DL Demystified 13 - Model Compression - Quantization, LoRA & QLoRA

January 29, 2025

[1]:
```
##################################################################
#                                                                #
#  CS435 Generative AI: Security, Ethics and Governance          #
#                                                                #
#  Instructor: Dr. Adnan Masood                                  #
#  Contact:    adnanmasood@gmail.com                             #
#                                                                #
#  Notebook is MIT Licensed                                      #
##################################################################
```

1 Quantization, LoRA, and QLoRA Explained in a Jupyter Notebook

Welcome! In this notebook, we'll explore **Quantization**, **LoRA**, and **QLoRA** in detail. We will begin with an intuitive, simple explanation, and progressively dive deeper into the technical and mathematical aspects. This notebook is structured to provide a multi-level explanation: from a very simple level to a very advanced level. We'll go step-by-step with examples and code. By the end of this notebook, you'll have a thorough understanding of how these techniques work, why they matter, and how to apply them.

2 Table of Contents

14. Glossary

# 1. Building an Intuitive Understanding Below is a progression of explanations for **Quantization, LoRA, and QLoRA**, each adding more detail but focusing on the same core ideas:

Quantization: Imagine you have a big box of colored pencils with many shades. Quantization means you're picking fewer shades to color your picture—this makes your box smaller and easier to carry, but you can still draw almost the same picture.

LoRA: Think of it like having a big puzzle, but you only add a small set of new pieces to update your puzzle's picture. This is much easier than remaking the whole puzzle from scratch.

QLoRA: Combine the idea of picking fewer shades (Quantization) **and** adding small puzzle pieces (LoRA) together to make learning and using a big AI model faster and cheaper.

When dealing with neural networks, we store numbers (weights) in computers. Normally, we use full 32-bit numbers (floats). Quantization shrinks them to fewer bits (like 8-bit or 4-bit) to use less memory and compute faster.

LoRA (Low-Rank Adaptation) is a method to update a large model with fewer parameters. Instead of changing all the model's weights, we factorize them into smaller matrices (low-rank) and only update those parts, saving memory and training time.

QLoRA is a technique that uses 4-bit quantization for a large model while applying LoRA for fine-tuning. It drastically cuts down memory usage and speeds up training and inference.

Neural networks rely on large weight matrices. Using high-precision floating-point data can be overkill. Quantization maps these continuous values to discrete sets of integers in fewer bits, e.g., using 8-bit or 4-bit integers instead of 32-bit floats. This requires some scaling and offset for each weight.

LoRA decouples the main model weights from the update process. Instead of fine-tuning every single weight, we introduce low-dimensional matrices (A and B). We only learn these matrices during training, which drastically reduces the number of trainable parameters.

QLoRA combines 4-bit quantization (to store and run the large base model more efficiently) with LoRA to fine-tune it on new data. It offers the benefit of minimal hardware requirements alongside parameter-efficient fine-tuning.

We can view quantization as approximating the weight vector space with a smaller set of possible values. For instance, 4-bit quantization defines 2^4 = 16 possible levels. Each weight is mapped to a scaled integer representation, e.g., $w_{quant} = \text{round}((w - \alpha)/\delta)$ where α is an offset and δ is the step size.

If a weight matrix W is dimension $d \times d$, LoRA factorizes the update into A and B, each with dimensions $d \times r$ and $r \times d$, leading to a low-rank update: $\Delta W = AB^T$. We only train A and B, keeping the base W fixed, thereby drastically reducing the parameter count.

With QLoRA, the base weights are quantized to 4 bits (using specific algorithms designed for large language models). The LoRA low-rank matrices remain in higher precision. This effectively combines memory savings of quantization with the parameter efficiency of LoRA.

In large-scale LLMs, advanced quantization strategies might be used, such as per-channel scaling or block-wise quantization. There's ongoing research on the representational capacity of low-rank updates, how the rank r determines the model's flexibility, and how quantization error interacts with low-rank decomposition. QLoRA enables fine-tuning multi-billion-parameter models on GPUs with limited memory.

# 2. Intuition Behind Quantization, LoRA, and QLoRA **Intuition**: - **Quantization**: We reduce the precision of numbers. Think of it like rounding prices to the nearest dollar instead of using pennies. You lose some fine detail, but you speed up calculations and reduce storage. - **LoRA**: Instead of rewriting the entire book (model weights), you add notes in the margins that reference which parts of the book need updating. This is more efficient. - **QLoRA**: It's like having a short summary (4-bit quantization) of the big book plus those margin notes (LoRA) combined. You can store everything in a smaller space and still adapt it for new tasks.

# 3. Brief History and Underlying Tech - **Quantization**: Has been around in digital signal processing for decades. Became popular in neural networks to speed up inference on edge devices and reduce memory. - **LoRA**: Introduced as a parameter-efficient fine-tuning technique; it's a response to the exploding size of large language models (LLMs). The original paper demonstrated significant savings in GPU memory and training speed. - **QLoRA**: Proposed to leverage both 4-bit quantization for base LLM weights and LoRA for fine-tuning. Helps reduce the massive cost of hardware (GPU) and memory usage, enabling LLM tuning on more modest systems.

# 4. Math Behind the Techniques ### 4.1 Quantization - **Simple equation**: If w is a real-valued weight,

$$w_q = \text{round}\left(\frac{w - \alpha}{\delta}\right),$$

where w_q is the quantized integer, α is a zero point (offset), and δ is the scale. The real value is approximated by:

$$w_{approx} = w_q \times \delta + \alpha.$$

2.0.1 4.2 LoRA

- A weight matrix W is large. We keep it **frozen** and add a small update ΔW that can be written as AB^T:

  $$W_{\text{new}} = W + AB^T,$$

  where A and B are much smaller, rank-r matrices, typically with dimension $d \times r$ and $r \times d$, respectively.

2.0.2 4.3 QLoRA

- **4-bit Quantization of the base weights**: In practice, a technique like GPTQ or bitsandbytes is used.
- **LoRA applied on top**: The original weights are stored in 4-bit precision, while the LoRA matrices remain in higher precision. We can train just the LoRA part, combining memory efficiency with effective fine-tuning.

# 5. Illustrative Example with Code In this section, we'll do a very simplified demonstration of how quantization might work on a small tensor, then how LoRA can be conceptually applied in PyTorch.

[2]:
```python
# Let's do a toy example of quantization in Python
import torch
import numpy as np
```

```python
def simple_quantize(tensor, num_bits=4):
    # For demonstration, we do a naive min-max scaling
    # Determine min and max
    t_min = tensor.min()
    t_max = tensor.max()

    # Number of levels
    levels = 2 ** num_bits

    # Scale and zero-point
    scale = (t_max - t_min) / (levels - 1)
    zero_point = t_min

    # Quantize
    quantized = torch.round((tensor - zero_point) / scale)

    return quantized, scale, zero_point

def simple_dequantize(quantized, scale, zero_point):
    return quantized * scale + zero_point

# Example tensor
data = torch.tensor([0.1, 0.2, 0.25, 0.75, 1.0, 1.1], dtype=torch.float32)
print("Original Data:", data)

q_data, scale, zp = simple_quantize(data, num_bits=4)
dq_data = simple_dequantize(q_data, scale, zp)

print("\nQuantized Data (4-bit):", q_data)
print("Scale:", scale, "Zero Point:", zp)
print("Dequantized Data:", dq_data)
```

```
Original Data: tensor([0.1000, 0.2000, 0.2500, 0.7500, 1.0000, 1.1000])

Quantized Data (4-bit): tensor([ 0.,  2.,  2., 10., 13., 15.])
Scale: tensor(0.0667) Zero Point: tensor(0.1000)
Dequantized Data: tensor([0.1000, 0.2333, 0.2333, 0.7667, 0.9667, 1.1000])
```

In the code above, we: 1. **Compute** the range of the data (t_{min} and t_{max}). 2. **Derive** the scale and zero-point for a given bit precision (e.g., 4 bits gives 16 levels). 3. **Round** the data to these discrete levels. 4. **Dequantize** by reversing the process.

This is a simplistic demonstration of what quantization does.

2.0.3 LoRA Conceptual Example

Let's do a toy example of applying a low-rank update to a matrix in PyTorch. Note: This is a contrived example just to illustrate the idea of $W + AB^T$.

```
[3]: import torch

d = 6  # dimension
r = 2  # rank

# Original weight matrix W
W = torch.randn(d, d)

# LoRA matrices
A = torch.randn(d, r)
B = torch.randn(r, d)

# Low-rank update Delta W = A.mm(B)
Delta_W = A.mm(B)

# New weight
W_new = W + Delta_W

print("Original W:\n", W)
print("\nLoRA update Delta_W = A*B:\n", Delta_W)
print("\nW_new = W + Delta_W:\n", W_new)
```

```
Original W:
 tensor([[-1.6709,  0.1817, -0.9945, -0.0473, -0.1308, -0.8295],
        [ 0.3378, -0.5218,  0.5219,  0.4855, -0.3505,  0.7716],
        [ 0.5225,  0.9816,  0.5470,  1.4447, -0.7710, -0.5612],
        [ 0.0310,  1.7092,  0.3299,  0.2574,  0.5099, -1.2652],
        [-1.0065, -0.6935, -0.6856,  0.9836,  1.0478,  1.3924],
        [-0.0584, -2.2209,  0.7246, -0.8486, -0.7438,  0.7352]])

LoRA update Delta_W = A*B:
 tensor([[ 4.5739e-01, -2.7724e-01, -3.3288e-03,  3.6313e-01,  2.5559e-01,
          1.1802e+00],
        [ 6.7996e-01, -5.5937e-01,  3.8879e-02,  2.6342e-01,  4.8200e-01,
          1.8959e+00],
        [-6.5047e-02, -2.4404e-01,  8.4853e-02, -5.8380e-01,  1.6010e-01,
          1.0445e-01],
        [-1.0409e+00,  8.5025e-01, -5.7706e-02, -4.1470e-01, -7.3367e-01,
         -2.8966e+00],
        [ 8.0745e-01, -5.1029e-01,  3.3829e-04,  6.0184e-01,  4.6567e-01,
          2.1035e+00],
        [-2.6922e-01,  2.0129e-01, -9.3858e-03, -1.4218e-01, -1.7685e-01,
         -7.3128e-01]])

W_new = W + Delta_W:
 tensor([[-1.2135e+00, -9.5578e-02, -9.9783e-01,  3.1582e-01,  1.2483e-01,
          3.5071e-01],
        [ 1.0177e+00, -1.0812e+00,  5.6078e-01,  7.4888e-01,  1.3149e-01,
          2.6676e+00],
        [ 4.5747e-01,  7.3757e-01,  6.3185e-01,  8.6094e-01, -6.1094e-01,
         -4.5671e-01],
        [-1.0100e+00,  2.5594e+00,  2.7220e-01, -1.5730e-01, -2.2378e-01,
         -4.1618e+00],
        [-1.9910e-01, -1.2038e+00, -6.8526e-01,  1.5855e+00,  1.5134e+00,
          3.4959e+00],
        [-3.2765e-01, -2.0196e+00,  7.1524e-01, -9.9074e-01, -9.2063e-01,
          3.9345e-03]])
```

Here, we are simulating the **LoRA** technique. In an actual training scenario, W (the base model weights) would be frozen, and only A and B would be updated during backpropagation. Then the updated weight would be $W + AB^T$.

# 6. Example Calculations Let's do a small conceptual calculation: - Suppose you have a single weight $w = 0.78$ in float32. - You decide to quantize to 2 bits (4 levels: $\{0, 1, 2, 3\}$). - If your range is [0, 1], then scale $\delta = (1 - 0)/3 = 0.3333$, zero-point $\alpha = 0$. - Quantized value $w_q = \text{round}((0.78 - 0)/0.3333) = \text{round}(2.34) = 2$. - Dequantized $w_{approx} = 2 \times 0.3333 = 0.6667$.

Here, you see that you lost some precision (went from 0.78 to ~0.67), but you gained efficiency.

2.0.4 Terms:

- **Weight**: The parameter in the neural network.
- **Bias**: An additional parameter added to the weighted sum before passing through an activation.
- **Rank**: The dimension of the subspace for the LoRA update.
- **Scale & Zero Point**: Used in quantization to map continuous ranges to discrete integer levels.

# 7. Step-by-Step Example from Scratch We'll build a **tiny** neural network and do the following: 1. **Initialize** the network. 2. **Quantize** its parameters. 3. **Apply** a LoRA-like update.

Note: This is a conceptual demonstration and not a full-blown training session.

[4]:
```python
import torch
import torch.nn as nn

class TinyNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(TinyNet, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

```python
# Step 1: Initialize network
model = TinyNet(input_dim=4, hidden_dim=4, output_dim=2)
print("Original model parameters (fc1 weight):\n", model.fc1.weight)

# Step 2: Quantize the fc1 weight
fc1_weight = model.fc1.weight.data
q_fc1_weight, scale, zp = simple_quantize(fc1_weight, num_bits=4)
model.fc1.weight.data = simple_dequantize(q_fc1_weight, scale, zp)

print("\nQuantized + Dequantized model parameters (fc1 weight):\n", model.fc1.
  ↪weight)

# Step 3: Apply a LoRA-like update
r = 2
A = torch.randn(model.fc1.weight.data.shape[0], r)
B = torch.randn(r, model.fc1.weight.data.shape[1])
Delta = A.mm(B)
model.fc1.weight.data += Delta
print("\nAfter LoRA-like update (fc1 weight):\n", model.fc1.weight)
```

```
Original model parameters (fc1 weight):
 Parameter containing:
tensor([[-0.4468, -0.0502,  0.4679,  0.1325],
        [-0.4030,  0.0108, -0.3204,  0.0597],
        [ 0.1513,  0.4134,  0.4890, -0.0888],
        [ 0.4202,  0.0199,  0.3172,  0.4601]], requires_grad=True)

Quantized + Dequantized model parameters (fc1 weight):
 Parameter containing:
tensor([[-0.4468, -0.0725,  0.4890,  0.1147],
        [-0.3844, -0.0101, -0.3220,  0.0523],
        [ 0.1771,  0.4266,  0.4890, -0.0725],
        [ 0.4266, -0.0101,  0.3018,  0.4890]], requires_grad=True)

After LoRA-like update (fc1 weight):
 Parameter containing:
tensor([[ 4.7902,  4.0630, -2.1885,  1.8887],
        [ 1.7763,  1.6905, -1.3997,  0.8723],
        [-1.5059, -0.9240,  1.4538, -0.3012],
        [-2.0380, -2.0002,  1.7736,  0.3468]], requires_grad=True)
```

We first **initialized** a tiny network. Then, we **quantized** its `fc1` weight parameter to 4 bits. Next, we **dequantized** it back and replaced the parameter in the model, simulating storing it in quantized form. Finally, we added a LoRA-like low-rank update to illustrate how we can adapt a quantized model.

# 8. Illustrative Problem It Solves For instance, if you want to run a **Large Language Model** on a **resource-limited device** (like a smaller GPU or edge device), you can't afford storing massive

32-bit floats for billions of parameters. By using 4-bit quantization, you drastically reduce memory.

But you also want to **adapt** or fine-tune the model for a new language or domain. Full fine-tuning of billions of parameters is also expensive. Instead, you apply **LoRA** (only a small set of trainable parameters), and combine that with the quantized model. This is **QLoRA**, which solves the problem of running and updating large models cheaply.

# 9. Real-World Problems Solved 1. **Domain adaptation**: Fine-tune a large model on a specialized domain (e.g., medical data) using LoRA, while storing the base model in 4-bit. 2. **Edge deployment**: Deploy on embedded devices or small data centers with limited GPU memory. 3. **Cost reduction**: Lower the hardware requirement means cheaper training and inference.

# 10. How to Solve a Real-World Problem Using This Tech 1. **Pick a big pretrained model** (like a 7B or 13B parameter language model). 2. **Quantize** the model using a 4-bit or 8-bit quantization tool. 3. **Apply LoRA**: Freeze the base weights and add low-rank adapters. 4. **Train** the LoRA parameters with a small learning rate on your domain data. 5. **Deploy**: Use the quantized model + LoRA parameters in production. You only need to load the small LoRA overhead in full precision. Everything else is compressed.

# 11. What Other Questions Can You Ask? (Points to Ponder) 1. *How do we choose the right number of bits for quantization?* 2. *How do we select the rank r for LoRA?* 3. *When does quantization error significantly degrade performance?* 4. *How does outlier-aware quantization improve results?* 5. *How do we handle gradient updates in QLoRA for the quantized weights?*

# 12. Answers to the Questions (with Code Examples) ### 1. How do we choose the right number of bits for quantization? - Typically, we try 8-bit or 4-bit because they balance memory savings and accuracy. 2-bit might be too aggressive. The best way is to **experiment** on a validation set.

2.0.5 2. How do we select the rank r for LoRA?

- This depends on how complex the new task is. A higher rank means more parameters to learn, but better capacity. A typical range is $r \in [4, 64]$ for LLM fine-tuning.

2.0.6 3. When does quantization error significantly degrade performance?

- If your network is very sensitive to small changes in weights or if you quantize to too few bits (2-bit or 3-bit). Large networks tend to be somewhat robust to quantization, but tasks requiring very high precision might suffer.

2.0.7 4. How does outlier-aware quantization improve results?

- Some weights (outliers) can be very large, messing up min-max scaling. Outlier-aware quantization uses separate scaling for outliers or specialized algorithms to reduce the impact of these extreme values.

2.0.8 5. How do we handle gradient updates in QLoRA for the quantized weights?

- In practice, the gradients are computed in higher precision. We don't update the quantized weights directly. Instead, we update the LoRA parameters. The quantized weights remain mostly unchanged.

# 13. A Sample Exercise Below is a **template** code you can run and fill in the TODO items. It demonstrates building a small network, quantizing it, and applying a LoRA update. Complete the TODOs to see how it works.

[6]:
```
# TODO: Complete and run this code.
import torch
import torch.nn as nn

def my_quantize(tensor, num_bits=4):
    # TODO: Implement a simple min-max quantization
    # 1. Find the min and max of the tensor
    # 2. Compute scale = (max - min) / (levels - 1)
    # 3. Compute zero_point = min
    # 4. Quantize by rounding
    tensor_min = tensor.min()
    tensor_max = tensor.max()
    scale = (tensor_max - tensor_min) / (2 ** num_bits - 1)
    zero_point = tensor_min
    quantized_tensor = torch.round((tensor - zero_point) / scale)
    return quantized_tensor, scale, zero_point

def my_dequantize(quantized_tensor, scale, zero_point):
    # TODO: Implement the reverse process
    return quantized_tensor * scale + zero_point

# Define a simple linear layer
linear = nn.Linear(3, 2, bias=False)

# Print original parameters
print("Original params:", linear.weight)

# TODO: Call my_quantize on linear.weight.data
q_weight, scale, zp = my_quantize(linear.weight.data, num_bits=4)

# TODO: Dequantize
dq_weight = my_dequantize(q_weight, scale, zp)

# TODO: Assign dq_weight back to linear.weight.data
linear.weight.data = dq_weight

# Create LoRA-like update
r = 1
A = torch.randn(linear.weight.size(0), r)
B = torch.randn(r, linear.weight.size(1))
Delta = A.mm(B)

# TODO: Add Delta to linear.weight.data
```

```
linear.weight.data += Delta

# Print updated parameters
print("\nUpdated params after LoRA-like update (complete the TODOs first):",␣
↪linear.weight)
```

```
Original params: Parameter containing:
tensor([[-0.5225, -0.0100,  0.1037],
        [ 0.1947, -0.1677, -0.2551]], requires_grad=True)

Updated params after LoRA-like update (complete the TODOs first): Parameter
containing:
tensor([[-0.1693,  1.4029, -1.0591],
        [ 0.3295,  0.3458, -0.6773]], requires_grad=True)
```

By filling in the TODOs, you'll practice implementing your own min-max quantization logic, dequantization, and finally adding a LoRA-like update.

# 14. Glossary - **Quantization**: Reducing the number of bits used to represent numbers in neural networks. - **LoRA (Low-Rank Adaptation)**: A method that updates only a small set of parameters through low-rank factorization. - **QLoRA**: 4-bit quantization + LoRA to fine-tune large models efficiently. - **Scale**: Factor used to scale floating-point range into a smaller discrete range. - **Zero Point (Offset)**: A shift added so that the minimum value can map to 0 in quantized integer space. - **Rank**: The number of columns in matrix A (and rows in matrix B) for LoRA. - **Parameter-Efficient Fine-Tuning**: Methods that avoid updating all model parameters. - **Outlier-Aware Quantization**: Specialized approach for dealing with extremely large or small weight values. - **Min-Max Quantization**: The simplest form of quantization using the min and max of the data. - **bitsandbytes**: A popular library for 8-bit and 4-bit optimization and quantization.

2.1 Conclusion

With these examples and explanations, you should now have a comprehensive understanding of **Quantization**, **LoRA**, and **QLoRA**, along with their theory, mathematics, and practical implementation details. Happy exploring and coding!

[8]:
```
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in␣
↪reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-29 01:41:51.289597
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```

[ ]:

```
```