Inference Optimization Through Quantization Strategies

We'll take a look at how effective post-training quantization strategies can be when it comes to decreasing inference latency.

```
!pip install -qU transformers==4.46.3 autoawq==0.2.5 accelerate openai huggingface-hub peft==0.14.0 torch==2.4.1 torchvision
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.1/44.1 kB 3.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.0/10.0 MB 107.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.3/84.3 kB 7.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 797.1/797.1 MB 1.4 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 410.6/410.6 MB 2.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 14.1/14.1 MB 101.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 23.7/23.7 MB 74.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 823.6/823.6 kB 50.4 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 1.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 121.6/121.6 MB 17.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.5/56.5 MB 38.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 124.2/124.2 MB 18.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 196.0/196.0 MB 6.0 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 176.2/176.2 MB 12.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 99.1/99.1 kB 8.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 209.4/209.4 MB 3.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 345.1/345.1 kB 28.0 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 567.4/567.4 kB 37.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 469.0/469.0 kB 37.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.0/7.0 MB 106.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.0/3.0 MB 97.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 37.3/37.3 MB 57.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 485.4/485.4 kB 38.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 11.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 143.5/143.5 kB 13.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.8/194.8 kB 17.8 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is
torchaudio 2.5.1+cu124 requires torch==2.5.1, but you have torch 2.4.1 which is incompatible.
```

```
import os
from getpass import getpass

os.environ["HF_TOKEN"] = getpass("HF_TOKEN")
```

```
HF_TOKEN··········
```

```
from huggingface_hub import notebook_login

notebook_login()
```

```
Note: Environment variable`HF_TOKEN` is set and is the current active token independently from the token you've just configu
```

Simple Benchmark: `BitsAndBytes` vs. GPTQ vs. AWQ

We've learned, at this point, about BitsAndBytes and how much memory footprint we can save using Tim Dettmer's libraries. However, we've always had to include a small caveat:

> BitsAndBytes does not improve inference performance, in fact, there is a small inference penalty for using it.

Enter: GPTQ & AWQ!

By taking advantage of a number of innovative techniques, GPTQ & AWQ are able to provide inference-time benefits *without* sacrificing as much accuracy/etc. as BitsAndBytes.

There is one distinct *disadvantage* to GPTQ and AWQ, which is that they are post-training Quantization strategies - and so are not useful while fine-tuning/training models.

Let's take a look at the inference-time benefits by comparing three Hugging Face Inference Endpoints, each running the model in GPTQ/AWQ/BNBs respectively, to see how they perform.

> NOTE: The following cell will take a while to install - please move on to Activity #1 while you wait for the installation to complete.

```
!pip install -qU gptqmodel --no-build-isolation
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 280.5/280.5 kB 17.8 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.0/62.0 kB 5.4 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.0/44.0 kB 4.1 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
  Preparing metadata (setup.py) ... done
  Preparing metadata (setup.py) ... done
 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.6/3.6 MB 98.5 MB/s eta 0:00:00
 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 16.4/16.4 MB 107.8 MB/s eta 0:00:00
 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 316.2/316.2 kB 24.0 MB/s eta 0:00:00
 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.0/10.0 MB 113.2 MB/s eta 0:00:00
 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.0/3.0 MB 80.2 MB/s eta 0:00:00
  Building wheel for gptqmodel (setup.py) ... done
  Building wheel for device-smi (setup.py) ... done
  Building wheel for logbar (setup.py) ... done
  Building wheel for tokenicer (setup.py) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is
google-cloud-language 2.16.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but
pytensor 2.27.1 requires numpy<2,>=1.17.0, but you have numpy 2.2.3 which is incompatible.
google-cloud-bigtable 2.29.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but
proto-plus 1.26.0 requires protobuf<6.0.0dev,>=3.19.0, but you have protobuf 6.30.1 which is incompatible.
google-cloud-aiplatform 1.79.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, bu
google-cloud-bigquery-connection 1.18.1 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3
tensorflow-metadata 1.16.1 requires protobuf<6.0.0dev,>=4.25.2; python_version >= "3.11", but you have protobuf 6.30.1 which
googleapis-common-protos 1.69.0 requires protobuf!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0.dev0,>=3.20.2, but you
google-cloud-pubsub 2.25.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but yo
google-cloud-dataproc 5.18.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but
numba 0.60.0 requires numpy<2.1,>=1.22, but you have numpy 2.2.3 which is incompatible.
wandb 0.19.7 requires protobuf!=4.21.0,!=5.28.0,<6,>=3.19.0; python_version > "3.9" and sys_platform == "linux", but you hav
google-cloud-bigquery-storage 2.28.0 requires protobuf!=3.20.0,!=3.20.1,!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.
langchain 0.3.19 requires numpy<2,>=1.26.4; python_version < "3.12", but you have numpy 2.2.3 which is incompatible.
google-cloud-translate 3.19.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but
google-cloud-datastore 2.20.2 requires protobuf!=3.20.0,!=3.20.1,!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.
tensorflow 2.18.0 requires numpy<2.1.0,>=1.26.0, but you have numpy 2.2.3 which is incompatible.
tensorflow 2.18.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3, but you have pr
google-cloud-resource-manager 1.14.1 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20
google-cloud-functions 1.19.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but
google-cloud-firestore 2.20.1 requires protobuf!=3.20.0,!=3.20.1,!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.
google-ai-generativelanguage 0.6.15 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.
google-api-core 2.24.1 requires protobuf!=3.20.0,!=3.20.1,!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0.dev0,
gensim 4.3.3 requires numpy<2.0,>=1.18.5, but you have numpy 2.2.3 which is incompatible.
google-cloud-iam 2.18.1 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but you h
thinc 8.2.5 requires numpy<2.0.0,>=1.19.0; python_version >= "3.9", but you have numpy 2.2.3 which is incompatible.
google-cloud-spanner 3.52.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but y
grpc-google-iam-v1 0.14.1 requires protobuf!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.2, but you have pro
```

🏗️ Activity #1:

You must spin up a total of 3 inference endpoints:

1. Regular `meta-llama/Llama-3.1-8B-Instruct` (or NousResearch variant if you don't have access to Meta's weights)
2. Hugging Quants Meta-Llama-3.1-8B-Instruct-GPTQ-INT4 weights.
3. Hugging Quants Meta-Llama-3.1-8B-Instruct-AWQ-INT4 weights.

> NOTE: You can spin these up all at once, or serially, depending on your preference.

Big Wall of Test Strings

~100 test strings to do some generations with.

Wall of Strings

[ ] ↳ 1 cell hidden

Inference Test Helper Function

We'll create a function that tests a few key inference related metrics:

1. Time to First Token (TTFT): How long does it take before our endpoint starts returning tokens, TTFT is key in creating applications that *feel* responsive due to responses from LLMs typically being streamed to users as tokens are generated.
2. Inter-Token Latency (ITL): Inter-Token Latency talks about the amount of time between tokens being generated. Lower ITL helps the response come through fast enough to keep up with typical reading speeds.
3. Tokens Per Second (TPS): A classic metric that simply indicates how many Tokens are produced per second.

Let's create this function below - and then use it evaluate our AWQ endpoint and our BNB endpoint.

```
import os
import time
from openai import OpenAI
from typing import List, Dict
from tqdm import tqdm

def measure_endpoint_performance(endpoint_url: str, sentences: List[str]) -> Dict[str, float]:
    client = OpenAI(
        base_url=endpoint_url,
        api_key=os.environ["HF_TOKEN"]
    )

    total_time = 0
    total_first_token_time = 0
    total_tokens = 0
    total_inter_token_time = 0
    total_inter_token_intervals = 0

    for sentence in tqdm(sentences):
        start_time = time.time()
        first_token_received = False
        tokens_received = 0
        last_token_time = start_time

        chat_completion = client.chat.completions.create(
            model="tgi",
            messages=[
                {
                    "role": "user",
                    "content": sentence
                }
            ],
            stream=True,
            max_tokens=20
        )

        for message in chat_completion:
            current_time = time.time()
            if not first_token_received:
                first_token_time = current_time - start_time
                total_first_token_time += first_token_time
                first_token_received = True
            else:
                inter_token_time = current_time - last_token_time
                total_inter_token_time += inter_token_time
                total_inter_token_intervals += 1

            content = message.choices[0].delta.content
            if content:
                tokens_received += 1
                last_token_time = current_time

        request_time = time.time() - start_time
        total_time += request_time
        total_tokens += tokens_received

    num_sentences = len(sentences)
    average_time = total_time / num_sentences
    average_first_token_time = total_first_token_time / num_sentences
    average_tokens_per_second = total_tokens / total_time
    average_inter_token_latency = total_inter_token_time / total_inter_token_intervals if total_inter_token_intervals > 0 else 0

    return {
        "average_request_time": average_time,
        "average_time_to_first_token": average_first_token_time,
        "average_tokens_per_second": average_tokens_per_second,
        "average_inter_token_latency": average_inter_token_latency
    }
```

Bits and Bytes Endpoint Evaluation

First, let's baseline with [Llama 3.1 8B Instruct]() powered by BitsAndBytes quantization through TGI.

```
bitsandbytes_endpoint_url = "https://sraiyefl4yqjy8d5.us-east-1.aws.endpoints.huggingface.cloud" + "/v1/"
```

```
bnb_results = measure_endpoint_performance(bitsandbytes_endpoint_url, test_sentences)
```

```
100%|██████████| 103/103 [02:27<00:00,  1.43s/it]
```

```
print("\nResults:")
for key, value in bnb_results.items():
    print(f"{key}: {value:.2f}")
```

```
Results:
average_request_time: 1.43
average_time_to_first_token: 0.34
average_tokens_per_second: 13.58
average_inter_token_latency: 0.06
```

GPTQ Evaluation

We'll be using the [Hugging Quants Meta-Llama-3.1-8B-Instruct-GPTQ-INT4]() model as our GPTQ test model.

Given what we've learned about GPTQ - this endpoint should outperform our naive BitsAndBytes quantized endpoint.

GPTQ: Under the Hood

The basic outline of what's happening in GPTQ is as follows:

1. Start with a pre-trained language model
2. For each layer:
    - Compute an approximation of the layer's Hessian matrix
    - Quantize weights column-by-column
    - After each column, update remaining weights to compensate
3. Use special tricks to make this efficient:
    - Quantize in fixed order instead of greedy order
    - Process weights in batches
    - Use Cholesky decomposition for numerical stability
4. Result: Compressed model that can run much faster

Hessian Matrix:

Okay, so that makes sense - but there's a question: What is the layer's "Hessian matrix"?

In essence, we can think of the layer's "Hessian Matrix" as a map of how sensitive a layer's output is to changes in its weights. This gives us a matrix that corresponds to how much the output will change based on changes to each weight.

The process described in GPTQ uses a fast approximation to get the Hessian Matrix and then uses that to determine how to compress (or quantize) the model's weights such that they don't mess up the model's outputs.

So where a process like AWQ uses the activations to determine how each parameter (weight) impacts the outputs - GPTQ uses each layer's Hessian Matrix.

```
gptq_endpoint_url = "https://inepjrkz8uy61z55.us-east-1.aws.endpoints.huggingface.cloud" + "/v1/"
```

```
gptq_results = measure_endpoint_performance(gptq_endpoint_url, test_sentences)
```

```
100%|██████████| 103/103 [01:09<00:00,  1.49it/s]
```

```
print("\nResults:")
for key, value in gptq_results.items():
```

```
    print(f"{key}: {value:.2f}")
```

```
Results:
average_request_time: 0.67
average_time_to_first_token: 0.29
average_tokens_per_second: 27.79
average_inter_token_latency: 0.02
```

AWQ Evaluation

We'll be using the [Hugging Quants Meta-Llama-3.1-8B-Instruct-AWQ-INT4]() as our AWQ test model.

Given what we've learned about AWQ - this endpoint should outperform our naive BitsAndBytes quantized endpoint - let's test it out!

AWQ: Under the Hood

There is a key set of assumptions that AWQ is working off of:

1. Some weights are more important than other weights.
2. That proportion of important weights is extremely small (~=1%)
3. We are working with hardware optimzed for specific kinds of computations
4. Moving things around in GPU memory is slow and inefficient for most use-cases

```
awq_endpoint_url = "https://ecrtxcyhfobgl7q9.us-east-1.aws.endpoints.huggingface.cloud" + "/v1/"
```

```
awq_results = measure_endpoint_performance(awq_endpoint_url, test_sentences)
```

```
100%|██████████| 103/103 [01:08<00:00,  1.49it/s]
```

```
print("\nResults:")
for key, value in awq_results.items():
    print(f"{key}: {value:.2f}")
```

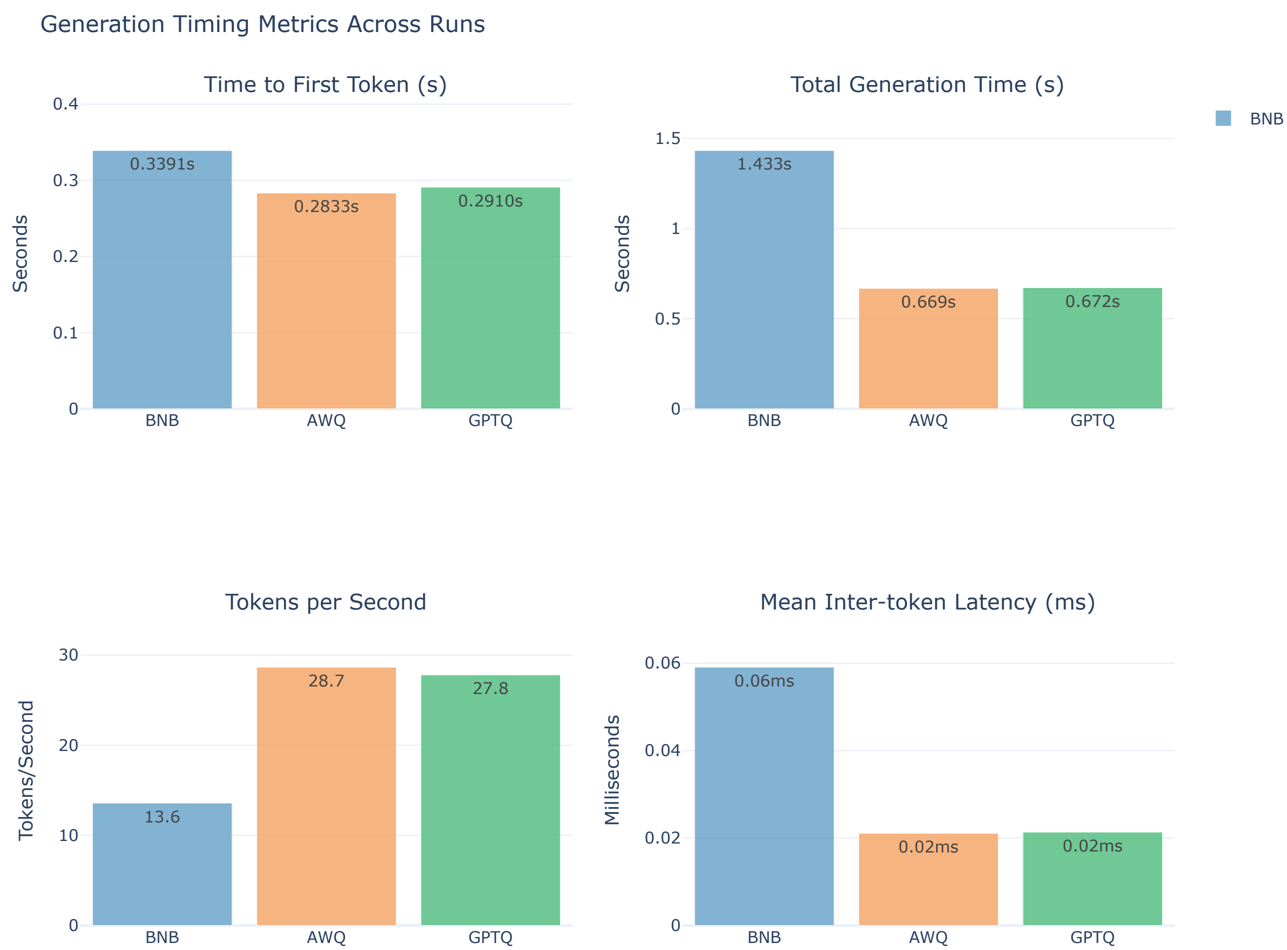
```
Results:
average_request_time: 0.67
average_time_to_first_token: 0.28
average_tokens_per_second: 28.65
average_inter_token_latency: 0.02
```

Graphing the Outputs

```
import plotly.graph_objects as go
import plotly.subplots as sp

# Define your model names
models = ["BNB", "AWQ", "GPTQ"]

# Create data arrays with your actual data
time_to_first_token = [
    bnb_results["average_time_to_first_token"],
    awq_results["average_time_to_first_token"],
    gptq_results["average_time_to_first_token"]
]

request_time = [
    bnb_results["average_request_time"],
    awq_results["average_request_time"],
    gptq_results["average_request_time"]
]

tokens_per_second = [
    bnb_results["average_tokens_per_second"],
    awq_results["average_tokens_per_second"],
    gptq_results["average_tokens_per_second"]
]

inter_token_latency = [
    bnb_results["average_inter_token_latency"],
```

```
        awq_results["average_inter_token_latency"],
        gptq_results["average_inter_token_latency"]
    ]

    # Colors for each model
    colors = ['rgba(100, 160, 200, 0.8)', 'rgba(244, 162, 97, 0.8)', 'rgba(76, 187, 123, 0.8)']

    # Create subplots
    fig = sp.make_subplots(
        rows=2, cols=2,
        subplot_titles=(
            "Time to First Token (s)",
            "Total Generation Time (s)",
            "Tokens per Second",
            "Mean Inter-token Latency (ms)"
        )
    )

    # Add Time to First Token bars
    for i, model in enumerate(models):
        fig.add_trace(
            go.Bar(
                x=[model],
                y=[time_to_first_token[i]],
                name=model,
                marker_color=colors[i],
                showlegend=i==0,
                text=[f"{time_to_first_token[i]:.4f}s"],
                textposition="auto"
            ),
            row=1, col=1
        )

    # Add Total Generation Time bars
    for i, model in enumerate(models):
        fig.add_trace(
            go.Bar(
                x=[model],
                y=[request_time[i]],
                name=model,
                marker_color=colors[i],
                showlegend=False,
                text=[f"{request_time[i]:.3f}s"],
                textposition="auto"
            ),
            row=1, col=2
        )

    # Add Tokens per Second bars
    for i, model in enumerate(models):
        fig.add_trace(
            go.Bar(
                x=[model],
                y=[tokens_per_second[i]],
                name=model,
                marker_color=colors[i],
                showlegend=False,
                text=[f"{tokens_per_second[i]:.1f}"],
                textposition="auto"
            ),
            row=2, col=1
        )

    # Add Inter-token Latency bars
    for i, model in enumerate(models):
        fig.add_trace(
            go.Bar(
                x=[model],
                y=[inter_token_latency[i]],
                name=model,
                marker_color=colors[i],
                showlegend=False,
                text=[f"{inter_token_latency[i]:.2f}ms"],
                textposition="auto"
            ),
            row=2, col=2
        )
```

```
# Update y-axes scales based on data ranges
y1_max = max(time_to_first_token) * 1.2  # 20% headroom
y2_max = max(request_time) * 1.2
y3_max = max(tokens_per_second) * 1.2
y4_max = max(inter_token_latency) * 1.2

fig.update_yaxes(title_text="Seconds", range=[0, y1_max], row=1, col=1)
fig.update_yaxes(title_text="Seconds", range=[0, y2_max], row=1, col=2)
fig.update_yaxes(title_text="Tokens/Second", range=[0, y3_max], row=2, col=1)
fig.update_yaxes(title_text="Milliseconds", range=[0, y4_max], row=2, col=2)

# Update layout
fig.update_layout(
    title_text="Generation Timing Metrics Across Runs",
    height=800,
    width=1000,
    template="plotly_white",
    bargap=0.15,
    barmode='group'
)

# Display the figure
fig.show()
```



❓ Question:

Describe the difference in performance profiles between the three solutions.

Answer

Performance Comparison: Bits and Bytes (BNB) vs. GPTQ vs. AWQ

1. Time to First Token (Lower is Better)

AWQ has the fastest response time, followed extremely closely by GPTQ, while BNB lags slightly behind.

2. Total Request Time (Lower is Better)

GPTQ and AWQ have significantly lower total request times (~2.1x faster than BNB).
Bits and Bytes is significantly slower overall.

3. Tokens per Second (Higher is Better)

AWQ and GPTQ are over twice as fast as BNB in token generation speed.

4. Mean Inter-Token Latency (Lower is Better)

GPTQ and AWQ have significantly lower inter-token latency, making them better for streaming responses.

Overall Performance Summary

Model	Time to First Token (s)	Request Time (s)	Tokens/sec	Inter-token Latency (s)
BNB	0.34	**1.43** ⏳ (Slowest)	**13.58** 🚨 (Lowest)	0.06 🚨 (Highest)
GPTQ	0.29	0.67 ✅	27.79 ✅	0.02 ✅
AWQ	**0.28** ✅ (Fastest)	**0.67** ✅	**28.65** ✅ (Fastest)	**0.02** ✅

Conclusion

- **AWQ is the best performer**, slightly ahead of GPTQ in most metrics.
- **GPTQ is nearly identical to AWQ**, with a very minor lag in token generation speed.
- **Bits and Bytes performs the worst**, being **2.1x slower in request time** and **2x slower in tokens per second** compared to GPTQ/AWQ.

Checkpoint Conversion (Optional)

For both the AWQ and GPTQ quantization strategies - there exist easy conversion strategies.

We'll go over both of those strategies below.

Using AutoAWQ to Convert a Model

We'll see in the following step how easy it is to convert a model to AWQ by leveraging AutoAWQ.

First, we'll need a model we wish to convert. We'll stick with Llama 3.1 8B Instruct for demonstration purposes.

We'll start by setting some parameters that will help us define the resultant model:

- `zero-point` - this indicates to use Zero-Point Quantization as our quantization strategy.
  - `q = round((x / scale) + zero_point)`, this quantization strategy allows us to better represent numbers asymetrically around zero, meaning we can use unsigned 8bit integers to describe both positvie and negative weights.
- `q_group_size` - like in BNB quantization, we quantize many weights under the same `scale` and `zero_point` to add additional efficiency during quanitzation without losing as much precision. These are sometimes called "bins" or "blocks".
- `w_bit` - the size of the weight in bits

```
model_path = "meta-llama/Meta-Llama-3.1-8B-Instruct"
quant_path = "hugging-quants/Meta-Llama-3.1-8B-Instruct-AWQ-INT4"

quant_config = {
  "zero_point": True,
  "q_group_size": 128,
  "w_bit": 4,
  "version": "GEMM",
}
```

```
!pip install numpy==1.19.5
```

```
Collecting numpy==1.19.5
  Using cached numpy-1.19.5.zip (7.3 MB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: numpy
  error: subprocess-exited-with-error

  × Building wheel for numpy (pyproject.toml) did not run successfully.
  │ exit code: 1
  ╰─> See above for output.

  note: This error originates from a subprocess, and is likely not a problem with pip.
  Building wheel for numpy (pyproject.toml) ... error
  ERROR: Failed building wheel for numpy
Failed to build numpy
ERROR: ERROR: Failed to build installable wheels for some pyproject.toml based projects (numpy)
```

```
!pip install numpy==1.19.5
from awq import AutoAWQForCausalLM
from transformers import AutoTokenizer

# Load model
model = AutoAWQForCausalLM.from_pretrained(
  model_path, low_cpu_mem_usage=True, use_cache=False, device_map="auto",
)
tokenizer = AutoTokenizer.from_pretrained(model_path)

# Quantize
model.quantize(tokenizer, quant_config=quant_config)

# Save quantized model
model.save_quantized(quant_path)
tokenizer.save_pretrained(quant_path)

print(f'Model is quantized and saved at "{quant_path}"')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/transformers/utils/import_utils.py in _get_module(self, module_name)
   1862         try:
-> 1863             return importlib.import_module("." + module_name, self.__name__)
   1864         except Exception as e:

                                  37 frames
AttributeError: module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in existing code, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and guidance see the original release note at:
    https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/transformers/utils/import_utils.py in _get_module(self, module_name)
   1863             return importlib.import_module("." + module_name, self.__name__)
   1864         except Exception as e:
-> 1865             raise RuntimeError(
   1866                 f"Failed to import {self.__name__}.{module_name} because of the following error (look up to see
its"
   1867                 f" traceback):\n{e}"

RuntimeError: Failed to import transformers.generation.utils because of the following error (look up to see its traceback):
module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in existing code, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and guidance see the original release note at:
    https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
```

Next steps: Explain error

Using GPTQModel to Convert a Model

We'll see in the following step how easy it is to convert a model to GPTQ by leveraging GPTQModel.

First, we'll need a model we wish to convert. We'll stick with Llama 3.1 8B Instruct for demonstration purposes.

We'll start by setting some parameters that will help us define the resultant model:

- `bits` - the target bit width for the final quanitzed model
- `group_size` - the number of weights to be quantized under a single scaling factor

```
from gptqmodel import GPTQModel, QuantizeConfig

pretrained_model_dir = "NousResearch/Meta-Llama-3.1-8B-Instruct"
quantized_model_dir = "Meta-Llama-3.1-8B-Instruct-4bit"

quant_config = QuantizeConfig(
    bits=4,
    group_size=128,
)
```

```
INFO  ENV: Auto setting PYTORCH_CUDA_ALLOC_CONF='expandable_segments:True' for memory saving.
INFO  ENV: Auto setting CUDA_DEVICE_ORDER=PCI_BUS_ID for correctness.
/usr/local/lib/python3.11/dist-packages/numpy/_core/_dtype.py:106: FutureWarning:

In the future `np.bool` will be defined as the corresponding NumPy scalar.

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/transformers/utils/import_utils.py in _get_module(self, module_name)
   1862         try:
-> 1863             return importlib.import_module("." + module_name, self.__name__)
   1864         except Exception as e:

                                 51 frames
AttributeError: module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in existing code, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and guidance see the original release note at:
    https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
RuntimeError: Failed to import transformers.generation.utils because of the following error (look up to see its traceback):
module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in existing code, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and guidance see the original release note at:
    https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/transformers/utils/import_utils.py in _get_module(self, module_name)
   1863             return importlib.import_module("." + module_name, self.__name__)
   1864         except Exception as e:
-> 1865             raise RuntimeError(
   1866                 f"Failed to import {self.__name__}.{module_name} because of the following error (look up to see
its"
   1867                 f" traceback):\n{e}"

RuntimeError: Failed to import transformers.models.auto.tokenization_auto because of the following error (look up to see
its traceback):
Failed to import transformers.generation.utils because of the following error (look up to see its traceback):
module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in existing code, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and guidance see the original release note at:
```

Next steps: Explain error

```
import logging

logging.basicConfig(
    format="%(asctime)s %(levelname)s [%(name)s] %(message)s", level=logging.INFO, datefmt="%Y-%m-%d %H:%M:%S"
)


from transformers import AutoTokenizer, TextGenerationPipeline

# Load model
model = GPTQModel.from_pretrained(pretrained_model_dir, quant_config)
```

```
tokenizer = AutoTokenizer.from_pretrained(pretrained_model_dir, use_fast=True)
calibration_dataset = [tokenizer(text) for text in test_sentences]

# Quantize
model.quantize(calibration_dataset)

# Save quantized model
model.save_quantized(quantized_model_dir)

print(f'Model is quantized and saved at "{quantized_model_dir}"')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/transformers/utils/import_utils.py in _get_module(self, module_name)
   1862         try:
-> 1863             return importlib.import_module("." + module_name, self.__name__)
   1864         except Exception as e:

46 frames
AttributeError: module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in existing code, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and guidance see the original release note at:
    https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
RuntimeError: Failed to import transformers.generation.utils because of the following error (look up to see its traceback):
module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in existing code, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and guidance see the original release note at:
    https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/transformers/utils/import_utils.py in _get_module(self, module_name)
   1863             return importlib.import_module("." + module_name, self.__name__)
   1864         except Exception as e:
-> 1865             raise RuntimeError(
   1866                 f"Failed to import {self.__name__}.{module_name} because of the following error (look up to see
its"
   1867                 f" traceback):\n{e}"

RuntimeError: Failed to import transformers.models.auto.tokenization_auto because of the following error (look up to see
its traceback):
Failed to import transformers.generation.utils because of the following error (look up to see its traceback):
module 'numpy' has no attribute 'bool'.
`np.bool` was a deprecated alias for the builtin `bool`. To avoid this error in existing code, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
The aliases was originally deprecated in NumPy 1.20; for more details and guidance see the original release note at:
```

Next steps: Explain error

```
model = GPTQModel.from_quantized(quantized_model_dir)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-25-74d6793cc7e8> in <cell line: 0>()
----> 1 model = GPTQModel.from_quantized(quantized_model_dir)

NameError: name 'GPTQModel' is not defined
```

Next steps: Explain error

```
print(tokenizer.decode(model.generate(**tokenizer("gptqmodel is", return_tensors="pt").to(model.device))[0]))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-26-7cea32300fde> in <cell line: 0>()
----> 1 print(tokenizer.decode(model.generate(**tokenizer("gptqmodel is", return_tensors="pt").to(model.device))[0]))

NameError: name 'tokenizer' is not defined
```

Next steps: Explain error