Appendix B. Mixed Precision and Quantization

By default, PyTorch uses 32-bit floats to represent model parameters: that's 4 bytes per parameter. If your model has 1 billion parameters, then you need at least 4 GB of RAM just to hold the model. At inference time you also need enough RAM to store the activations, and at training time you need enough RAM to store all the intermediate activations as well (for the backward pass), and to store the optimizer parameters (e.g., Adam needs two additional parameters for each model parameter—that's an extra 8 GB). This is a lot of RAM, and it's also plenty of time spent transferring data between the CPU and the GPU, not to mention storage space, download time, and energy consumption.

So how can we reduce the model's size? A simple option is to use a reduced precision float representation—typically 16-bit floats instead of 32-bit floats. If you train a 32-bit model then shrink it to 16-bits after training, its size will be halved, with little impact on its quality. Great!

However, if you try to train the model using 16-bit floats, you may run into convergence issues, as we will see. So a common strategy is *mixed-precision training* (MPT), where we keep the weights and weight updates at 32-bit precision during training, but the rest of the computations use 16-bit precision. After training, we shrink the weights down to 16-bits.

Finally, to shrink the model even further, you can use *quantization*: the parameters are discretized and represented as 8-bit integers, or even 4-bit integers or less. This is harder, and it degrades the model's quality a bit more, but it reduces the model size by a factor of 4 or more, and speeds it up significantly.

In this appendix, we will cover reduced precision, mixed precision training, and quantization. But to fully understand these, we must first discuss common number representations in Machine Learning.

Common Number Representations

By default, PyTorch represents weights and activations using 32-bit floats based on the *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), which specifies how floating point numbers are represented in memory. It's a flexible and efficient format which can represent tiny values and huge values, as well as special values such as ±0,[1] ±infinity, and NaN (i.e., Not a Number).

The float32 data type (fp32 for short) can hold numbers as small as $\pm 1.4e^{-45}$ and as large as $\pm 3.4e^{38}$. It is represented at the top of Figure B-1. The first bit determines the *sign S*: 0 means positive, 1 means negative. The next 8 bits hold the *exponent E*, ranging from 0 to 255. And the last 23 bits represent the *fraction F*, ranging from 0 to $2^{23} - 1$. Here is how to compute the value:

- If E is between 1 and 254, then the number is called *normalized*: this is the most common scenario. In this case, the value v can be computed using $v = (-1)^S \cdot 2^{E-127} \cdot (1 + F \cdot 2^{-23})$. The last term $(1 + F \cdot 2^{-23})$ corresponds to the most significant digits, so it's called the *significand*.
- If $E = 0$ and $F > 0$, then the number is called *subnormal*: it is useful to represent the tiniest values.[2] In this case, $v = (-1)^S \cdot 2^{E+1-127} \cdot (0 + F \cdot 2^{-23}) = (-1)^S \cdot F \cdot 2^{-149}$.
- If $E = 0$ and $F = 0$, then $v = \pm 0$.
- If $E = 255$ and $F > 0$, then v = NaN.
- If $E = 255$ and $F = 0$, then v = ±infinity.

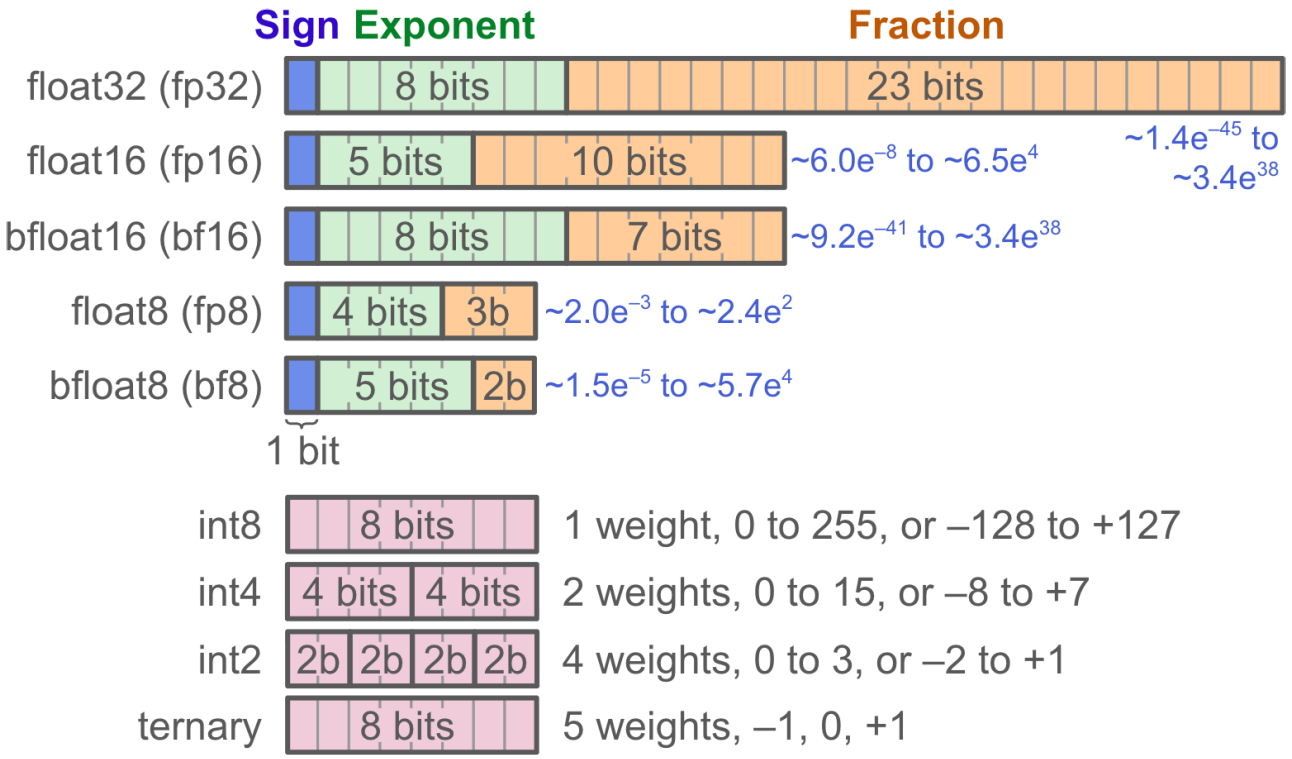

Figure B-1. Common number representations in Machine Learning

The other floating point formats represented in Figure B-1 differ only by the number of bits used for the exponent and the fraction. For example float16 uses 5 bits for the exponent (i.e., it ranges from 0 to 31) and 10 bits for the fraction (ranging from 0 to 1,023), while float8 uses 4 bits for the exponent (from 0 to 15) and 3 bits for the fraction, so it's often noted fp8 E4M3.[3] The equations to compute the value are adjusted accordingly, for example normalized float16 values are computed using $v = (-1)^S \cdot 2^{E-15} \cdot (1 + F \cdot 2^{-10})$.

The bfloat16 and bfloat8 formats were proposed by Google Brain (hence the *b*), and they offer a wider range for the values, at the cost of a significantly reduced precision. We will come back to that.

Integers are often represented using 64 bits, with values ranging from 0 to $2^{64} - 1$ (about $1.8e^{19}$) for unsigned integers, or –2^32 to 2^32 – 1 (about $\pm 4.3e^{9}$) for signed integers. Integers are also frequently represented using 32 bits, 16 bits, or 8 bits depending on the use case. In Figure B-1, I only represented the integer types frequently used for quantization, such as 8-bit integers (which can be unsigned or signed).

When quantizing down to 4 bits, we usually pack 2 weights per byte, and when quantizing down to 2 bits, we pack 4 weights per byte. It's even possible to quantize down to ternary val-

ues, where each weight can only be equal to –1, 0, or +1. In this case, it's common to store five weights per byte. For example, the byte 178 can be written as 20121 in base 3 (since $178 = 2 \cdot 3^4 + 0 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0$), and if we subtract 1 from each digit, we get 1, –1, 0, 1, 0: these are the 5 ternary weights stored in this single byte. This format only uses 1.6 bits per weight on average, which is 20 times less than using 32-bit floats!

It's technically possible to quantize weights down to a single bit each, storing 8 weights per byte: each bit represents a weight equal to either –1 or +1 (or sometimes 0 or 1). However, it's very difficult to get reasonable accuracy using such severe quantization.

As you can see, PyTorch's default weight representation (32-bit floats) takes up a *lot* of space compared to other representations: there is room for us to shrink our models quite a bit! Let's start by reducing the precision from 32 bits down to 16 bits.

Reduced Precision Models

If you have a 32-bit PyTorch model, you can convert all of its parameters to 16-bit floats—which is called *half-precision*—by calling the model's `half()` method:

```
import torch
import torch.nn as nn

model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(), nn.Linear(100, 1))
# [...] pretend the 32-bit model is trained here
model.half()  # convert the model parameters to half precision (16 bits)
```

This is a quick and easy way to halve the size of a trained model, usually without much impact on its quality. Moreover, since many GPUs have 16-bit float optimizations, and since there will be less data to transfer between the CPU and the GPU, the model will typically run almost twice as fast.

TIP

When downloading a pretrained model using the Transformers library's `from_pretrained()` method, you can set `torch_dtype="auto"` to let the library choose the optimal float representation for your hardware.

To use the model, you now need to feed it 16-bit inputs, and it will output 16-bit outputs as well:

```
X = torch.rand(3, 10, dtype=torch.float16)  # some 16-bit input
y_pred = model(X)  # 16-bit output
```

But what if you want to build and train a 16-bit model right from the start? In this case, you

But what if you want to build and train a 16-bit model right from the start? In this case, you can set `dtype=torch.float16` whenever you create a tensor or a module with parameters, for example:

```
model = nn.Sequential(nn.Linear(10, 100, dtype=torch.float16), nn.ReLU(),
                      nn.Linear(100, 1, dtype=torch.float16))
```

TIP

If you prefer to avoid repeating `dtype=torch.float16` everywhere, then you can instead set the default data type to `torch.float16` using `torch.set_default_dtype(torch.float16)`. Be careful: this will apply to *all* tensors and modules created after that.

However, the reduced precision can cause some issues during training. Indeed, 16-bit floats have a limited *dynamic range* (i.e., the ratio between the largest and smallest positive representable values): the smallest positive representable value is about 0.00000006 (i.e., $6.0e^{-8}$), while the largest is 65,504 (i.e., ~$6.5e^{4}$). This implies that any gradient update smaller than ~$6.0e^{-8}$ will *underflow*, meaning it will be rounded down to zero, and thus ignored. And conversely, any value larger than ~$6.5e^{4}$ will *overflow*, meaning it will be rounded up to infinity, causing training to fail (once some weights are infinite, the loss will be infinite or NaN).

To avoid underflows, one solution is to scale up the loss by a large factor (e.g., multiply it by 256): this will automatically scale up the gradients by the same factor during the backward pass, which will prevent them from being smaller than the smallest 16-bit representable value. However, you must scale the gradients back down before performing an optimizer step, and at this point you may get an underflow. Also, if you scale up the loss too much, you will run into overflows.

If you can't find a good scaling factor that avoids both underflows and overflows, you can try to use `torch.bfloat16` rather than `torch.float16`, since bloat16 has more bits for the exponent: the smallest value is ~$9.2e^{-41}$, while the largest is ~$3.4e^{38}$, so there's less risk of any significant gradient updates being ignored, or reasonable values being rounded up to infinity.

However, bfloat16 has historically had less hardware support (although this is improving), and it offers fewer bits for the fraction, which can cause some gradient updates to be ignored when the corresponding parameter values are much larger, causing training to stall. For example, if the gradient update is $4.5e^{-2}$ (i.e., 0.045) and the corresponding parameter value is equal to $1.23e^{2}$ (i.e., 123), then the sum should be $1.23045e^{2}$ (i.e., 123.045) but bfloat16 does not have enough fraction bits to store all these digits, so it must round the result to $1.23e^{2}$ (i.e., 123): as you can see, the gradient update is completely ignored. With regular 16-bit floats, the result would be 123.0625, which is not exactly right due to floating point precision errors, but at least the parameter makes a step in the right direction. That said, if the gradient update was a bit smaller (e.g., 0.03), it would be ignored even in regular 16-bit float precision.

So if you try float16 and bfloat16 but you still encounter convergence issues during training,

then you can try *mixed precision training* instead.

Mixed Precision Training

Mixed precision training (MPT) was proposed by Baidu and Nvidia researchers in 2017,[4] to address the issues often observed with 16-bit training. Here's how it works:

- MPT stores a primary copy of the model parameters as 32-bit floats, and at each training iteration, it creates a 16-bit copy of these model parameters (see step 1 in Figure B-2), and uses them for the forward pass (step 2).
- The loss is then scaled up by a large factor (step 3) to avoid underflows, as we discussed earlier.
- Lastly, we switch back to 32-bit precision to scale the gradients back down: this greater precision avoids the risk of underflow. Next we use the gradients to perform one optimizer step, improving the primary parameters (step 5). Performing the actual optimizer step in 32-bit precision ensures that small weight updates are not ignored when applied to much larger parameter values, since 32-bit floats have a very large fraction (23 bits).

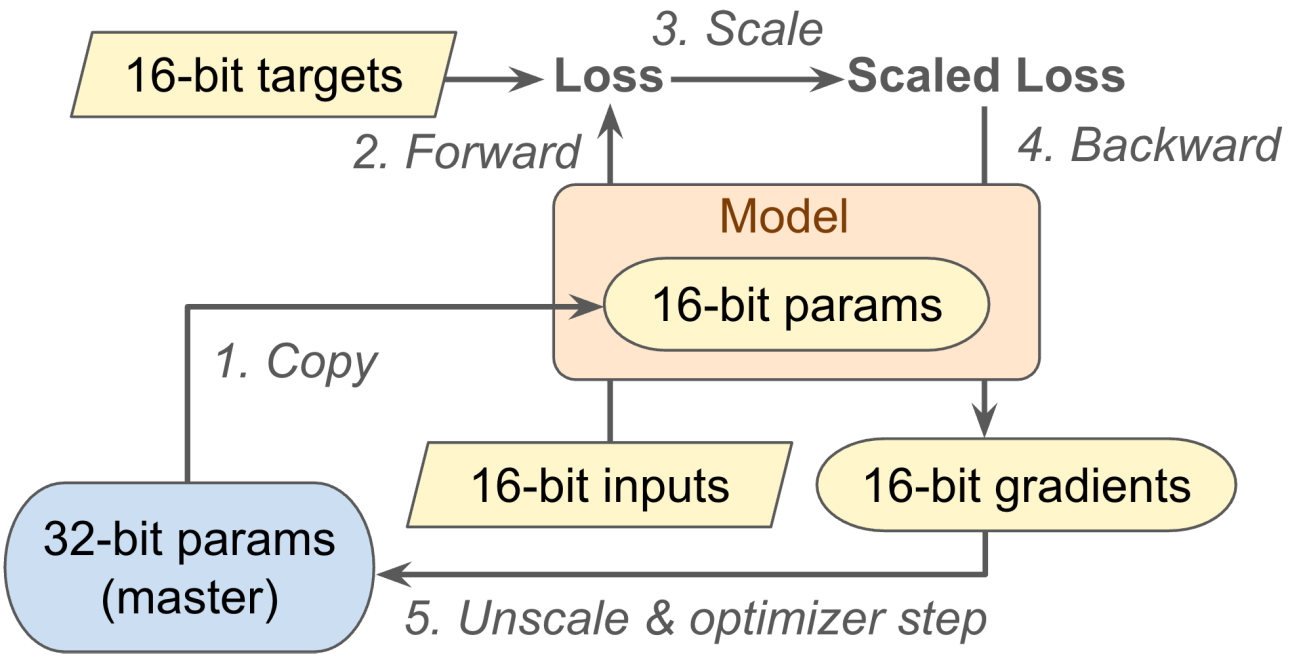

Figure B-2. Mixed precision training

MPT offers almost all of the benefits of 16-bit training, without the instabilities. However, the model parameters take 50% more space than in 32-bit training because of the 16-bit copy at each training iteration, so how is this any better? Well, during training, most of the RAM is used to store the activations, not the model parameters, so in practice MPT requires just a bit more than half the RAM used by regular 32-bit training. And it typically runs twice as fast. Moreover, once training is finished, we no longer need 32-bit parameters, we can convert them to 16 bits, and we get a pure 16-bit model.

WARNING

MPT does not always accelerate training: it depends on the model, the batch size, and the hardware. That said, most large transformers are trained using MPT.

Rather than finding the best scaling factor by trial and error, you can run training in 32-bit precision for a little while (assuming you have enough RAM) and measure the gradient statistics to find the optimal scaling factor for your task: it should be large enough to avoid underflows, and small enough to avoid overflows.

Alternatively, your training script can adapt the factor dynamically during training: if some gradients are infinite or NaN, this means that an overflow occurred so the factor must be reduced (e.g., halved) and the training step must be skipped, but if no overflow is detected then the scaling factor can be gradually increased (e.g., doubled every 2,000 training steps). PyTorch provides a `torch.amp.GradScaler` class that implements this approach, and also scales down the learning rate appropriately (we will see an example in the next section).[5]

PyTorch also provides a `torch.autocast()` function that returns a context within which many operations will automatically run in 16-bit precision. This includes operations that typically benefit the most from 16-bit precision, such as matrix multiplication and convolutions, but it does not include operations like reductions (e.g., `torch.sum()`) since running these in half precision offers no significant benefit and can damage precision.

Let's update our training function to run the forward pass within an autocast context and use a `GradScaler` to dynamically scale the loss:

```
from torch.amp import GradScaler

def train_mpt(model, optimizer, criterion, train_loader, n_epochs,
              dtype=torch.float16, init_scale=2.0**16):
    grad_scaler = GradScaler(device=device, init_scale=init_scale)
    model.train()
    for epoch in range(n_epochs):
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            with torch.autocast(device_type=device, dtype=dtype):
                y_pred = model(X_batch)
                loss = criterion(y_pred, y_batch)
            grad_scaler.scale(loss).backward()
            grad_scaler.step(optimizer)
            grad_scaler.update()
            optimizer.zero_grad()
```

Reducing precision down to 16-bits often works great, but can we shrink our models even further? Yes, we can, using quantization.

Quantization

Quantization means mapping continuous values to discrete ones. In deep learning, this typi

Quantization means mapping continuous values to discrete ones. In deep learning, this typically involves converting parameters, and often activations as well, from floats to integers—usually 32-bit floats to 8-bit integers. More generally, the goal is to shrink and speed up our model by reducing the number of bits used in parameters, and often in activations as well. Moreover, some mobile devices and embedded devices do not support floating point operations at all (in part to reduce their cost and energy consumption), so models have to be quantized entirely (both weights and activations) before you can use them on the device.

The simplest approach is *linear quantization*, so we'll discuss it now. We will discuss a few nonlinear quantization methods later in this appendix.

Linear Quantization

Linear quantization dates back to digital signal processing in the 1950s, but it has become particularly important in Machine Learning over the past decade since models have become gigantic, and yet we wish to run them on mobile phones and other limited devices. It has two variants: asymmetric and symmetric. In *asymmetric linear quantization*, float values are simply mapped linearly to unsigned bytes with values ranging from 0 to 255 (or more generally from 0 to $2^n - 1$ when quantizing to n-bit integers). For example, if the weights range between $a = -0.1$ and $b = 0.6$, then the float –0.1 will be mapped to the byte 0, the float 0.0 to integer 36, 0.1 to 72, …, 0.6 to 255, and more generally, the float tensor $\mathbf{w}$ will be mapped to the integer tensor $\mathbf{q}$ using Equation B-1.

Equation B-1. Asymmetric linear quantization

In this equation:

- w_i is the i^{th} float in the original tensor $\mathbf{w}$.
- q_i is the i^{th} integer in the quantized tensor $\mathbf{q}$. It is clampled between 0 and $2^n - 1$ (e.g., 255 for 8-bit quantization).
- s is the *quantization scale*. Note that some authors define it as $1 / s$ and adapt the equation accordingly (i.e., they multiply rather than divide).
- z is the *quantization bias* or *zero point*.
- a is the minimum value of $\mathbf{w}$, and b is the maximum value of $\mathbf{w}$.

The range $[a, b]$ is known for weights, since their values do not change after training. However, the range of activation values depends on the inputs we feed to the model. As a result, for each activation that we want to quantize (e.g., the inputs of each layer), we will either have to compute a and b on the fly for each new input batch–this is called *dynamic quantization*–or run a calibration dataset once through the model to determine the typical range of activation values, then use this range to quantize the activations of all subsequent batches–this is called *static quantization*. Static quantization is a faster but less precise

quantization. Static quantization is a faster but less precise.

To approximately recover the original value w_i from a quantized value q_i, we can compute $w_i \approx s \times (q_i - z)$. This is called *dequantization*. For example, if $q_i = 72$, then we get $w_i \approx 0.0988$, which is indeed close to 0.1. The difference between the dequantized value (0.0988) and the original value (0.1) is called the *quantization noise*: with 8-bit quantization, the quantization noise usually leads to a slightly degraded accuracy. With 6-bit, 4-bit, or less, the quantization noise can hurt even more, especially since it has a cumulative effect: the deeper the network, the stronger the impact.

NOTE

Equation B-1 guarantees that any float equal to 0.0 can be quantized and dequantized back to 0.0 exactly: indeed, if $w_i = 0.0$ then $q_i = z$, and dequantizing q_i gives back $w_i = s \times (z - z) = 0.0$. This is particularly useful for sparse models where many weights are equal to zero. It is also important when using activations like ReLU which produce many zero activations.

In PyTorch, the `torch.quantize_per_tensor()` function lets you create a quantized tensor: this is a special kind of tensor that contains the quantized values (i.e., integers), as well as the *quantization parameters* (i.e., the scale and zero point). Let's use this function to quantize a tensor, then dequantize it. In this example we will use the data type `torch.quint8`, which uses 8-bit unsigned integers:

```
>>> w = torch.tensor([0.1, -0.1, 0.6, 0.0])  # 32-bit floats
>>> s = (w.max() - w.min()) / 255.  # compute the scale
>>> z = -(w.min() / s).round()  # compute the zero point
>>> qw = torch.quantize_per_tensor(w, scale=s, zero_point=z, dtype=torch.quint8)
>>> qw  # this is a quantized tensor internally represented using integers
tensor([ 0.0988, -0.0988,  0.6012,  0.0000], size=(4,), dtype=torch.quint8,
       quantization_scheme=torch.per_tensor_affine, scale=0.002745098201557994,
       zero_point=36)
>>> qw.dequantize()  # back to 32-bit floats (close to the original tensor)
tensor([ 0.0988, -0.0988,  0.6012,  0.0000])
```

Quantizing a model to 8-bits divides its size by almost 4. For example, suppose we have a convolutional layer with 64 kernels, 3 × 3 each, and it has 32 input channels. This layer requires 64 × 32 × 3 × 3 = 18,432 parameters (ignoring the bias terms). That's 18,432 × 4 = 73,728 bytes before quantization, and just 18,432 bytes after quantization, plus 2 × 4 = 8 bytes to store s and z (indeed, they are both stored as 32 bit floats, so 4 bytes each).

TIP

PyTorch also has a `torch.quantize_per_channel()` function which quantizes each channel separately: this offers better precision but requires a bit more space for the additional quantization parameters.

When the float values are approximately symmetric around zero, we can use *symmetric linear quantization*, where the values are mapped between –127 and +127, or more generally between $-r$ and $+r$ with $r = 2^{n-1} - 1$, using Equation B-2.

Equation B-2. Symmetric linear quantization

To implement symmetric linear quantization in PyTorch, we can use the `torch.quantize_per_tensor()` function again, but using a zero point equal to 0, and data type `qint8` (quantized signed 8-bit integer):

```
>>> w = torch.tensor([0.0, -0.94, 0.92, 0.93])  # 32-bit floats
>>> s = w.abs().max() / 127.
>>> qw = torch.quantize_per_tensor(w, scale=s, zero_point=0, dtype=torch.qint8)
>>> qw
tensor([ 0.0000, -0.9400,  0.9178,  0.9326], size=(4,), dtype=torch.qint8,
       quantization_scheme=torch.per_tensor_affine, scale=0.007401574868708849,
       zero_point=0)
```

Figure B-3 shows some floats ranging between –0.94 and +0.93, quantized to signed bytes (i.e., 8-bits) ranging between –127 and +127,[6] using symmetric linear quantization. Notice that float 0.0 is always mapped to integer 0.

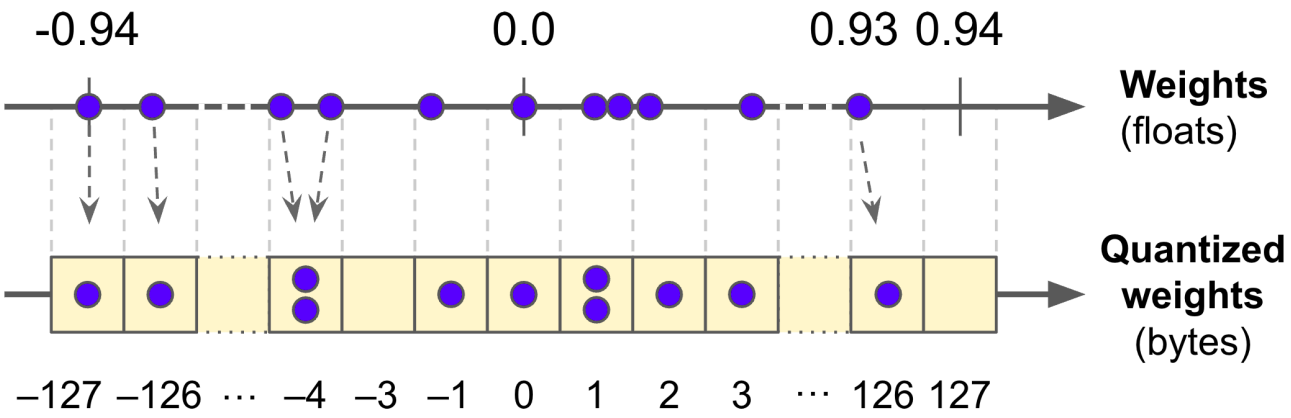

Figure B-3. Symmetric linear quantization

Symmetric mode is often a bit faster than asymmetric mode, because there's no zero point z to worry about. However, if the values are not symmetric, part of the integer range will be wasted. For example, if all the weights are positive, then symmetric mode will only use bytes 0 to 127 (rather than –127 to 127). As a result, symmetric mode can be a bit less precise than asymmetric mode. In practice, symmetric mode is generally preferred for weights (which are often fairly symmetric), and asymmetric mode for activations (especially when using ReLU, since it outputs only non-negative values).

$$q_i = \text{round}\left(\frac{w_i}{s}\right) \text{ with } s = \frac{\max_i |w_i|}{2^{n-1} - 1}$$

Let's now see how to quantize your models in practice using PyTorch's `torch.ao.quantization` package. The first approach is to quantize a trained model, which is called *Post-Training Quantization* (PTQ). The second is to train (or fine-tune) your model with

some fake quantization to get it used to the noise: this is called *Quantization-Aware Training* (QAT). Let's start with PTQ.

Post-Training Quantization Using `torch.ao.quantization`

The `torch.ao` package contains tools for architecture optimization (hence the name), including pruning, sparsity, and quantization. The `torch.ao.quantization` package offers two solutions to quantize trained models: dynamic quantization and static quantization. Let's see how to implement both.

Dynamic Quantization

Dynamic quantization is best for MLPs, RNNs and transformers. To implement it using PyTorch's `torch.ao.quantization` package, you must first choose a quantization engine: PyTorch currently supports the *Facebook General Matrix Multiplication* (FBGEMM) engine for x86 CPUs, plus a newer x86 engine that supports recent x86 CPUs but is less battle-tested, and finally the *Quantized Neural Networks Package* (QNNPACK) engine for ARM/mobile. This code will pick the appropriate engine depending on the platform:

```
import platform

machine = platform.machine().lower()
engine = "qnnpack" if ("arm" in machine or "aarch64" in machine) else "x86"
```

WARNING

PyTorch does not offer an engine for CUDA or other hardware accelerators, but other libraries do, such as the bitsandbytes library (as we will see shortly).

Once you have selected an engine, you can use the `quantize_dynamic()` function from the `torch.ao.quantization` package: just pass it your trained model, tell it the types of layers to quantize (typically just the `Linear` and RNN layers), specify the quantized data type, and boom, you have a ready-to-use quantized model:

```
from torch.ao.quantization import quantize_dynamic

model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(), nn.Linear(100, 1))
# [...] pretend the 32-bit model is trained here
torch.backends.quantized.engine = engine
qmodel = quantize_dynamic(model, {nn.Linear}, dtype=torch.qint8)
X = torch.randn(3, 10)
y_pred = qmodel(X)  # float inputs and outputs, but quantized internally
```

The `quantize_dynamic()` function replaces each `Linear` layer with a

`DynamicQuantizedLinear` layer, with int8 weights. This layer behaves just like a regular linear layer, with float inputs and outputs, but it quantizes its inputs on the fly (recomputing the zero points and scales for each batch), performs matrix multiplication using integers only (with 32-bit integer accumulators), and dequantizes the result so the next layer gets float inputs. Now let's look at static quantization.

Static Quantization

This option is best for CNNs, and max inference speed. It's also compulsory for edge devices without a *Floating-Point Unit* (FPU), as they don't support floats at all. Both the weights and activations are prepared for quantization ahead of time, for all layers. As we discussed earlier, weights are constant so they can be quantized once, while activations require a calibration step to determine their typical range. The model is then converted to a fully quantized model. Here is how to implement it:

```
from torch.ao.quantization import get_default_qconfig, QuantStub, DeQuantStub

model = nn.Sequential(QuantStub(),
                      nn.Linear(10, 100), nn.ReLU(), nn.Linear(100, 1),
                      DeQuantStub())
# [...] pretend the 32-bit model is trained here
model.qconfig = get_default_qconfig(engine)
torch.ao.quantization.prepare(model, inplace=True)
for X_batch, _ in calibration_loader:
    model(X_batch)
torch.ao.quantization.convert(model, inplace=True)
```

Let's go through this code step by step:

- After the imports, we create our 32-bit model, but this time we add a `QuantStub` layer as the first layer, and a `DeQuantStub` layer as the last. Both layers are just passthrough for now.
- Next, the model can be trained normally (another option would be to take a pretrained model and place it between a `QuantStub` layer and a `DeQuantStub` layer).
- Next, we set the model's `qconfig` to the output of the `get_default_qconfig()` function: this function takes the name of the desired quantization engine and returns a `QConfig` object containing a default quantization configuration for this engine. It specifies the quantization data type (e.g., `torch.qint8`), the quantization scheme (e.g., symmetric linear quantization per tensor), and two functions that will observe the weights and activations to determine their ranges.
- Next we call the `torch.ao.quantization.prepare()` function: it uses the weight observer specified in the configuration to determine the weights range, which it immediately uses to compute the zero points and scales for the weights. Since we don't know what the input data looks like at this point, the function cannot compute the quantization parameters for the activations yet, so it inserts activation observers in the model itself: these are attached to the outputs of the `QuantStub` and `Linear` layers. The observer appended to the

`QuantStub` layer is responsible for tracking the input range.

- Next, we take a representative sample of input batches (i.e., the kind the model will get in production), and we pass these batches through the model: this allows the activation observers to track the activations.
- Once we have given the model enough data, we finally call the `torch.ao.quantization.convert()` function, which removes the observers from the model and replaces the layers with quantized versions. The `QuantStub` layer is replaced with a `Quantize` layer which will quantize the inputs. The `Linear` layers are replaced with `QuantizedLinear` layers. And the `DeQuantStub` layer is replaced with a `DeQuantize` layer which will dequantize the outputs.

NOTE

There are a few observers to choose from: they can just keep track of the mininimum and maximum values for each tensor (`MinMaxObserver`), or for each channel (`PerChannelMinMaxObserver`), or they can compute an exponential moving average of the min/max values, which reduces the impact of a few ouliers. Finally, they can even record a histogram of the observed values (`HistogramObserver`), making it possible to find an optimal quantization range that minimizes the quantization error. That said, the default observers are usually fine.

We now have a model that we can use normally, with float inputs and outputs, but which works entirely with integers internally, making it lightweight and fast. To deploy it to mobile or embedded devices, there are many options to choose from (which are beyond the scope of this book), including:

- Use ExecuTorch, which is PyTorch's lightweight edge runtime
- Export the model to ONNX and run it with ONNX Runtime (cross-platform)
- Convert it to TFLite or TFLite Micro
- Compile it for the target device using TVM or microTVM

Moreover, the PyTorch team has released a separate library named *PyTorch-native Architecture Optimization* (TorchAO), designed to be a robust and extensible model optimization framework. Over time, many features in PyTorch's `torch.ao` package are expected to be migrated to—or superseded by—TorchAO. The library already includes advanced features such as 4-bit weight support and *per-block quantization*, in which each tensor is split into small blocks and each block is quantized independently, trading space for improved precision.

Post-training quantization (either dynamic or static) can shrink and speed up your models significantly, but it will also degrade their accuracy. This is particularly the case when quantizing down to 4 bits or less, and it's worse for static quantization than for dynamic quantization (which can at least adapt to each input batch independently). When the accuracy drop is unacceptable, you can try quantization-aware training, as we will discuss now.

Quantization-Aware Training (QAT)

QAT was introduced in a 2017 paper by Google researchers.[7] It rests upon a simple idea: why

QAT was introduced in a 2017 paper by Google researchers. It rests upon a simple idea: why not introduce some fake quantization noise during training so the model can learn to cope with it? After training, we can then quantize the model for real, and it should remain fairly accurate. QAT also makes it possible to quantize more agressively without losing too much accuracy, down to 4 bits, or even less. Sounds promising? Let's see how it can be done.

To add fake quantization noise to weights, we can simply quantize them and immediately dequantize them. For example, a weight equal to 0.42 might be quantized to the 4-bit integer 7, and immediately dequantized back to 0.39: we've successfully introduced quantization noise, and it's precisely the quantization noise that we would get if the model were really quantized. This fake quantization operation can be executed at each training step, and it can also be applied to some of the activations (e.g., to each layer output).

However, there is one little problem: quantization involves rounding to the nearest integer, and the rounding operation has gradients equal to zero (or undefined at integer boundaries), so gradient descent cannot make any progress. Luckily, we can sidestep this issue by using the *straight-through estimator* (STE) trick: during the backward phase, we pretend that the fake quantization operation was just the identity function, so the gradients flow straight through it untouched. This works because the loss landscape is generally fairly smooth locally, so gradients are likely to be similar within a small region around the quantized value, including at the original value.

Implementing QAT in PyTorch is fairly straightforward:

```
from torch.ao.quantization import get_default_qat_qconfig

model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(), nn.Linear(100, 1))
model.qconfig = get_default_qat_qconfig(engine)
torch.ao.quantization.prepare_qat(model, inplace=True)
train(model, optimizer, [...])  # train the model normally
torch.ao.quantization.convert(model.eval(), inplace=True)
```

After the import, we create our model, set its `qconfig` attribute to the default QAT configuration object for the chosen quantization engine, then we call the `prepare_qat()` function to add fake quantization operations to the model. This step also adds observers to determine the usual range of activation values. Next, we can train the model normally. Lastly, we switch the model to eval mode, and we call the `convert()` function to truly quantize it.

TIP

QAT doesn't have to be used during all of training: you can take a pretrained model and just fine-tune it for a few epochs using QAT, typically using a lower learning rate to avoid damaging the pretrained weights.

We've seen how to implement PTQ and QAT using PyTorch. However, it's only for CPUs. What if you want to run an LLM on a GPU that doesn't quite have enough RAM? A popular option is to

you want to run an LLM on a GPU that doesn't quite have enough RAM? A popular option is to use the bitsandbytes library by Hugging Face: let's discuss it now.

Quantizing LLMs Using the bitsandbytes Library (bnb)

The Hugging Face bitsandbytes library is designed to make it easier to train and run large models on GPUs with limited VRAM. For this, it offers:

- Quantization tools, including 4-bit quantization, block-wise quantization, and more.
- Memory-efficient versions of popular optimizers such as Adam or AdamW, that operate on 8-bit tensors.
- Custom CUDA kernels written specifically for 8-bit or 4-bit quantized models, for maximum speed.

WARNING

The bitsandbytes library is designed for Nvidia GPUs. It also has some limited CPU support.

For example, let's see how to implement post-training static quantization down to 4 bits. If you are using Colab, you must first install the bitsandbytes library using `%pip install bitsandbytes`, then run this code:

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

model_id = "TinyLlama/TinyLlama-1.1B-Chat-v1.0"
bnb_config = BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4",
                                bnb_4bit_compute_dtype=torch.bfloat16)
model = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto",
                                             quantization_config=bnb_config)
```

This code starts by importing the necessary classes from the Transformers library (introduced in Chapter 14), then it creates a `BitsAndBytesConfig` object, which I will explain shortly. Lastly, it downloads a pretrained model (in this case a 1.1 billion parameter version of Llama named TinyLlama, fine-tuned for chat), specifying the desired quantization configuration.

Under the hood, the Transformers library uses the bitsandbytes library to quantize the model weights down to 4 bits just as they are loaded into the GPU: no extra step is required. You can now use this model normally to generate text (see Chapter 15). During inference, whenever some weights are needed, they are dequantized on the fly to the type specified by the `bnb_4bit_compute_dtype` argument (`bfloat16` in this case), and the computations are performed in this higher precision. As soon as the dequantized weights are no longer needed, they are dropped, so memory usage remains low.

In this example, the `BitsAndBytesConfig` object specifies *4-bit Normal Float* (NF4) quantization using `bfloat16` for computations. NF4 is a non-linear 4-bit scheme where each of the 16

possible integer values represents a specific float value between –1 and +1. Instead of being equally spaced (as in linear quantization), these values correspond to the quantiles of the Normal distribution centered on zero: this means that they are closer together near zero. This improves accuracy because model weights tend to follow a Normal distribution centered on zero, so having more precision near zero is helpful.

NF4 was introduced as part of QLoRA,[8] a technique that quantizes a frozen pretrained model with NF4, then uses LoRA adapters (see Chapter 17) for fine-tuning, along with gradient checkpointing (see Chapter 12). This approach drastically reduces VRAM usage and compute: the authors managed to fine-tune a 65-billion parameter model using a single GPU with 48 GB of RAM, with only a small accuracy drop. Although gradient checkpointing reduces VRAM usage overall, it can lead to memory spikes when processing batches with long sequences. To deal with such spikes, the QLoRA authors also introduced *paged optimizers* which take advantage of Nvidia unified memory: the CUDA driver automatically moves pages of data from GPU VRAM to CPU RAM whenever needed. Lastly, the authors also used *double quantization*, meaning that the quantization parameters themselves were quantized to save a bit more VRAM.

For more details on 4-bit quantization in the Hugging Face ecosystem, check out this great post by the QLoRA authors and other contributors.

Using Pre-Quantized Models

Many popular pretrained models have already been quantized and published online, in particular on the Hugging Face Hub. For example, Tom Jobbins, better known by his Hugging Face username TheBloke, has published thousands of quantized models available at *https://huggingface.co/TheBloke*. Many of these models were quantized using one of the following modern methods:

Generative pre-training quantization (GPTQ)

GPTQ[9] is a 4-bit post-training quantization method that treats quantization as an optimization problem. GPTQ goes through each layer, one by one, and optimizes the 4-bit weights to minimize the MSE between the layer's original outputs (i.e., using the full precision weights) and the approximate outputs (i.e., using the 4-bit weights). Once the optimal 4-bit weights are found, the approximate outputs are passed to the next layer, and the process is repeated all the way to the output layer. During inference, the weights are dequantized whenever they are needed. GPTQ only quantizes the weights, not the activations: this is called *weight-only quantization*, which is great for inference, not for training. You can use the Hugging Face Optimum library or the GPTQModel library to quantize your models with GPTQ.

Activation-aware Weight Quantization (AWQ)

AWQ[10] aims to improve the accuracy of block-wise weight-only quantization (typically 4-bit quantization). The idea is to preserve the precision of the most important weights. To identify these so-called *salient weights*, the algorithm runs a calibration dataset through

the model and finds the largest activations for each quantization group (e.g., the largest 0.1% to 1% activations), and the corresponding weights are considered salient. The authors observed that storing the salient weights using float16 greatly reduces the model's *perplexity* (a common metric equal to the exponential of the cross-entropy). However, mixing 4-bit and 16-bit weights is not hardware-friendly, so AWQ uses another method to preserve the salient weight's precision: they simply scale them up by some factor and add an operation in the model to scale down the corresponding activations (but this operation can generally be fused into the previous operation). Rather than using a fixed scaling factor, AWQ performs a search for the optimal factor, leading to the lowest quantization error. To implement AWQ, you can use the Hugging Face Optimum library.

Llama.cpp quantization using the GPT-Generated Unified Format (GGUF)

GGUF is a binary file format designed to store LLMs efficiently. It was introduced by Georgi Gerganov, the creator of llama.cpp, and it supersedes previous file formats such as GGML, GGMF and GGJT. A GGUF file includes the weights, the tokenizer, special tokens, the model architecture, the vocabulary size, and other metadata. Llama.cpp offers quantizers (e.g., using the `quantize` tool) to convert the model weights to one of GGUF's supported quantized formats, such as Q4_K_M. Q4 stands for 4-bit quantization, K stands for per-block quantization (typically 32 or 64 weights per block depending on the chosen format), and M means medium size and precision for this quantization level (other options are S = Small and L = Large). There are also more recent and efficient quantization options such as Importance-aware Quantization (IQ) which uses various techniques to improve accuracy (e.g., nonlinear quantization), and Ternary Quantization (TQ).

NOTE

On the Hugging Face Hub, every repository is backed by Git, so it has branches and commits. When you call `from_pretrained()`, the model is fetched from the default branch, which is almost always `main`. But quantized models are often placed in a different branch. When calling `from_pretrained()`, you can choose a branch, a tag, or even a commit hash, by using the `revision` argument. Check the model card for the list of available files and versions. For GGUF models, you must specify the file name using the `gguf_file` argument.

In conclusion, reduced precision, mixed-precision training, and quantization are arguably the most important tools to allow large models to run on limited hardware. But there are many more, including the following:

- You could tweak the model's architecture before training, by reducing the number of layers, or the number of neurons per layer, or by sharing weights across layers (e.g., as in the ALBERT model, introduced in Chapter 15).
- If you have a large trained model, you can shrink it by removing some of its weights, for example the ones with the smallest magnitude, or the ones with the smallest effect on the loss. You can also remove whole channels, layers, or attention heads. This is called *model pruning*, and you can implement it using the `torch.nn.utils.prune` module, or the Hugging Face Optimum library.

- As we saw in Chapter 15, you can also use a large trained model as a teacher to train a smaller model: this is called distillation.
- A trained model can also be shrunk by fusing some of its layers, removing redundancy. For example, a batch-norm layer (introduced in Chapter 11) performs a linear operation, so if it comes immediately after a linear layer, you can fuse both layers into a single linear layer. Similarly, you can fuse a convolutional layer followed by a batch-norm layer into a single convolutional layer. This only works after training, since the batch-norm layer must compute running averages during training. You can implement layer fusion with the `torch.quantization.fuse_modules()` function, or with the Hugging Face Optimum library. In any case, make sure to fuse layers *before* quantizing your model: less layers means less quantization noise.
- You can use low-rank approximations, where a large matrix is replaced by the product of two smaller ones. For example, replace a large linear layer such as `Linear(10_000, 20_000)` with two linear layers `Linear(10_000, 100)` and `Linear(100, 20_000)`. This reduces the number of parameters from about 200 million down to just three million, and also drastically reduces computations. The intermediate dimensionality (100 in this example) is a hyperparameter you can tune to balance accuracy and model size. This technique can be performed after training by factorizing the weight matrix using SVD (see the notebook for an example).

Give these techniques a try: shrink the models!

1 In general, –0 and +0 are considered equal, but some operations give different results, for example 1 / –0 = –infinity, while 1 / +0 = +infinity.

2 Some high-performance computing applications deactivate subnormal numbers because they slow down computations, and normalized numbers are generally sufficient (e.g., normalized fp32 can represent numbers as small as $\pm 1.2e^{-38}$).

3 The *M* stands for *mantissa*, which is a term often used as a synonym for fraction. Unfortunately, it's also used as a synonym for significand, leading to some confusion. This is why IEEE 754 no longer uses the term mantissa.

4 P. Micikevicius et al., "Mixed Precision Training", arXiv preprint 2017, ICLR (2018).

5 At the time of writing, `GradScaler` only works on the CPU and on Nvidia CUDA devices, but it should eventually work on other accelerators as well.

6 PyTorch implements *restricted symmetric quantization*, meaning that it excludes the lowest possible signed integer (e.g., –128 for 8-bit integers) to ensure that the range is symmetric (e.g., –127 to +127). Some other implementations allow the full signed byte range (from –128 to +127): this is called *unrestricted symmetric quantization*. These implementations also subtract 0.5 instead of 1 in the denominator of Equation B-2.

7 Benoit Jacob et al., "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", arXiv preprint arXiv:1712.05877 (2017)."

[8] Tim Dettmers et al., “QLORA: Efficient Finetuning of Quantized LLMs”, arXiv preprint arXiv:2305.14314 (2023).

[9] Elias Frantar et al., “GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers”, arXiv preprint arXiv:2210.17323 (2022).

[10] Ji Lin et al., “AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration”, arXiv preprint arXiv:2306.00978 (2023).