Team number: 27
Team members:
Siju Niyati Samji 23110312
Makkena Lakshmi Manasa 23110193

Google colab link: **STT_11.ipynb**
Github Repository: https://github.com/Manasa2810/STT_A11

Screenshots of the code:

1. Dataset Preparation

```
[1] #import all the necessary libraries
    import pandas as pd
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import LabelEncoder
    import torch
    from torch.utils.data import Dataset, DataLoader
    from sklearn.feature_extraction.text import TfidfVectorizer
    import torch.nn as nn
    import torch.optim as optim
    from sklearn.metrics import accuracy_score
    import matplotlib.pyplot as plt
    import time
    import copy
    import os
```

Load the training dataset and test data.

```
# Load the training and test datasets (no headers)
train_df = pd.read_csv(
    "https://raw.githubusercontent.com/clairett/pytorch-sentiment-classification/master/data/SST2/train.tsv",
    sep='\t',
    header=None,
    names=["sentence", "label"]
)

test_df = pd.read_csv(
    "https://raw.githubusercontent.com/clairett/pytorch-sentiment-classification/master/data/SST2/test.tsv",
    sep='\t',
    header=None,
    names=["sentence", "label"]
)
```

```
[3] train_df.head()
```

	sentence	label
0	a stirring , funny and finally transporting re...	1
1	apparently reassembled from the cutting room f...	0
2	they presume their audience wo n't sit still f...	0
3	this is a visually stunning rumination on love...	1
4	jonathan parker 's bartleby should have been t...	1

```
[4] test_df.head()
```

	sentence	label
0	no movement , no yuks , not much of anything	0
1	a gob of drivel so sickly sweet , even the eag...	0
2	gangs of new york is an unapologetic mess , wh...	0
3	we never really feel involved with the story ,...	0
4	this is one of polanski 's best films	1

Next steps: Generate code with test_df | View recommended plots | New interactive sheet

Use 20% of the training dataset as the validation set.

```
# Split into train and validation sets
train_texts, val_texts, train_labels, val_labels = train_test_split(
    train_df["sentence"], train_df["label"], test_size=0.2, random_state=42
)
```

Construct a Multi-Layer Perceptron (MLP) model.

```
[6]

# Initialize the vectorizer
vectorizer = TfidfVectorizer(max_features=10000)  # You can tune max_features

# Fit on training data and transform train/val
X_train = vectorizer.fit_transform(train_texts).toarray()
X_val = vectorizer.transform(val_texts).toarray()
```

```
X_train.shape
```

```
(5536, 10000)
```

```
[8]
y_train = torch.tensor(train_labels.values, dtype=torch.long)
y_val = torch.tensor(val_labels.values, dtype=torch.long)

X_train = torch.tensor(X_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
```

```
[9] X_train.shape
```

```
torch.Size([5536, 10000])
```

The parameter should be with:
hidden_sizes=[512, 256, 128, 64]
Output should have two labels.

```
class MLPClassifier(nn.Module):
    def __init__(self, input_dim, hidden_sizes, output_dim=2):
        super(MLPClassifier, self).__init__()

        layers = []
        in_dim = input_dim
        for h in hidden_sizes:
            layers.append(nn.Linear(in_dim, h))
            layers.append(nn.ReLU())
            in_dim = h
        layers.append(nn.Linear(in_dim, output_dim))  # Output layer

        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)
```

```
[11] input_dim = X_train.shape[1]
     hidden_sizes = [512, 256, 128, 64]

     model = MLPClassifier(input_dim=input_dim, hidden_sizes=hidden_sizes)
```

```
print(model.state_dict)
```

```
<bound method Module.state_dict of MLPClassifier(
  (model): Sequential(
    (0): Linear(in_features=10000, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=128, bias=True)
    (5): ReLU()
    (6): Linear(in_features=128, out_features=64, bias=True)
    (7): ReLU()
    (8): Linear(in_features=64, out_features=2, bias=True)
  )
)>
```

Count the number of trainable parameters in the model using the automated function

```
[13] def count_parameters(model):
         return sum(p.numel() for p in model.parameters() if p.requires_grad)

     print("Total trainable parameters:", count_parameters(model))
```

```
Total trainable parameters: 5293122
```

Train the model with 10 epochs and create the best-performing model (checkpoint.pt)

```
[14]
     # Define loss and optimizer
     criterion = nn.CrossEntropyLoss()
     optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

Lecture22: TOC 202 | (180) Multitape Tur | CS 203: Software To | Inference Time Eval | STT_11.ipynb - Cola | STT_A11 - Google D

colab.research.google.com/drive/1XIiJ8LxjzUdWzRSn4JiIA2O9N3pzHCqz?usp=chrome_ntp#scrollTo=GbduF5jRS9Xq

Commands + Code + Text   T4 RAM Disk   Toggle header visibility

```
epochs = 10
train_losses = []
val_accuracies = []
best_val_acc = 0.0

for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    train_losses.append(loss.item())

    # Validation
    model.eval()
    with torch.no_grad():
        val_preds = model(X_val)
        val_preds_labels = torch.argmax(val_preds, dim=1)
        val_acc = accuracy_score(y_val.cpu(), val_preds_labels.cpu())
        val_accuracies.append(val_acc)

        # Save best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), "checkpoint.pt")
```

✓ 0s completed at 11:10 PM

Search   ENG IN   11:23 PM 4/20/2025

```
        optimizer.step()

        train_losses.append(loss.item())

        # Validation
        model.eval()
        with torch.no_grad():
            val_preds = model(X_val)
            val_preds_labels = torch.argmax(val_preds, dim=1)
            val_acc = accuracy_score(y_val.cpu(), val_preds_labels.cpu())
            val_accuracies.append(val_acc)

            # Save best model
            if val_acc > best_val_acc:
                best_val_acc = val_acc
                torch.save(model.state_dict(), "checkpoint.pt")

        print(f"Epoch {epoch+1}/{epochs} - Loss: {loss.item():.4f} - Val Acc: {val_acc:.4f}")
```

```
Epoch 1/10 - Loss: 0.6929 - Val Acc: 0.5152
Epoch 2/10 - Loss: 0.6922 - Val Acc: 0.5152
Epoch 3/10 - Loss: 0.6913 - Val Acc: 0.5152
Epoch 4/10 - Loss: 0.6899 - Val Acc: 0.5152
Epoch 5/10 - Loss: 0.6879 - Val Acc: 0.5152
Epoch 6/10 - Loss: 0.6845 - Val Acc: 0.5152
Epoch 7/10 - Loss: 0.6798 - Val Acc: 0.5152
Epoch 8/10 - Loss: 0.6736 - Val Acc: 0.5152
Epoch 9/10 - Loss: 0.6658 - Val Acc: 0.5152
Epoch 10/10 - Loss: 0.6561 - Val Acc: 0.5152
```

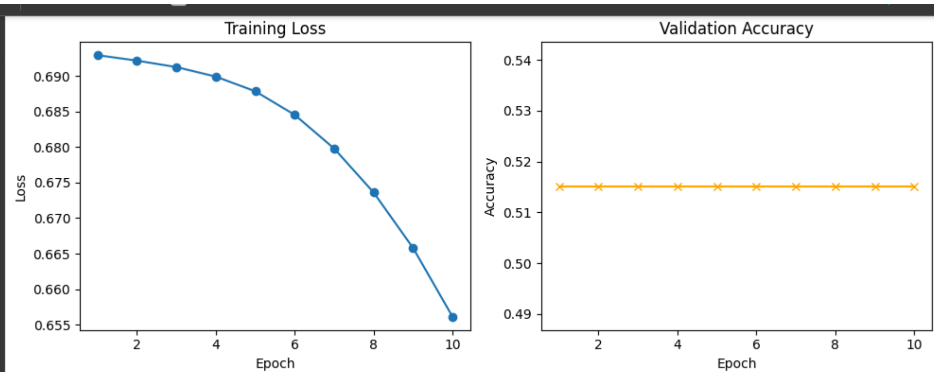
Plot the validation accuracy + loss (epochs vs accuracy-loss).

```
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(range(1, epochs + 1), train_losses, marker='o')
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")

plt.subplot(1, 2, 2)
plt.plot(range(1, epochs + 1), val_accuracies, marker='x', color='orange')
plt.title("Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")

plt.tight_layout()
plt.show()
```




```
[17] model.load_state_dict(torch.load("checkpoint.pt"))
     model.eval()
```

```python
model.load_state_dict(torch.load("checkpoint.pt"))
model.eval()
```

```
MLPClassifier(
  (model): Sequential(
    (0): Linear(in_features=10000, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=128, bias=True)
    (5): ReLU()
    (6): Linear(in_features=128, out_features=64, bias=True)
    (7): ReLU()
    (8): Linear(in_features=64, out_features=2, bias=True)
  )
)
```

[19]
```python
test_texts = test_df["sentence"]
X_test = vectorizer.transform(test_texts).toarray()
X_test = torch.tensor(X_test, dtype=torch.float32)
test_labels = test_df["label"]
y_test = torch.tensor(test_labels.values, dtype=torch.long)
```

```python
import time

def get_avg_inference_time(model, X_test, num_runs=10):
    times = []
    with torch.no_grad():
        for _ in range(num_runs):
            start = time.time()
            _ = model(X_test)
            end = time.time()
            times.append((end - start) * 1000)
    return sum(times) / len(times)  # in ms
```

Lecture22: TOC 2025 | (180) Multitape Turing Ma | CS 203: Software Tools & | STT_11.ipynb - Colab | STT_A11 - Google Docs

colab.research.google.com/drive/1XIiJ8LxjzUdWzRSn4JiIA2O9N3pzHCqz?usp=chrome_ntp#scrollTo=CoUWTYifnl5o

```python
model.eval()
with torch.no_grad():
    outputs = model(X_test)
    preds = torch.argmax(outputs, dim=1)

orig_acc = accuracy_score(y_test.cpu(), preds.cpu())
print(f"[Original] Accuracy: {orig_acc:.4f}")
orig_time = get_avg_inference_time(model, X_test)
print(f"[Original] Inference Time: {orig_time:.2f} ms")
torch.save(model.state_dict(), "original.pt")
orig_size = os.path.getsize("original.pt") / (1024 * 1024)
print(f"[Original] Model Size: {orig_size:.2f} MB")
```

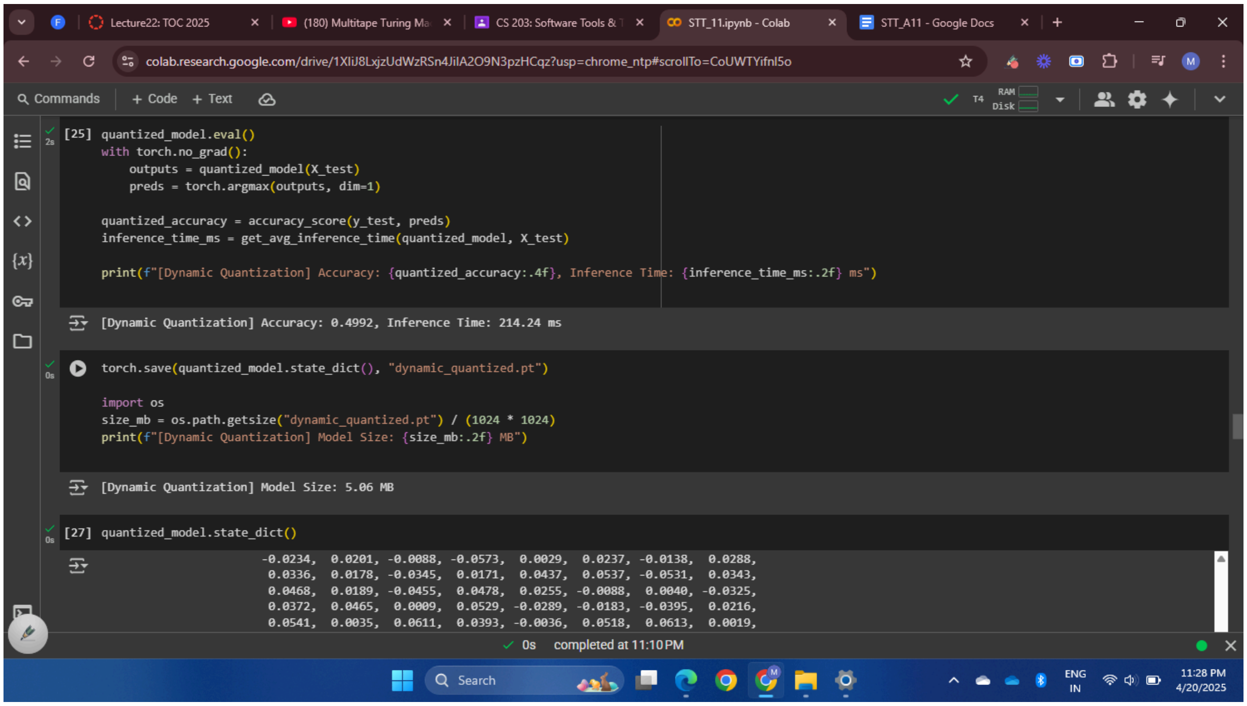
```
[Original] Accuracy: 0.4992
[Original] Inference Time: 313.35 ms
[Original] Model Size: 20.20 MB
```

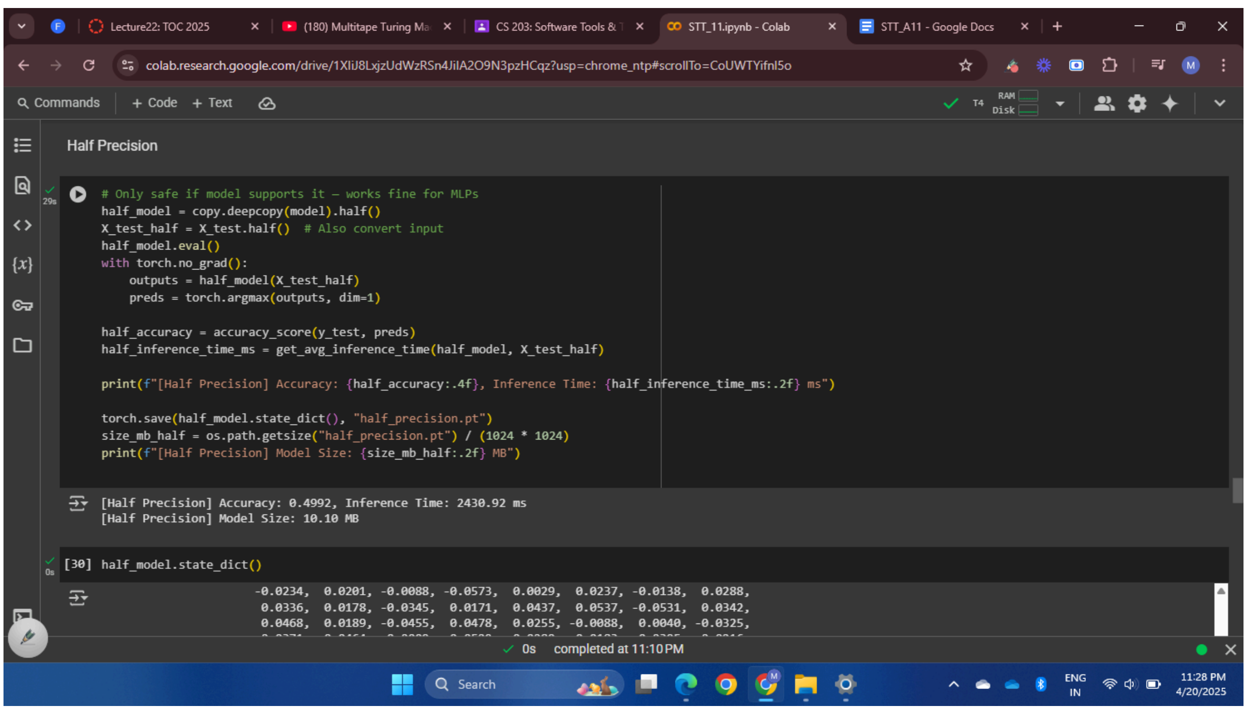
Dynamic Quantization with INT4

```python
# Only linear layers can be quantized dynamically
quantized_model = torch.quantization.quantize_dynamic(
    copy.deepcopy(model),  # make a copy of the model
    {nn.Linear},  # only quantize Linear layers
    dtype=torch.qint8  # use INT8 quantization
)
```

0s completed at 11:10 PM

11:27 PM 4/20/2025


Lecture22: TOC 2025
(180) Multitape Turing Ma
CS 203: Software Tools &
STT_11.ipynb - Colab
STT_A11 - Google Docs
colab.research.google.com/drive/1XliJ8LxjzUdWzRSn4JilA2O9N3pzHCqz?usp=chrome_ntp#scrollTo=CoUWTYifnl5o
Commands
+ Code
+ Text
[25] quantized_model.eval()
with torch.no_grad():
outputs = quantized_model(X_test)
preds = torch.argmax(outputs, dim=1)
quantized_accuracy = accuracy_score(y_test, preds)
inference_time_ms = get_avg_inference_time(quantized_model, X_test)
print(f"[Dynamic Quantization] Accuracy: {quantized_accuracy:.4f}, Inference Time: {inference_time_ms:.2f} ms")
[Dynamic Quantization] Accuracy: 0.4992, Inference Time: 214.24 ms
torch.save(quantized_model.state_dict(), "dynamic_quantized.pt")
import os
size_mb = os.path.getsize("dynamic_quantized.pt") / (1024 * 1024)
print(f"[Dynamic Quantization] Model Size: {size_mb:.2f} MB")
[Dynamic Quantization] Model Size: 5.06 MB
[27] quantized_model.state_dict()
-0.0234, 0.0201, -0.0088, -0.0573, 0.0029, 0.0237, -0.0138, 0.0288,
0.0336, 0.0178, -0.0345, 0.0171, 0.0437, 0.0537, -0.0531, 0.0343,
0.0468, 0.0189, -0.0455, 0.0478, 0.0255, -0.0088, 0.0040, -0.0325,
0.0372, 0.0465, 0.0009, 0.0529, -0.0289, -0.0183, -0.0395, 0.0216,
0.0541, 0.0035, 0.0611, 0.0393, -0.0036, 0.0518, 0.0613, 0.0019,
0s completed at 11:10 PM
11:28 PM
4/20/2025


STT_11.ipynb - Colab
Half Precision
# Only safe if model supports it — works fine for MLPs
half_model = copy.deepcopy(model).half()
X_test_half = X_test.half() # Also convert input
half_model.eval()
with torch.no_grad():
outputs = half_model(X_test_half)
preds = torch.argmax(outputs, dim=1)
half_accuracy = accuracy_score(y_test, preds)
half_inference_time_ms = get_avg_inference_time(half_model, X_test_half)
print(f"[Half Precision] Accuracy: {half_accuracy:.4f}, Inference Time: {half_inference_time_ms:.2f} ms")
torch.save(half_model.state_dict(), "half_precision.pt")
size_mb_half = os.path.getsize("half_precision.pt") / (1024 * 1024)
print(f"[Half Precision] Model Size: {size_mb_half:.2f} MB")
[Half Precision] Accuracy: 0.4992, Inference Time: 2430.92 ms
[Half Precision] Model Size: 10.10 MB
[30] half_model.state_dict()
-0.0234, 0.0201, -0.0088, -0.0573, 0.0029, 0.0237, -0.0138, 0.0288,
0.0336, 0.0178, -0.0345, 0.0171, 0.0437, 0.0537, -0.0531, 0.0342,
0.0468, 0.0189, -0.0455, 0.0478, 0.0255, -0.0088, 0.0040, -0.0325,
0s completed at 11:10 PM
11:28 PM
4/20/2025

Fill the table for different quantization techniques.

```python
[31]
# Final Table
data = {
    "Model Name": ["Original", "Dynamic", "Half"],
    "Accuracy (Out of 100)": [round(orig_acc * 100, 2), round(quantized_accuracy * 100, 2), round(half_accuracy * 100, 2)],
    "Storage (In MB)": [round(orig_size, 2), round(size_mb, 2), round(size_mb_half, 2)],
    "Inference time (In ms)": [round(orig_time, 2), round(inference_time_ms, 2), round(half_inference_time_ms, 2)],
}

results_df = pd.DataFrame(data)
print("\n Final Quantization Comparison Table:")
print(results_df.to_markdown(index=False))
```

```
 Final Quantization Comparison Table:
| Model Name   |   Accuracy (Out of 100) |   Storage (In MB) |   Inference time (In ms) |
|:-------------|------------------------:|------------------:|-------------------------:|
| Original     |                   49.92 |              20.2 |                   313.35 |
| Dynamic      |                   49.92 |              5.06 |                   214.24 |
| Half         |                   49.92 |              10.1 |                  2430.92 |
```

Lecture22: TOC 2025 | (180) Multitape Turing Ma | CS 203: Software Tools & | STT_11.ipynb - Colab | STT_A11 - Google Docs

colab.research.google.com/drive/1XIiJ8LxjzUdWzRSn4JiA2O9N3pzHCqz?usp=chrome_ntp#scrollTo=CoUWTYifnl5o

Commands | + Code | + Text

Half Precision With GPU

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


half_model_gpu = copy.deepcopy(model).half().to(device)
X_test_half_gpu = X_test.half().to(device)


half_model_gpu.eval()
with torch.no_grad():
    outputs_gpu = half_model_gpu(X_test_half_gpu)
    preds_gpu = torch.argmax(outputs_gpu, dim=1)

half_accuracy_gpu = accuracy_score(y_test.cpu(), preds_gpu.cpu())

# Measure average inference time over multiple runs
def get_avg_inference_time_gpu(model, X_test, num_runs=10):
    times = []
    with torch.no_grad():
        for _ in range(num_runs):
            start = time.time()
            _ = model(X_test)
            if device.type == "cuda":
                torch.cuda.synchronize()  # wait for GPU to finish
            end = time.time()
            times.append((end - start) * 1000)  # ms
```

0s completed at 11:10 PM

Search | ENG IN | 11:29 PM 4/20/2025

```python
def get_avg_inference_time_gpu(model, X_test, num_runs=10):
    times = []
    with torch.no_grad():
        for _ in range(num_runs):
            start = time.time()
            _ = model(X_test)
            if device.type == "cuda":
                torch.cuda.synchronize()  # wait for GPU to finish
            end = time.time()
            times.append((end - start) * 1000)  # ms
    return sum(times) / len(times)

avg_time_ms = get_avg_inference_time_gpu(half_model_gpu, X_test_half_gpu)

# Save and get model size
torch.save(half_model_gpu.state_dict(), "half_precision_gpu.pt")
size_mb_half_gpu = os.path.getsize("half_precision_gpu.pt") / (1024 * 1024)

# Final Output
print(f"[Half Precision - {device.type.upper()}] Accuracy: {half_accuracy_gpu:.4f}, Inference Time (avg): {avg_time_ms:.2f} ms")
print(f"[Half Precision GPU] Model Size: {size_mb_half_gpu:.2f} MB")
```

```
[Half Precision - CUDA] Accuracy: 0.4992, Inference Time (avg): 1.36 ms
[Half Precision GPU] Model Size: 10.10 MB
```

```python
# Final Table
data = {
    "Model Name": ["Original", "Dynamic", "Half", "Half( WITH GPU)"],
    "Accuracy (Out of 100)": [round(orig_acc * 100, 2), round(quantized_accuracy * 100, 2), round(half_accuracy * 100, 2),round(half_accuracy_gpu * 100, 2)],
    "Storage (In MB)": [round(orig_size, 2), round(size_mb, 2), round(size_mb_half, 2),round(size_mb_half_gpu, 2)],
    "Inference time (In ms)": [round(orig_time, 2), round(inference_time_ms, 2), round(half_inference_time_ms, 2),round(avg_time_ms, 2)],
}

results_df = pd.DataFrame(data)
print("\n Final Quantization Comparison Table:")
print(results_df.to_markdown(index=False))
```

```
 Final Quantization Comparison Table:
| Model Name      |   Accuracy (Out of 100) |   Storage (In MB) |   Inference time (In ms) |
|:----------------|------------------------:|------------------:|-------------------------:|
| Original        |                   49.92 |              20.2 |                   313.35 |
| Dynamic         |                   49.92 |              5.06 |                   214.24 |
| Half            |                   49.92 |              10.1 |                  2430.92 |
| Half( WITH GPU) |                   49.92 |              10.1 |                     1.36 |
```

Half-precision (FP16) models are slower on CPU because most CPUs lack native support for FP16 arithmetic and optimized instructions to handle it efficiently. As a result, CPUs often emulate FP16 operations using slower FP32 computations, which introduces overhead and significantly increases inference time. This makes FP16 unsuitable for CPU inference, even though it performs exceptionally well on GPUs that are designed to accelerate half-precision operations.