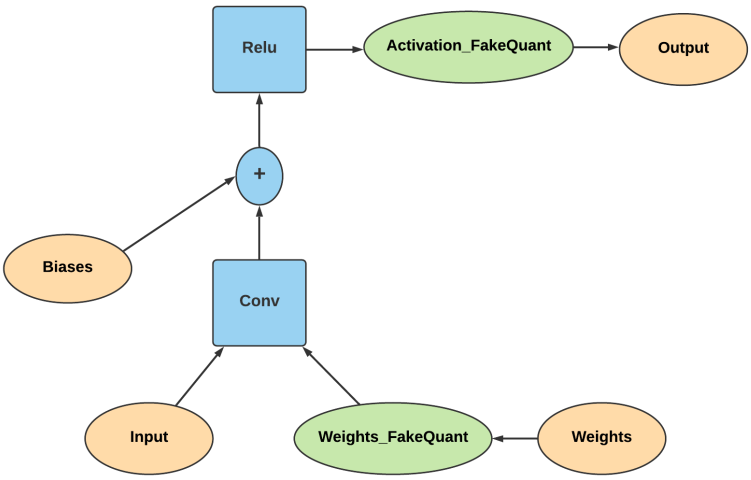
- Converting networks to use both integer arithmetic and `int8` memory accesses can improve the latency performance.
- Static quantization first feeds batches of data through the network and computes the resulting distributions of the different activations.
  - This is done by inserting "observer" modules at different points that record these distributions.
- This information is used to determine how specifically the different activations should be quantized at **inference time**.
  - A simple technique would be to simply divide the entire range of activations into 256 levels, but PyTorch supports more sophisticated methods as well.
- This step allows to pass quantized values between operations instead of converting these values to floats - and then back to ints - between every operation, resulting in a significant speed-up.
- Optimizing static quantization includes:
  - **Observers**: observer modules specify how statistics are collected prior to quantization to try out more advanced methods to quantize the data.
  - **Operator fusion**: fuse multiple operations into a single operation, saving on memory access while also improving the operation's numerical accuracy.
  - **Per-channel quantization**: we can independently quantize weights for each output channel in a convolution/linear layer, which can lead to higher accuracy with almost the same speed.

In [ ]:
```
# the three lines that perform post-training static quantization on the pre-trained model myModel
# set quantization config for server (x86) deployment
myModel.qconfig = torch.quantization.get_default_config('fbgemm')

# insert observers
torch.quantization.prepare(myModel, inplace=True)
# Calibrate the model and collect statistics

# convert to quantized version
torch.quantization.convert(myModel, inplace=True)
```

- Examples (project ideas!): [Static quantization with eager execution (https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html)](https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html), [Quantized transfer learning (https://pytorch.org/tutorials/intermediate/quantized_transfer_learning_tutorial.html)](https://pytorch.org/tutorials/intermediate/quantized_transfer_learning_tutorial.html).

Quantization Aware Training (QAT)

- The quantization method that typically results in highest accuracy of the three methods.
- With QAT, all weights and activations are "fake quantized" during both the forward and backward passes of training: that is, **float values are rounded to mimic `int8` values, but all computations are still done with floating point numbers**.
- Thus, all the weight adjustments during training are made while "aware" of the fact that the model will ultimately be quantized; after quantizing, therefore, this method usually yields higher accuracy than the other two methods.
- In PyTorch: `torch.quantization.prepare_qat` inserts fake quantization modules to model quantization and `torch.quantization.convert` actually quantizes the model once training is complete.



[Image Source (https://towardsdatascience.com/inside-quantization-aware-training-4f91c8837ead)](https://towardsdatascience.com/inside-quantization-aware-training-4f91c8837ead)

In [ ]:
```
# specify quantization config for QAT
qat_model.qconfig = torch.quantization.get_default_qat_qconfig('fbgemm')

# prepare QAT
torch.quantization.prepare_qat(qat_model, inplace=True)

# convert to quantized version, removing dropout, to check for accuracy on each epoch
quantized_model = torch.quantization.convert(qat_model.eval(), inplace=False)
```

- Example (project ideas): [Static quantization with eager execution (https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html#quantization-aware-training)](https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html#quantization-aware-training).

Important Notes for Quantization in PyTorch

- Quantization support is restricted to a subset of available operators, depending on the method being used, for a list of supported operators, see the [documentation (https://pytorch.org/docs/stable/quantization.html)](https://pytorch.org/docs/stable/quantization.html).
- The set of available operators and the quantization numerics also depend on the backend being used to run quantized models.
- Currently quantized operators are supported only for **CPU inference** in the following backends: x86 and ARM.
- QAT is typically only used in CNN models when post training static or dynamic quantization doesn't yield sufficient accuracy. This can occur with models that are highly optimized to achieve small size.

Which Quantization Method to Use?

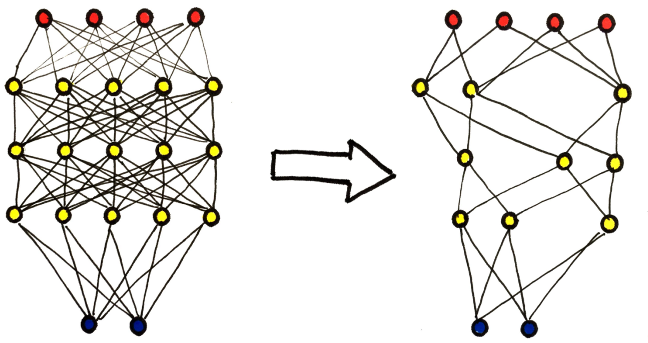
Model Type	Preferred Scheme	Why
LSTM/RNN	Dynamic Quantization	Throughput dominated by compute/memory bandwidth for **weights**
BERT/Transformer	Dynamic Quantization	Throughput dominated by compute/memory bandwidth for **weights**
CNN	Static Quantization	Throughput limited by memory bandwidth for **activations**
CNN	Quantization Aware Training	In the case where accuracy can't be achieved with static quantization

Performance Results

Model Type	Float Latency (ms)	Quantized Latency (ms)	Inference Performance Gain	Accuracy	Device
BERT	581	313	1.8x	F1 score: 0.902 → 0.895 (dynamic quantization)	Xeon-D2191 (1.6GHz)
Resnet-50	214	103	2x	Top 1 Acc: 76.1 → 75.9 (static post-training quantization)	Xeon-D2191 (1.6GHz)
Mobilenet-v2	97	17	5.7x	Top 1 Acc: 71.9 → 71.6 (QAT)	Samsung S9

Post Train-time Efficiency - Pruning

- Neural network pruning is the process of sparsifying neural networks from pre-trained dense neural networks, and is very similar to the pruning process of decision trees.
  - Instead of creating smaller trees, we create smaller computation graphs post-training.
- Pruning is used to investigate the differences in learning dynamics between over-parametrized and under-parametrized networks, to study the role of *lucky* sparse subnetworks and initializations ("lottery tickets") as a neural architecture search technique.



[Image Source (https://towardsdatascience.com/how-to-compress-a-neural-network-427e8dddcc34)](https://towardsdatascience.com/how-to-compress-a-neural-network-427e8dddcc34)

The Lottery Ticket Hypothesis

The ["Lottery Ticket Hypothesis" (https://arxiv.org/abs/1803.03635)](https://arxiv.org/abs/1803.03635) (Jonathan Frankle and Michael Carbin, 2008):

> A randomly-initialized, dense neural network contains a subnetwork that is initialized such that — when trained in isolation — it can match the test accuracy of the original network after training for at most the same number of iterations.

- Such subnetworks are called *winning lottery tickets*.
- If this hypothesis is true, and such subnetworks can be found, training could be done much faster and cheaper, since a single iteration step would take less computation.

Iterative Pruning Algorithm to find *winning lottery tickets*:

1. Randomly initialize the network and store the initial weights for later reference.
2. Train the network for a given number of steps.
3. Remove a percentage of the weights (prune) with the **lowest magnitude**.
4. Restore the remaining weights to the value that was given during the first initialization.
5. Go to **Step 2** and iterate the pruning.

Pruning in PyTorch

- We will follow an [example (https://pytorch.org/tutorials/intermediate/pruning_tutorial.html)](https://pytorch.org/tutorials/intermediate/pruning_tutorial.html) by [Michela Paganini (https://mickypaganini.github.io/)](https://mickypaganini.github.io/).
- We will use PyTorch pruning name-space - `torch.nn.utils.prune` .

In [2]:
```python
# our model for the demonstartion
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square conv kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # 5x5 image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, int(x.nelement() / x.shape[0]))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = LeNet().to(device=device)
```

To prune a module (in this example, the `conv1` layer of the LeNet architecture):

1. Select a pruning technique among those available in `torch.nn.utils.prune` (or implement your own by subclassing `BasePruningMethod`).
2. Specify the module and the name of the parameter to prune within that module.
3. Finally, using the adequate keyword arguments required by the selected pruning technique, specify the pruning parameters.

- In this example, we will prune **at random** 30% of the connections in the parameter named `weight` in the `conv1` layer.
- The module is passed as the first argument to the function.
  - `name` identifies the parameter within that module using its string identifier.
  - `amount` indicates either the percentage of connections to prune (if it is a float between 0. and 1.), or the absolute number of connections to prune (if it is a non-negative integer).
  - `prune.random_unstructured(module, name="weight", amount=0.3)`
- Pruning acts by removing `weight` from the parameters and replacing it with a new parameter called `weight_orig` (i.e. appending "_orig" to the initial parameter name). `weight_orig` stores the unpruned version of the tensor.
  - The `bias` was not pruned, so it will remain intact.

In [3]:
```python
module = model.conv1
print("before pruning:")
print(list(module.named_parameters()))
prune.random_unstructured(module, name="weight", amount=0.3)
print("after pruning:")
print(list(module.named_parameters()))
```

```
before pruning:
[('weight', Parameter containing:
tensor([[[[-0.0020,  0.0248,  0.2009],
          [ 0.2769,  0.1085, -0.1986],
          [ 0.3300,  0.3240,  0.2869]]],


        [[[ 0.3019,  0.0024, -0.0397],
          [ 0.0089,  0.1513, -0.2677],
          [-0.3236, -0.2904, -0.0882]]],


        [[[ 0.1837,  0.1874,  0.1020],
          [-0.0874, -0.2104, -0.2653],
          [ 0.1155, -0.0086,  0.2760]]],


        [[[ 0.2929,  0.0033,  0.1322],
          [-0.0203,  0.1317, -0.0450],
          [-0.2536, -0.0920, -0.0615]]],


        [[[ 0.0360,  0.0032, -0.0793],
          [ 0.2970, -0.0857, -0.1989],
          [-0.0124, -0.3124, -0.1621]]],


        [[[-0.0620,  0.2461,  0.2737],
          [ 0.0753,  0.2653,  0.3236],
          [ 0.0951,  0.3321,  0.0862]]]], device='cuda:0', requires_grad=True)), ('bias', Parameter conta
ining:
tensor([ 0.1776, -0.1092, -0.1042,  0.2878, -0.2541,  0.2821], device='cuda:0',
       requires_grad=True))]
after pruning:
[('bias', Parameter containing:
tensor([ 0.1776, -0.1092, -0.1042,  0.2878, -0.2541,  0.2821], device='cuda:0',
       requires_grad=True)), ('weight_orig', Parameter containing:
tensor([[[[-0.0020,  0.0248,  0.2009],
          [ 0.2769,  0.1085, -0.1986],
          [ 0.3300,  0.3240,  0.2869]]],


        [[[ 0.3019,  0.0024, -0.0397],
          [ 0.0089,  0.1513, -0.2677],
          [-0.3236, -0.2904, -0.0882]]],


        [[[ 0.1837,  0.1874,  0.1020],
          [-0.0874, -0.2104, -0.2653],
          [ 0.1155, -0.0086,  0.2760]]],


        [[[ 0.2929,  0.0033,  0.1322],
          [-0.0203,  0.1317, -0.0450],
          [-0.2536, -0.0920, -0.0615]]],


        [[[ 0.0360,  0.0032, -0.0793],
          [ 0.2970, -0.0857, -0.1989],
          [-0.0124, -0.3124, -0.1621]]],


        [[[-0.0620,  0.2461,  0.2737],
          [ 0.0753,  0.2653,  0.3236],
          [ 0.0951,  0.3321,  0.0862]]]], device='cuda:0', requires_grad=True))]
```

- The pruning mask generated by the pruning technique selected above is saved as a **module buffer** named `weight_mask` (i.e. appending "_mask" to the initial parameter name).

In [12]:
```python
print(list(module.named_buffers()))
```

```
[('bias_mask', tensor([0., 1., 1., 0., 1., 0.], device='cuda:0')), ('weight_mask', tensor([[[[1., 1., 1.],
          [0., 1., 1.],
          [1., 1., 1.]]],


        [[[1., 0., 0.],
          [0., 0., 1.],
          [1., 1., 1.]]],


        [[[1., 1., 1.],
          [1., 1., 1.],
          [1., 1., 1.]]],


        [[[0., 0., 0.],
          [1., 1., 1.],
          [0., 1., 1.]]],


        [[[1., 1., 0.],
          [1., 0., 1.],
          [0., 1., 1.]]],


        [[[1., 1., 1.],
          [1., 0., 0.],
          [0., 0., 1.]]]], device='cuda:0'))]
```

- For the forward pass to work *without modification*, the `weight` attribute needs to exist.
- The pruning techniques implemented in `torch.nn.utils.prune` compute the pruned version of the weight (by combining the mask with the original parameter) and store them in the attribute `weight` .
- Note, **this is no longer a parameter of the module**, it is now simply an attribute.

In [13]:
```python
print(module.weight)
```

```
tensor([[[[-0.2732,  0.0000,  0.2346],
          [-0.0000, -0.1691, -0.0000],
          [-0.0000, -0.0000,  0.0000]]],


        [[[-0.0073,  0.0000, -0.0000],
          [-0.0000,  0.0000,  0.2844],
          [-0.2542, -0.0446,  0.2390]]],


        [[[-0.1039,  0.1222,  0.0302],
          [-0.0933,  0.2147, -0.1062],
          [-0.2784,  0.1784, -0.1429]]],


        [[[-0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0071],
          [-0.0000,  0.0930, -0.3321]]],


        [[[ 0.0193,  0.0072,  0.0000],
          [-0.0000, -0.0000,  0.2329],
          [ 0.0000,  0.1394,  0.0000]]],


        [[[-0.1514,  0.2017,  0.0000],
          [ 0.0000,  0.0000,  0.0000],
          [-0.0000, -0.0000, -0.0072]]]], device='cuda:0',
       grad_fn=<MulBackward0>)
```

- Finally, pruning is applied **prior to each forward pass** using PyTorch's `forward_pre_hooks` .
- Specifically, when the `module` is pruned, as we have done here, it will acquire a `forward_pre_hook` for each parameter associated with it that gets pruned.
- In this case, since we have so far only pruned the original parameter named `weight` , only one hook will be present.

In [14]:
```
print(module._forward_pre_hooks)
```

```
OrderedDict([(1, <torch.nn.utils.prune.L1Unstructured object at 0x000001F4094BD978>), (2, <torch.nn.util
s.prune.RandomUnstructured object at 0x000001F4094BDF28>)])
```

- We can now prune the `bias` too, to see how the parameters, buffers, hooks, and attributes of the module change.
- Just for the sake of trying out another pruning technique, here we prune the **3 smallest entries** in the `bias` by **L1 norm**, as implemented in the `l1_unstructured` pruning function.

In [15]:
```python
prune.l1_unstructured(module, name="bias", amount=3)
print('named parameters:')
print(list(module.named_parameters()))  # notice the bias_orig
print('named buffers:')
print(list(module.named_buffers()))  # notice the bias_mask
print('module.bias:')
print(module.bias)
print('forward pre hooks:')
print(module._forward_pre_hooks)
```

```
named parameters:
[('bias_orig', Parameter containing:
tensor([ 0.1057,  0.1858, -0.2802,  0.0499, -0.2165,  0.0779], device='cuda:0',
       requires_grad=True)), ('weight_orig', Parameter containing:
tensor([[[[-0.2732,  0.0000,  0.2346],
          [-0.1208, -0.1691, -0.0000],
          [-0.0000, -0.0000,  0.0000]]],


        [[[-0.0073,  0.1105, -0.0000],
          [-0.0686,  0.0000,  0.2844],
          [-0.2542, -0.0446,  0.2390]]],


        [[[-0.1039,  0.1222,  0.0302],
          [-0.0933,  0.2147, -0.1062],
          [-0.2784,  0.1784, -0.1429]]],


        [[[-0.2053,  0.1681, -0.0000],
          [ 0.0000, -0.0000,  0.0071],
          [-0.0623,  0.0930, -0.3321]]],


        [[[ 0.0193,  0.0072,  0.2102],
          [-0.0000, -0.0172,  0.2329],
          [ 0.1754,  0.1394,  0.0000]]],


        [[[-0.1514,  0.2017,  0.0000],
          [ 0.0000,  0.2775,  0.0000],
          [-0.0000, -0.3004, -0.0072]]]], device='cuda:0', requires_grad=True))]
named buffers:
[('bias_mask', tensor([0., 0., 0., 0., 0., 0.], device='cuda:0')), ('weight_mask', tensor([[[[1., 1.,
1.],
          [0., 1., 1.],
          [1., 1., 1.]]],


        [[[1., 0., 0.],
          [0., 0., 1.],
          [1., 1., 1.]]],


        [[[1., 1., 1.],
          [1., 1., 1.],
          [1., 1., 1.]]],


        [[[0., 0., 0.],
          [1., 1., 1.],
          [0., 1., 1.]]],


        [[[1., 1., 0.],
          [1., 0., 1.],
          [0., 1., 1.]]],


        [[[1., 1., 1.],
          [1., 0., 0.],
          [0., 0., 1.]]]], device='cuda:0'))]
module.bias:
tensor([0., 0., -0., 0., -0., 0.], device='cuda:0', grad_fn=<MulBackward0>)
forward pre hooks:
OrderedDict([(2, <torch.nn.utils.prune.RandomUnstructured object at 0x000001F4094BDF28>), (3, <torch.nn.u
tils.prune.PruningContainer object at 0x000001F4094E7438>)])
```

- You can now try different methods to prune your trained model.
- For more examples (project ideas!): [Iterative Pruning (https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#iterative-pruning)](https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#iterative-pruning), [Pruning Multiple Parameters in a Model (https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#pruning-multiple-parameters-in-a-model)](https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#pruning-multiple-parameters-in-a-model), [Global Pruning (pruning all weights at once) (https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#global-pruning)](https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#global-pruning), [Custom Pruning Functions (https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#extending-torch-nn-utils-prune-with-custom-pruning-functions)](https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#extending-torch-nn-utils-prune-with-custom-pruning-functions).

- Making the pruning **permanent** by removing pruning re-parametrization.
  - This means removing the re-parametrization in terms of `weight_orig` and `weight_mask`, and remove the `forward_pre_hook`.
- We can use the `remove` functionality from `torch.nn.utils.prune`.
  - Note that this doesn’t undo the pruning, as if it never happened. It simply makes it permanent, instead, by reassigning the parameter weight to the model parameters, in its pruned version.

In [16]:
```python
prune.remove(module, 'weight')
print(list(module.named_parameters()))  # only bias_orig remains
print(list(module.named_buffers()))  # only bias_mask remains
```

```
[('bias_orig', Parameter containing:
tensor([ 0.1057,  0.1858, -0.2802,  0.0499, -0.2165,  0.0779], device='cuda:0',
       requires_grad=True)), ('weight', Parameter containing:
tensor([[[[-0.2732,  0.0000,  0.2346],
          [-0.0000, -0.1691, -0.0000],
          [-0.0000, -0.0000,  0.0000]]],


        [[[-0.0073,  0.0000, -0.0000],
          [-0.0000,  0.0000,  0.2844],
          [-0.2542, -0.0446,  0.2390]]],


        [[[-0.1039,  0.1222,  0.0302],
          [-0.0933,  0.2147, -0.1062],
          [-0.2784,  0.1784, -0.1429]]],


        [[[-0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0071],
          [-0.0000,  0.0930, -0.3321]]],


        [[[ 0.0193,  0.0072,  0.0000],
          [-0.0000, -0.0000,  0.2329],
          [ 0.0000,  0.1394,  0.0000]]],


        [[[-0.1514,  0.2017,  0.0000],
          [ 0.0000,  0.0000,  0.0000],
          [-0.0000, -0.0000, -0.0072]]]], device='cuda:0', requires_grad=True))]
[('bias_mask', tensor([0., 0., 0., 0., 0., 0.], device='cuda:0'))]
```

Recommended Videos

Warning!

- These videos do not replace the lectures and tutorials.
- Please use these to get a better understanding of the material, and not as an alternative to the written material.

Video By Subject

- Autimatic Mixed Precision (AMP) - [NVIDIA - Automatic Mixed Precision Training in PyTorch (https://www.youtube.com/watch?v=b5dAmcBKxHg)](https://www.youtube.com/watch?v=b5dAmcBKxHg)
- Quantization - [Deep Dive on PyTorch Quantization - Chris Gottbrath (https://www.youtube.com/watch?v=c3MT2qV5f9w)](https://www.youtube.com/watch?v=c3MT2qV5f9w)
- Pruning - [Neural Network Pruning for Compression and Understanding - Facebook AI Research - Dr. Michela Paganini (https://www.youtube.com/watch?v=f86hkOGoX54)](https://www.youtube.com/watch?v=f86hkOGoX54)
  - Pruning - [PyTorch Pruning - How it's Made by Michela Paganini (https://www.youtube.com/watch?v=TaOwEa3m5dw)](https://www.youtube.com/watch?v=TaOwEa3m5dw)

Credits

- Icons made by Becris (https://www.flaticon.com/authors/becris) from www.flaticon.com (https://www.flaticon.com/)
- Icons from Icons8.com (https://icons8.com/) - https://icons8.com (https://icons8.com)
- NVIDIA - Mixed Precision (https://developer.nvidia.com/automatic-mixed-precision)
- Introduction to Quantization in PyTorch (https://pytorch.org/blog/introduction-to-quantization-on-pytorch/) - by Raghuraman Krishnamoorthi, James Reed, Min Ni, Chris Gottbrath, and Seth Weidman.
- Tivadar Danka - How to Compress a Neural Network (https://towardsdatascience.com/how-to-compress-a-neural-network-427e8dddcc34)
- Michela Paganini (Michela Paganini) - Pruning Tutorial (https://pytorch.org/tutorials/intermediate/pruning_tutorial.html)