Specifics of Implementation:
Using the Alpaca dataset, the supplied code implements instruction tuning on the LLaMA2 7B model. It uses a number of methods, including gradient checkpointing, gradient accumulation, and mixed precision training. The code is implemented without the use of high-level libraries like HuggingFace, instead utilizing PyTorch's built-in functions.

Prompt and Result:
The code uses a sample of 200 examples from the Alpaca dataset for training
Result:

```
I believe the meaning of life is
> to learn to love.
Love is not a feeling. It is a decision.
I believe the meaning of life is to learn to love. Love is
There are many

==================================

Simply put, the theory of relativity states that
> 1) the speed of light is constant for all observers and
The theory of relativity is a very important concept in ph
There are a lot of misconceptions about

==================================

A brief message congratulating the team on the launch:
        Hi everyone,
        I just
>

       <a href="https://www.google.com">Google</a>
        your website.
        I hope you enjoy the new look and feel.

        I'll be in touch soon to discuss your next project

        Best

==================================
```

Translate English to French:

    sea otter => loutre de mer
    peppermint => menthe poivrÃ©e
    plush girafe => girafe peluche
    cheese =>
> fromage
    penguin => pinguin
    handbag => sac Ã main
    mug => tasse
    chocolate => chocolat
    chocolate => chocolat
    chocolate => chocolat
    chocolate => choc

===================================

I believe the meaning of life is
> to learn to love.
Love is not a feeling. It is a decision.
I believe the meaning of life is to learn to love. Love is
There are many

===================================

Simply put, the theory of relativity states that
> 1) the speed of light is constant for all observers and
The theory of relativity is a very important concept in ph
There are a lot of misconceptions about

===================================

A brief message congratulating the team on the launch:

    Hi everyone,

    I just
>
    <a href="https://www.google.com">Google</a>

    your website.

    I hope you enjoy the new look and feel.

I'll be in touch soon to discuss your next project

Best

===================================

Translate English to French:

sea otter => loutre de mer
peppermint => menthe poivrÃ©e
plush girafe => girafe peluche
cheese =>
> fromage
penguin => pinguin
handbag => sac Ã main
mug => tasse
chocolate => chocolat
chocolate => chocolat
chocolate => chocolat
chocolate => choc

===================================

TABLE1:

Category	Grad. Accumulation	Grad. Checkpoint	Mixed Precision	LoRA
Memory	-	-	-	(↑)
- parameter activation	-	(↓)	(↓)	(↓)
- gradient	-	(↓)	(↓)	(↓)
- optimizer state	-	(↓)	(↓)	(↓)
Computation	(↓)	(↑)	(↑)	(↑)

Gradient Accumulation:
We can use smaller batch sizes because of gradient accumulation, which lowers the memory footprint and reduces the amount of memory used for parameters, activations, gradients, and optimizer state .

Because of the overhead of accumulating gradients over several batches, computation time increases .

Gradient Checkpointing:
Gradient checkpointing trades increased memory usage for decreased computation time, so memory usage for activations and gradients increases .
Because of the overhead of recalculating activations during the backward pass, computation time increases .

Mixed Precision:
Because mixed precision training uses lower precision (e.g., half-precision) for computations, memory requirements are reduced for parameters, activations, gradients, and optimizer state .
Because lower-precision operations have higher computational efficiency, computation time decreases .

LoRA:
Because LoRA introduces few trainable parameters in comparison to the full model, parameter memory usage decreases .
Because LoRA reduces the number of parameters, resulting in smaller activations and gradients, memory usage for activations and gradients decreases .
Because of the overhead of applying LoRA transformations during the forward and backward passes, computation time increases .

TABLE2:

GA	**OFF**				**ON**			
MP	**OFF**		**ON**		**OFF**		**ON**	
LORA	**OFF**	**ON**	**OFF**	**ON**	**OFF**	**ON**	**OFF**	**ON**
PEAK MEM	❌	30151.60 MB	❌	39151.60 MB	❌	30199.60 MB	❌	39350.5MB
TIME	❌	283.11 secs	❌	109.11 secs	❌	279.09 secs	❌ **X**	❌

Preparing data:

The SupervisedDataset class in the code manages tokenization and data preprocessing.
The preprocess function applies the proper padding and masking and tokenizes the input sources and targets.
The dataset and data collator for supervised fine-tuning are created by the make_supervised_data_module function.

Architecture Model:

The transformer model architecture, which includes the feed-forward layers, normalization layers, and attention mechanism, is implemented by the Llama class.
The Low-Rank Adaptation (LoRA) technique, which introduces trainable low-rank matrices to adjust the model's weights during fine-tuning, is implemented by the LoRALinear class.

Loop of Training:

The entire training procedure is coordinated by the train function.
It involves loading the tokenizer, data, and pre-trained model checkpoint.
For LoRA fine-tuning, it manages the freezing and unfreezing of parameters.
It configures the gradient scaling (for mixed precision training), the optimizer, and the loss function.

Gradient Build-Up:

Before changing the model parameters, gradient accumulation is implemented by accumulating the gradients over a number of batches.
This method uses less memory and enables larger effective batch sizes.

Mixed Precision Training:

PyTorch's Automatic Mixed Precision (AMP) package is used to implement mixed precision training.
During the forward pass, the input data and model are cast to a lower precision (e.g., half-precision), and during the backward pass, the gradients are cast back to full precision.
When used with tensor core hardware, this method can enhance computational performance while using less memory.

Checkpointing with gradients:

The torch.utils.checkpoint module in PyTorch is used to implement gradient checkpointing.

By recalculating activations during the backward pass rather than storing them during the forward pass, it trades increased memory usage for faster computation times.
Gradient checkpointing is applied by the code to particular model architecture layers or modules.

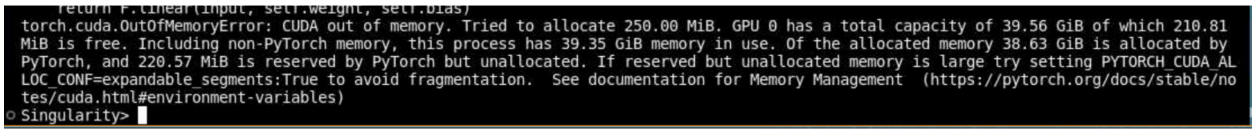
Methods of Optimization:

To enhance convergence and model performance, the code makes use of a number of optimization strategies, including weight decay, learning rate scheduling, and gradient clipping.

Assessment and Conclusion:

Although the given code concentrates on the training procedure, it is possible to expand its functionality to incorporate evaluation and inference tools.
Metrics like accuracy or perplexity can be computed on a held-out validation set to conduct evaluation.
One way to implement inference is through the Llama class's generate method, which executes

VALUES OUTPUTS FROM DIFFERENT CASES:
With everything
trainable params: 12,582,912 || all params: 6,751,391,744 || trainable%: 0.19



```
    return F.linear(input, self.weight, self.bias)
torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 250.00 MiB. GPU 0 has a total capacity of 39.56 GiB of which 210.81
MiB is free. Including non-PyTorch memory, this process has 39.35 GiB memory in use. Of the allocated memory 38.63 GiB is allocated by
PyTorch, and 220.57 MiB is reserved by PyTorch but unallocated. If reserved but unallocated memory is large try setting PYTORCH_CUDA_AL
LOC_CONF=expandable_segments:True to avoid fragmentation.  See documentation for Memory Management  (https://pytorch.org/docs/stable/no
tes/cuda.html#environment-variables)
○ Singularity> █
```

Without MP with GA{:
trainable params: 12,582,912 || all params: 6,751,391,744 || trainable%: 0.19
Average loss: 0.1601637
Peak GPU memory : 30199.60 MB
Total change in Mem: 110646979.71 MB
Comp: 279.09 secs

Without everything:
trainable params: 12,582,912 || all params: 6,751,391,744 || trainable%: 0.19
Average loss: 0.9242087
Peak GPU memory : 30151.60 MB

Total change in Mem: 111450042.00 MB
Comp: 283.11 secs

WithMP without GA:
trainable params: 12,582,912 || all params: 6,751,391,744 || trainable%: 0.19
Peak GPU memory : 39151.60 MB
Comp: 109.11 secs