number of words and `d` is the dimension of the word embeddings.

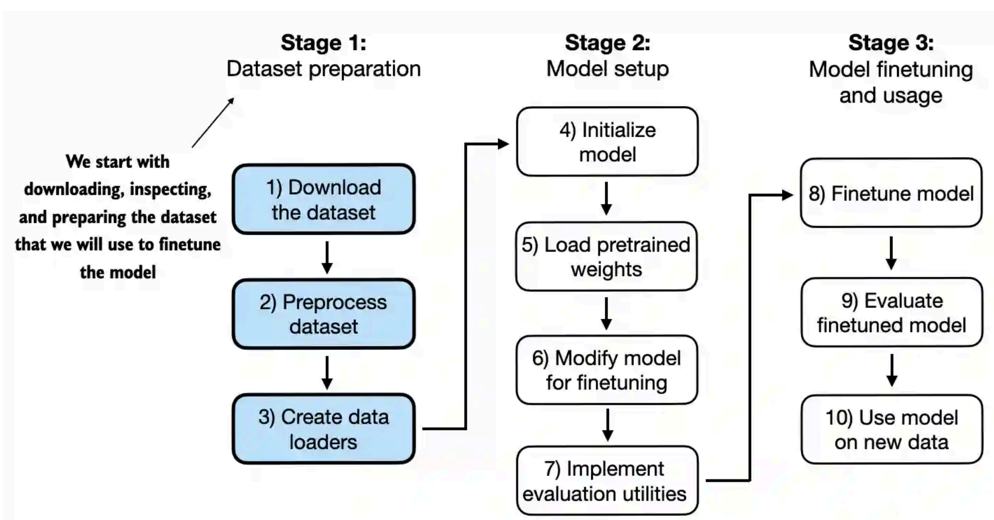
Computing Attention Scores: For each word, we compute attention scores with respect to all other words. Since there are `n` words and for each word, we perform a dot product with `n` key vectors, the complexity of this step is `O(n^2d)`.

Computing the Weighted Sum: This step involves a weighted sum operation for each word, which has a complexity of `O(nd)`.

When we sum up the complexities of all three steps, the dominant term is the second step, which grows quadratically with the length of the sequence `(n)`. Therefore, the overall complexity of the self-attention mechanism in the vanilla transformer architecture is `O(n^2d)` or squared compute cost. This quadratic growth becomes a bottleneck for long sequences, making it less scalable compared to architectures with linear or sub-linear complexities.

The critical point is that the number of computations grows with the product `(Np * Nci)` of the sequence length `(Np)` in the previous layer. Here, `Nci` represent the current token in the sequence and we compute attention scores for each token in the current sequence. In mathematical terms, this product represents the square of the sequence length `(N^2)`.

Fine-Tuning Mistral-7b



Mistral 7B v0.1, with 7.3 billion parameters, is the first LLM introduced by Mistral AI. The main novel techniques used in Mistral 7B's architecture are:

- Sliding Window Attention: Replace the full attention (square compute cost) with a sliding window based attention where each token can attend to at most 4,096 tokens from the previous layer (linear compute cost). This mechanism enables

Mistral 7B to handle longer sequences, where higher layers can access historical information beyond the window size of 4,096 tokens.

- Grouped-query Attention: used in Llama 2 as well, the technique optimizes the inference process (reduce processing time) by caching the key and value vectors for previously decoded tokens in the sequence.

https://magazine.sebastianraschka.com/p/finetuning-large-language-models

1. Split the dataset in Train and Test

In [46]:
```
# check device
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

f"Device: {device}"
```

Out[46]:
```
'Device: cuda'
```

In [47]:
```
# Let's print out data first
df2.sample(5)
```

Out[47]:

	texts	label
1485	Less: Other Other - Check Processing Fees	Expense
1380	Header BAD DEBT	Income
1921	Plus: Misc. Other Income Additional Other Income	Income
1410	Plus: Misc. Other Income Deposit Fee	Income
2037	Less: Office Expenses Employee Relations	Expense

In [48]:
```
# define a dict to encode labels
labels = {
          'Expense': 0,
          'Income': 1
         }
```

In [49]:
```
# Now, let's update the labels with 0 or 1
df2["label"] = df2["label"].map(labels)
df2["label"].value_counts()
```

Out[49]:
```
label
1    3000
0    3000
Name: count, dtype: int64
```

In [50]:
```
df2 = df2.sample(frac=1, random_state=42).reset_index(drop=True)
df2.head(5)
```

Out[50]:

	texts	label
0	Less: Gas Gas - Vacancy	0
1	Total TOTAL INCOME	1
2	Less: Marketing Permanent Signage	0
3	Gross Potential Rent Potential Rent	1
4	Less: Unit Turnover RR - Appliance - Dishwasher	0

In [51]:
```
train_df = df2.sample(frac=0.75, random_state=200)
rem_df = df2.drop(train_df.index)

# validation set
val_df = rem_df.sample(frac=0.5, random_state=200)
test_df = rem_df.drop(val_df.index)

train_df.to_csv("train.csv", index=None)
val_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

2. Create data loader

In [52]:
```
# Tokenizer
tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.1")
tokenizer.pad_token = tokenizer.eos_token
```

```
tokenizer_config.json:   0%|          | 0.00/967 [00:00<?, ?B/s]
tokenizer.model:   0%|          | 0.00/493k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/1.80M [00:00<?, ?B/s]
special_tokens_map.json:   0%|          | 0.00/72.0 [00:00<?, ?B/s]
```

In [53]:
```
tokenizer.eos_token_id
```

Out[53]: 2

In [54]:
```
class customDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None):
        self.data = pd.read_csv(csv_file)

        # as string
        self.data['texts'] = self.data['texts'].astype(str)

        self.encoded_texts = [tokenizer.encode(text) for text in self.data['

        # If max_length is not specified, use the longest text length
        if max_length is None:
            self.max_length = max(len(text) for text in self.encoded_texts)
        else:
            self.max_length = max_length

        # Pad sequences to the longest sequence
        self.encoded_texts = [
```

```
            encoded_text + [tokenizer.eos_token_id] * (self.max_length - ler
            for encoded_text in self.encoded_texts
        ]

    def __getitem__(self, index):
        encoded = self.encoded_texts[index]
        label = self.data.iloc[index]["label"]
        return (
            torch.tensor(encoded, dtype=torch.long),
            torch.tensor(label, dtype=torch.long)
        )

    def __len__(self):
        return len(self.data)
```

In [55]:

```
train_dataset = customDataset(
                                csv_file="train.csv",
                                tokenizer=tokenizer,
                                max_length=None
                             )

print(train_dataset.max_length)
```

25

In [56]:

```
val_dataset = customDataset(
                                csv_file="validation.csv",
                                tokenizer=tokenizer,
                                max_length=None
                             )

print(val_dataset.max_length)
```

23

In [57]:

```
test_dataset = customDataset(
                                csv_file="test.csv",
                                tokenizer=tokenizer,
                                max_length=None
                             )

print(test_dataset.max_length)
```

25

Next, we use the dataset to instantiate the data loaders

In [58]:

```
from torch.utils.data import DataLoader

num_workers = 2
batch_size = 25

torch.manual_seed(123)

train_loader = DataLoader(
                            dataset=train_dataset,
```

```
                                batch_size=batch_size,
                                shuffle=True,
                                num_workers=num_workers,
                                drop_last=True,
                            )


val_loader = DataLoader(
                                dataset=val_dataset,
                                batch_size=batch_size,
                                num_workers=num_workers,
                                drop_last=False,
                            )


test_loader = DataLoader(
                                dataset=test_dataset,
                                batch_size=batch_size,
                                num_workers=num_workers,
                                drop_last=False,
                            )
```

In [59]:

```
print("Train loader:")
for input_batch, target_batch in train_loader:
    pass

print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)
```

```
Train loader:
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
Input batch dimensions: torch.Size([25, 25])
Label batch dimensions torch.Size([25])
```

In [130…

```
print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")
```

```
180 training batches
30 validation batches
30 test batches
```

3. Initializing an LLM with pre-trained weights

Let's load a pre-trained LLM with 4-bit quantization precision for its weights.

- `load_in_4bit=True` : This flag indicates that the model's weights should be loaded using 4-bit precision. Using 4-bit precision reduces the model's memory footprint significantly compared to standard 32-bit or 16-bit precision.

- `bnb_4bit_quant_type="nf4"` : Specifies the type of 4-bit quantization to use. "nf4" stands for Normal Float 4-bit. This quantization method aims to maintain a normal distribution of floating-point values, which helps preserve the model's performance despite the lower precision.

- `bnb_4bit_compute_dtype=torch.bfloat16` : Sets the computation data type to bfloat16 (Brain Floating Point 16). While the model's weights are stored in 4-bit precision, the computations during inference or training will be carried out in bfloat16. This format is a 16-bit floating-point type that balances computational efficiency and numerical stability, providing a good trade-off between precision and performance.

- `bnb_4bit_quant_storage=torch.bfloat16` : Specifies that the storage format for the quantized weights should be bfloat16. This helps in maintaining a balance between reduced memory usage and preserving enough information to ensure model accuracy.

Key difference between `load_in_4bit` and `bnb_4bit_compute_dtype` :

Weight Storage: `load_in_4bit=True` ensures that the model's weights are stored in 4-bit precision, reducing memory usage.

Computation Precision: `bnb_4bit_compute_dtype=torch.bfloat16` sets the precision for the computations. While weights are in 4-bit, the activations and intermediate results during forward and backward passes will use torch.bfloat16, combining the benefits of quantization with the advantages of higher precision computations.

- `Mistral-7b` @ `32-bit (4-bytes)` precision would be 28gb memory. (7bx4) | 1 byte = 8 bit

- Quantized `Mistral-7b` @ `4-bit (0.5 bytes)` precision would be 3.5gb memory. (7bx0.5)

In [131…

```
bnb_config = BitsAndBytesConfig(
                                    load_in_4bit=True, # This flag indicates
                                    bnb_4bit_quant_type="nf4", # Specifies t
                                    bnb_4bit_compute_dtype=torch.bfloat16, #
                                    bnb_4bit_use_double_quant=True, # Enable
                                    bnb_4bit_quant_storage=torch.bfloat16 #
                                )
```

```
bnb_config
```

Out[131…

```
BitsAndBytesConfig {
  "_load_in_4bit": true,
  "_load_in_8bit": false,
  "bnb_4bit_compute_dtype": "bfloat16",
  "bnb_4bit_quant_storage": "bfloat16",
  "bnb_4bit_quant_type": "nf4",
  "bnb_4bit_use_double_quant": true,
  "llm_int8_enable_fp32_cpu_offload": false,
  "llm_int8_has_fp16_weight": false,
  "llm_int8_skip_modules": null,
  "llm_int8_threshold": 6.0,
  "load_in_4bit": true,
  "load_in_8bit": false,
  "quant_method": "bitsandbytes"
}
```

In [132…

```
# Set the device (replace 'cuda:0' with the appropriate GPU if you have mult
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Set the device for PyTorch
torch.cuda.set_device(device)
```

In [133…

```
device
```

Out[133…

```
device(type='cuda', index=0)
```

In [134…

```
# Before clearing GPU memory
print(torch.cuda.memory_allocated())

# Clear GPU memory
gc.collect()
torch.cuda.empty_cache()
gc.collect()

# After clearing GPU memory
print(torch.cuda.memory_allocated())
```

```
6089384448
4973572608
```

In [135…

```
if torch.cuda.memory_allocated() == 0:
    model_4bit = AutoModelForCausalLM.from_pretrained(
                                                    "mistralai/Mistral-7
                                                    quantization_config=
                                                    device_map="cuda:0"
                                                 )


    model_4bit
```

In [136…

```
model_4bit.hf_device_map
```

Out[136…

```
{'': device(type='cuda', index=0)}
```

In [67]:
```
# save model to disk
#model.save_pretrained(os.getcwd() + "/model")
```

In [68]:
```
# Load the model from disk
#model = AutoModelForCausalLM.from_pretrained(os.getcwd() + "/model")
```

We run into `OOM` error, when attempting to load `mistral-7b` via HF Transformers. As the Mistral model has 7 billion parameters, that would require about 14GB of GPU RAM in half precision (float16), since each parameter is stored in 2 bytes. However, one can shrink down the size of the model using quantization. If the model is quantized to 4 bits (or half a byte per parameter),that requires only about 3.5GB of RAM.

In [137…
```
# Print model configuration attributes
print(f"Model configuration: {model_4bit.config}")
```

```
Model configuration: MistralConfig {
  "_name_or_path": "mistralai/Mistral-7B-v0.1",
  "architectures": [
    "MistralForCausalLM"
  ],
  "attention_dropout": 0.0,
  "bos_token_id": 1,
  "eos_token_id": 2,
  "hidden_act": "silu",
  "hidden_size": 4096,
  "initializer_range": 0.02,
  "intermediate_size": 14336,
  "max_position_embeddings": 32768,
  "model_type": "mistral",
  "num_attention_heads": 32,
  "num_hidden_layers": 32,
  "num_key_value_heads": 8,
  "quantization_config": {
    "_load_in_4bit": true,
    "_load_in_8bit": false,
    "bnb_4bit_compute_dtype": "bfloat16",
    "bnb_4bit_quant_storage": "bfloat16",
    "bnb_4bit_quant_type": "nf4",
    "bnb_4bit_use_double_quant": true,
    "llm_int8_enable_fp32_cpu_offload": false,
    "llm_int8_has_fp16_weight": false,
    "llm_int8_skip_modules": null,
    "llm_int8_threshold": 6.0,
    "load_in_4bit": true,
    "load_in_8bit": false,
    "quant_method": "bitsandbytes"
  },
  "rms_norm_eps": 1e-05,
  "rope_theta": 10000.0,
  "sliding_window": 4096,
  "tie_word_embeddings": false,
  "torch_dtype": "bfloat16",
  "transformers_version": "4.41.2",
  "use_cache": true,
  "vocab_size": 32000
}
```

In [138…

```python
def print_trainable_parameters(model):
    """
    Prints the number of trainable parameters in the model.
    """
    trainable_params = 0
    all_param = 0
    for _, param in model.named_parameters():
        all_param += param.numel()
        if param.requires_grad:
            trainable_params += param.numel()
    print(
        f"trainable params: {trainable_params} || all params: {all_param} ||
    )
```

```
print_trainable_parameters(model_4bit)
```

```
trainable params: 18874368 || all params: 1895047168 || trainable%: 0.995984
07462964
```

Let's test / prompt the model.

In [139…

```
# Test
inputs = tokenizer("Do you have time", return_tensors="pt").to(0)
inputs
```

Out[139…

```
{'input_ids': tensor([[    1, 2378,   368,   506,   727]], device='cuda:0'), 'a
ttention_mask': tensor([[1, 1, 1, 1, 1]], device='cuda:0')}
```

Output is a vector of logits (one for each input token), we convert to a probability distn with a softmax, and can then convert this to a token (eg taking the largest logit, or sampling).

In [140…

```
with torch.no_grad(): # No gradient calculation during inference
    outputs = model_4bit(**inputs)

"""
Output is weighted inputs of the output layer.
"""

print("Outputs:\n", outputs.logits.shape)
print("Outputs Logits:\n", outputs.logits)
```

```
Outputs:
 torch.Size([1, 5, 1])
Outputs Logits:
 tensor([[[0.4914],
         [0.3885],
         [0.3254],
         [0.2467],
         [0.2212]]], device='cuda:0')
```

Convert the logits to a distribution with a softmax

In [141…

```
"""
For multi-class problem, we use softmax.

For binary classification problem, we use sigmoid activation.
"""
#log_probs = outputs.logits.log_softmax(dim=-1)
#print(log_probs.shape) # shape = batch x position x d_vocab

probs = torch.sigmoid(outputs.logits)
probs
```

Out[141…

```
tensor([[[0.6204],
         [0.5959],
         [0.5806],
         [0.5614],
         [0.5551]]], device='cuda:0')
```

In [142…

```
next_token = probs[0, -1].argmax(dim=-1)
# reshape
next_token = next_token.view(1,1)
next_token
```

Out[142…

```
tensor([[0]], device='cuda:0')
```

Append next_token to input tokens

In [143…

```
appended = torch.cat((inputs['input_ids'], next_token), dim=-1)
appended
```

Out[143…

```
tensor([[   1, 2378,  368,  506,  727,    0]], device='cuda:0')
```

In [144…

```
tokenizer.decode(appended.squeeze())
```

Out[144…

```
'<s> Do you have time<unk>'
```

For binary classification problem,

- The goal is to replace and finetune the output layer `lm_head`
- To achieve this, we first freeze the model, meaning that we make all layers non-trainable

In [145…

```
for param in model_4bit.parameters():
    param.requires_grad = False
```

- Then, we replace the output layer `(model.lm_head)`, which originally maps the layer inputs to 32,000 dimensions (the size of the vocabulary)
- Since we finetune the model for binary classification (predicting 2 classes, "Income" and "Expense"), we can replace the output layer as shown below, which will be trainable by default.

In [146…

```
model_4bit.lm_head
```

Out[146…

```
Linear(in_features=4096, out_features=1, bias=False)
```

In [147…

```
model_4bit.get_input_embeddings().embedding_dim
```

Out[147…

```
4096
```

`BCELoss` function expects inputs to be probabilities (values between 0 and 1). The loss function measures the discrepancy between predicted probs and true binary labels. By apply `sigmoid` function to the logits, we transform them into probabilities, making them suitable inputs for `nn.BCELoss`.

Let's update `num_classes` to 1 so that it is compatible with `nn.BCELoss` function.

In [148…
```
num_classes = 1
model_4bit.lm_head = torch.nn.Linear(in_features=model_4bit.get_input_embedd
model_4bit.lm_head
```

Out[148…
```
Linear(in_features=4096, out_features=1, bias=False)
```

In [149…
```
model_4bit
```

```
Out[149… MistralForCausalLM(
  (model): MistralModel(
    (embed_tokens): Embedding(32000, 4096)
    (layers): ModuleList(
      (0-31): 32 x MistralDecoderLayer(
        (self_attn): MistralSdpaAttention(
          (q_proj): lora.Linear4bit(
            (base_layer): Linear4bit(in_features=4096, out_features=4096, b
ias=False)
            (lora_dropout): ModuleDict(
              (default): Dropout(p=0.01, inplace=False)
            )
            (lora_A): ModuleDict(
              (default): Linear(in_features=4096, out_features=32, bias=Fal
se)
            )
            (lora_B): ModuleDict(
              (default): Linear(in_features=32, out_features=4096, bias=Fal
se)
            )
            (lora_embedding_A): ParameterDict()
            (lora_embedding_B): ParameterDict()
          )
          (k_proj): lora.Linear4bit(
            (base_layer): Linear4bit(in_features=4096, out_features=1024, b
ias=False)
            (lora_dropout): ModuleDict(
              (default): Dropout(p=0.01, inplace=False)
            )
            (lora_A): ModuleDict(
              (default): Linear(in_features=4096, out_features=32, bias=Fal
se)
            )
            (lora_B): ModuleDict(
              (default): Linear(in_features=32, out_features=1024, bias=Fal
se)
            )
            (lora_embedding_A): ParameterDict()
            (lora_embedding_B): ParameterDict()
          )
          (v_proj): lora.Linear4bit(
            (base_layer): Linear4bit(in_features=4096, out_features=1024, b
ias=False)
            (lora_dropout): ModuleDict(
              (default): Dropout(p=0.01, inplace=False)
            )
            (lora_A): ModuleDict(
              (default): Linear(in_features=4096, out_features=32, bias=Fal
se)
            )
            (lora_B): ModuleDict(
              (default): Linear(in_features=32, out_features=1024, bias=Fal
se)
            )
            (lora_embedding_A): ParameterDict()
            (lora_embedding_B): ParameterDict()
```

```
            )
            (o_proj): Linear4bit(in_features=4096, out_features=4096, bias=Fa
lse)
            (rotary_emb): MistralRotaryEmbedding()
          )
          (mlp): MistralMLP(
            (gate_proj): Linear4bit(in_features=4096, out_features=14336, bia
s=False)
            (up_proj): Linear4bit(in_features=4096, out_features=14336, bias=
False)
            (down_proj): Linear4bit(in_features=14336, out_features=4096, bia
s=False)
            (act_fn): SiLU()
          )
          (input_layernorm): MistralRMSNorm()
          (post_attention_layernorm): MistralRMSNorm()
        )
      )
      (norm): MistralRMSNorm()
    )
  (lm_head): Linear(in_features=4096, out_features=1, bias=False)
)
```

4. Apply LoRA to reduce memory footprint

In [150…

```
'''
pre-processing to prepare model for training
https://huggingface.co/docs/peft/v0.11.0/en/package_reference/peft_model#pef
'''

from peft import prepare_model_for_kbit_training
model_4bit.gradient_checkpointing_enable()

#model_4bit.gradient_checkpointing_enable(gradient_checkpointing_kwargs={"us
model_4bit = prepare_model_for_kbit_training(model_4bit)
```

In [151…

```
"""
For guidelines / general rule of thumb on setting `r` and `lora_alpha`:
A good heuristic is setting alpha at twice the rank's value.

Reference: https://magazine.sebastianraschka.com/p/practical-tips-for-finetu
"""

config = LoraConfig(
                        r=32, # Defines the size of the low-rank matrices.
                        lora_alpha=64, # Scaling factor for the low-rank mat
                        lora_dropout=0.01, # Dropout rate for the LoRA layer
                        target_modules=["q_proj", "k_proj", "v_proj"], # upd
                        bias="none", # Whether to include biases in the LoRA
                        task_type="CAUSAL_LM", # Indicates the type of task
                    )
```

`QLoRA` compute-memory Trade-offs: One can save ~33% of GPU memory when using QLoRA. However, this comes at a 39% increased training runtime caused by the additional quantization and dequantization of the pretrained model weights in QLoRA.

In [152…

```
peft_model = get_peft_model(model_4bit, config)
peft_model
```

Out[152…

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): MistralForCausalLM(
      (model): MistralModel(
        (embed_tokens): Embedding(32000, 4096)
        (layers): ModuleList(
          (0-31): 32 x MistralDecoderLayer(
            (self_attn): MistralSdpaAttention(
              (q_proj): lora.Linear4bit(
                (base_layer): Linear4bit(in_features=4096, out_features=409
6, bias=False)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.01, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=4096, out_features=32, bias
=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=32, out_features=4096, bias
=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
              )
              (k_proj): lora.Linear4bit(
                (base_layer): Linear4bit(in_features=4096, out_features=102
4, bias=False)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.01, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=4096, out_features=32, bias
=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=32, out_features=1024, bias
=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
              )
              (v_proj): lora.Linear4bit(
                (base_layer): Linear4bit(in_features=4096, out_features=102
4, bias=False)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.01, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=4096, out_features=32, bias
=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=32, out_features=1024, bias
=False)
                )
```

```
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                )
                (o_proj): Linear4bit(in_features=4096, out_features=4096, bia
s=False)
                (rotary_emb): MistralRotaryEmbedding()
              )
              (mlp): MistralMLP(
                (gate_proj): Linear4bit(in_features=4096, out_features=14336,
bias=False)
                (up_proj): Linear4bit(in_features=4096, out_features=14336, b
ias=False)
                (down_proj): Linear4bit(in_features=14336, out_features=4096,
bias=False)
                (act_fn): SiLU()
              )
              (input_layernorm): MistralRMSNorm()
              (post_attention_layernorm): MistralRMSNorm()
            )
          )
          (norm): MistralRMSNorm()
        )
        (lm_head): Linear(in_features=4096, out_features=1, bias=False)
      )
    )
)
```

In [153…

```
peft_model.print_trainable_parameters()
```

```
trainable params: 18,874,368 || all params: 7,129,538,560 || trainable%: 0.2
647
```

Notice, the drop in % of trainable parameters from `13.07%` to `0.26%`. `LoRa` allows you to fine-tune large language models using a much smaller set of training parameters while preserving the performance levels typically achieved through full fine-tuning.

In [154…

```
# Move the PEFT model to the device
peft_model = peft_model.to(device)
peft_model.device
```

Out[154…

```
device(type='cuda', index=0)
```

In [87]:

```
peft_model.config.__dict__
```

Out[87]:

```
{'vocab_size': 32000,
 'max_position_embeddings': 32768,
 'hidden_size': 4096,
 'intermediate_size': 14336,
 'num_hidden_layers': 32,
 'num_attention_heads': 32,
 'sliding_window': 4096,
 'num_key_value_heads': 8,
 'hidden_act': 'silu',
 'initializer_range': 0.02,
 'rms_norm_eps': 1e-05,
 'use_cache': True,
 'rope_theta': 10000.0,
 'attention_dropout': 0.0,
 'return_dict': True,
 'output_hidden_states': False,
 'output_attentions': False,
 'torchscript': False,
 'torch_dtype': torch.bfloat16,
 'use_bfloat16': False,
 'tf_legacy_loss': False,
 'pruned_heads': {},
 'tie_word_embeddings': False,
 'chunk_size_feed_forward': 0,
 'is_encoder_decoder': False,
 'is_decoder': False,
 'cross_attention_hidden_size': None,
 'add_cross_attention': False,
 'tie_encoder_decoder': False,
 'max_length': 20,
 'min_length': 0,
 'do_sample': False,
 'early_stopping': False,
 'num_beams': 1,
 'num_beam_groups': 1,
 'diversity_penalty': 0.0,
 'temperature': 1.0,
 'top_k': 50,
 'top_p': 1.0,
 'typical_p': 1.0,
 'repetition_penalty': 1.0,
 'length_penalty': 1.0,
 'no_repeat_ngram_size': 0,
 'encoder_no_repeat_ngram_size': 0,
 'bad_words_ids': None,
 'num_return_sequences': 1,
 'output_scores': False,
 'return_dict_in_generate': False,
 'forced_bos_token_id': None,
 'forced_eos_token_id': None,
 'remove_invalid_values': False,
 'exponential_decay_length_penalty': None,
 'suppress_tokens': None,
 'begin_suppress_tokens': None,
 'architectures': ['MistralForCausalLM'],
 'finetuning_task': None,
```

```
 'id2label': {0: 'LABEL_0', 1: 'LABEL_1'},
 'label2id': {'LABEL_0': 0, 'LABEL_1': 1},
 'tokenizer_class': None,
 'prefix': None,
 'bos_token_id': 1,
 'pad_token_id': None,
 'eos_token_id': 2,
 'sep_token_id': None,
 'decoder_start_token_id': None,
 'task_specific_params': None,
 'problem_type': None,
 '_name_or_path': 'mistralai/Mistral-7B-v0.1',
 '_commit_hash': '26bca36bde8333b5d7f72e9ed20ccda6a618af24',
 '_attn_implementation_internal': 'sdpa',
 'transformers_version': '4.34.0.dev0',
 'model_type': 'mistral',
 'quantization_config': BitsAndBytesConfig {
   "_load_in_4bit": true,
   "_load_in_8bit": false,
   "bnb_4bit_compute_dtype": "bfloat16",
   "bnb_4bit_quant_storage": "bfloat16",
   "bnb_4bit_quant_type": "nf4",
   "bnb_4bit_use_double_quant": true,
   "llm_int8_enable_fp32_cpu_offload": false,
   "llm_int8_has_fp16_weight": false,
   "llm_int8_skip_modules": null,
   "llm_int8_threshold": 6.0,
   "load_in_4bit": true,
   "load_in_8bit": false,
   "quant_method": "bitsandbytes"
 },
 '_pre_quantization_dtype': torch.float16}
```

5. Let's test the model

In [155…

```
# Test
inputs = tokenizer("Less: Eletricity", return_tensors="pt").to(0)

# Convert input tensors to the dtype expected by the model
#inputs = {key: tensor.to(torch.float32) for key, tensor in inputs.items()}
inputs
```

Out[155…

```
{'input_ids': tensor([[    1, 11385, 28747, 12450,   434, 25130]], device
='cuda:0'), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1]], device='cuda:
0')}
```

In [156…

```
for key, tensor in inputs.items():
    print(f"{key}: {tensor.dtype}")
```

```
input_ids: torch.int64
attention_mask: torch.int64
```

In [90]:

```
#inputs['input_ids'] = inputs['input_ids'].to(device)
#inputs['input_ids'].device
```

In [91]:
```
#inputs['attention_mask'] = inputs['attention_mask'].to(device)
#inputs['attention_mask'].device
```

In [157…
```
# check device: cuda
peft_model.device
```

Out[157…
```
device(type='cuda', index=0)
```

In [93]:
```
peft_model.hf_device_map
```

Out[93]:
```
{'': device(type='cuda', index=0)}
```

In [94]:
```
peft_model.lm_head.weight.dtype
```

Out[94]:
```
torch.float32
```

In [158…
```
with torch.no_grad(): # No gradients accumulation during inference
    outputs = peft_model(**inputs)

print("Outputs:\n", outputs.logits.shape)
print("Outputs Logits:\n", outputs.logits)
```

```
Outputs:
 torch.Size([1, 6, 1])
Outputs Logits:
 tensor([[[-0.0126],
         [-4.3610],
         [-3.1300],
         [-8.8063],
         [-6.1398],
         [-3.6898]]], device='cuda:0')
```

shape indicates:

1. Batch Size (1)
2. Sequence Length: # tokens
3. Binary classification, 2 as the output.

In this modified output layer, the model is not predicting the next token in the sequence as it would in language modeling but rather classifying each token in the input sequence into one of two classes.

In [159…
```
print("Last output token:", outputs.logits[:, -1, :])
```

```
Last output token: tensor([[-3.6898]], device='cuda:0')
```

Convert the outputs (logits) into probability scores via the `sigmoid` function and then obtain the index position of the largest probability value via the `argmax` function

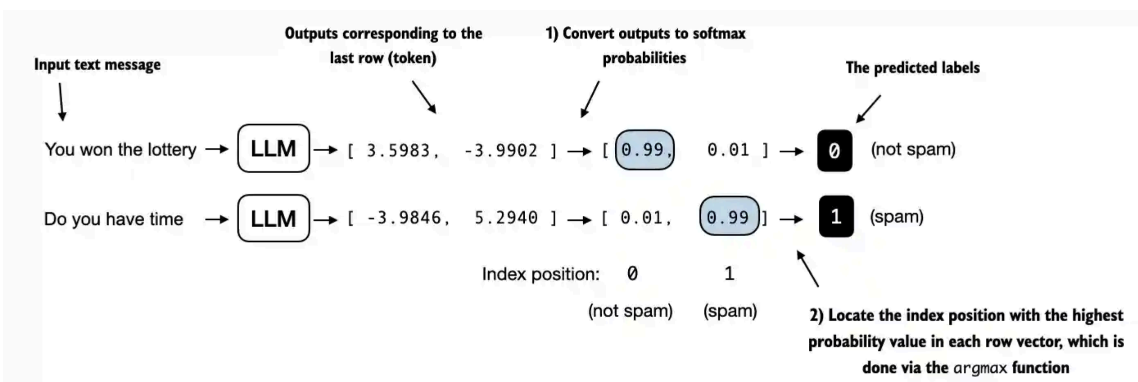
In [160…
```
probs = torch.sigmoid(outputs.logits[:, -1, :])
probs
```

```
Out[160…  tensor([[0.0244]], device='cuda:0')
```

```
In [161…  def get_label(probs):

              # single tensor

              if probs.shape[0] == 1:

                  # sigmoid probs represents the probability of belonging to 1
                  if probs.item() > 0.5:
                      return 1
                  else:
                      return 0

              # multiple tensor - batch
              if probs.shape[0] > 1:

                  preds = (probs > 0.5).float()
                  return preds.squeeze()
```

```
In [162…  predicted_labels = get_label(probs)
          print(f"Class label: {predicted_labels}")
```

```
Class label: 0
```



6. Classification accuracy without Fine-tuning

```
In [6]:  # for i, (input_batch, target_batch) in enumerate(train_loader):

         #     print(i)
         #     print('input batch:', input_batch)
         #     print('target labels:', target_batch)

         #     if i == 4:
         #         break;
```

```
In [101…  def calc_accuracy(data_loader, model, device):

              model.eval()

              correct_preds, num_examples = 0, 0
```

```
    for i, (input_batch, target_batch) in enumerate(data_loader):

        input_batch, target_labels = input_batch.to(device), target_batch.to

        with torch.no_grad():
            outputs = model(input_batch)

        """
        Convert raw logits to a probability distribution with softmax

        `outputs.logits[:, -1, :]`: Apply sigmoid to last token in each sequ
        `dim=-1`: Apply along a specific dimension. For classification, rang
        """
        probs = torch.sigmoid(outputs.logits[:, -1, :])

        """
        Locate index with highest probability value using argmax if using sc
        """
        predicted_labels = get_label(probs)

        num_examples += input_batch.size(0)

        correct_preds += torch.eq(predicted_labels, target_labels).sum().ite

        if i == 5:
            break;

    print('num examples', num_examples)
    print('correct preds', correct_preds)

    return round(correct_preds / num_examples,2)
```

In [104…

```
train_accuracy = calc_accuracy(train_loader, peft_model, device)
print(f"Training accuracy: {train_accuracy*100:.2f}%")
test_accuracy = calc_accuracy(test_loader, peft_model, device)
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

```
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
num examples 150
correct preds 77
Training accuracy: 51.00%
```

```
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
num examples 150
correct preds 75
Test accuracy: 50.00%
```

7. Fine-tuning

Define a custom class

In [163…

```
class BinaryClassification(nn.Module):

    def __init__(self, base_model):
        super().__init__()
        self.base_model = base_model
        #self.dropout = nn.Dropout(0.05)
        #self.classifier = nn.Linear(hidden_size, 1)
        #self.relu = nn.ReLU()
        #self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        outputs = self.base_model(x)
        #dropout_output = self.dropout(outputs.logits[:, -1, :])
        #relu_output = self.relu(dropout_output[:, -1, :])
        """
        Apply sigmoid to logits to get probabilities
        sigmoid(x) = 1 / (1 + exp(-x))
        """
        #probs = self.sigmoid(relu_output)

        #print('forward probs', probs)
        return outputs.logits[:, -1, :]
```

In [7]:

```
# Test the custom BinaryClassificationHead class
model_init = BinaryClassification(peft_model)
#model_init
```

Loss function and optimizer

- optimizer:

`learning rate` controls the step size at each iteration while moving toward a minimum of the loss function.

`weight decay` is (L2 penalty). Adds a small penalty for larger weights, which can help prevent overfitting

- Learning Rate Scheduler:

`ReduceLROnPlateau` is used to reduce the learning rate when the loss has stopped improving.

`factor=0.1` means the learning rate will be reduced by a factor of 10.

`patience=1` means the scheduler will wait for 1 epochs before reducing the learning rate if there is no improvement.

- In `BCELoss()` we pass the probabilities directly. The `0.5` threshold is only required for converting the probabilities into class labels when you want to calculate the accuracy and return the predictions.

  The `BCEWithLogitsLoss()` (Binary Cross Entropy with Logits Loss) function combines a Sigmoid layer and the BCELoss in one single class. This version is more numerically stable than using a plain Sigmoid followed by a BCELoss for several reasons:

  - It uses the log-sum-exp trick for numerical stability.
  - It combines the operations internally which can be optimized by the framework.

In [166…

```
#criterion = torch.nn.BCELoss()

"""
Given that we observe vanishing gradient problem with sigmoid; let's remove
"""
criterion = torch.nn.BCEWithLogitsLoss(reduction='mean')
optimizer = torch.optim.AdamW(model_init.parameters(), lr=0.0001, eps=1e-08)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=1,
```

In [167…

```
def calc_loss_batch(model, input_batch, target_batch):

    output = model(input_batch)

    # Reshape target to match the shape of probs
    target_batch = target_batch.unsqueeze(1)

    if output.shape != target_batch.shape:
        print('model output shape', output.shape)
        print('target labels shape', target_batch.shape)
        raise Exception("Shape mismatch between input logits and target labe

    # Logits of last output token
    loss = criterion(output, target_batch)

    return loss, output
```

Training Loop

In [168…

```
val_losses = []
val_accuracies = []

def validate(model, data_loader):

    # Set the model to evaluation mode
    model.eval()
    correct_preds = 0
    num_examples = 0
    total_val_loss = 0

    with torch.no_grad():  # No need to track gradients during validation
        for i, (input_batch, target_batch) in enumerate(data_loader):
            input_batch, target_labels = input_batch.to(device), target_batc

            print('iteration {}'.format(i))

            # calculate loss
            val_loss, output = calc_loss_batch(model, input_batch, target_la
            # Record loss per batch
            total_val_loss += val_loss.item()
            val_losses.append(val_loss)

            if i%50==0:
                print(f'iteration: {i}, validation loss:  {val_loss.item()}'

            # Calculate Accuracy
            predicted_labels = get_label(output)
            correct_preds += torch.eq(predicted_labels, target_labels).sum()
            num_examples += input_batch.size(0)

    val_accuracy = correct_preds / num_examples

    # Record Validation Accuracy per epoch
    val_accuracies.append(val_accuracy)
    avg_val_loss = total_val_loss / len(data_loader)

    return val_accuracy, avg_val_loss
```

In [171…

```
train_losses = []
train_accuracies = []
train_batch_idx = []
train_epochs = []

def train(epoch):

    # set the model to training mode
    model_init.train()
    correct_preds = 0
    num_examples = 0

    best_val_loss = float('inf')
    best_model_path = ''
```

```
    for i, (input_batch, target_batch) in enumerate(train_loader):

        input_batch, target_labels = input_batch.to(device), target_batch.to

        optimizer.zero_grad() # Reset loss gradients from previous batch

        print('iteration {}'.format(i))

        # calculate loss
        loss, output = calc_loss_batch(model_init, input_batch, target_label

        if i%50==0:
            print(f'epoch: {epoch}, training loss:  {loss.item()}')

        loss.backward() # Backpropogation - calculate loss gradients

        # print gradients
#         for name, param in model_init.named_parameters():
#             if param.grad is not None:
#                 print(f'Gradient for {name}: {param.grad.norm()}')

        # Clip gradients
        max_norm = 1
        torch.nn.utils.clip_grad_norm_(model_init.parameters(), max_norm)

        # update model weights using loss gradients
        optimizer.step()

        # Calculate Accuracy
        predicted_labels = get_label(output)
        correct_preds += torch.eq(predicted_labels, target_labels).sum().ite
        num_examples += input_batch.size(0)

        # Record Training Loss per batch
        train_losses.append(loss.item())
        train_batch_idx.append(i)

        # early stopping
        #if i == 2:
        #    break;

    # Calculate and Record Accuracy per epoch
    train_epochs.append(epoch)
    train_accuracy = correct_preds / num_examples
    train_accuracies.append(train_accuracy)

    avg_train_loss = sum(train_losses)/len(train_loader)

    print(f"Epoch {epoch} - Avg. Training Loss: {avg_train_loss:.2f}, Traini

    # Record validation loss and accuracy
    val_accuracy, avg_val_loss  = validate(model_init, val_loader)

    print(f"Epoch {epoch} - Avg. Validation Loss: {avg_val_loss:.2f}, Valida
```

```python
    # Step the scheduler after epoch completion
    scheduler.step(loss)

    # Save best model
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        torch.save(model_init.state_dict(), 'classifier.pth')
        print(f"Saved best model with validation loss: {best_val_loss:.2f}")

    return model_init
```

In [ ]:
```python
# Start the Training run
start_time = time.time()

EPOCHS = 4

for epoch in range(1, EPOCHS+1):
    print('Epoch:', epoch)
    model = train(epoch)

end_time = time.time()

exec_time_mins = (end_time - start_time)/60
print(f"Training completed in {exec_time_mins:.2f} minutes.")
```

```
Epoch: 1
```

```
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
`use_cache=True` is incompatible with gradient checkpointing. Setting `use_c
ache=False`...
```

```
iteration 0
```

```
/home/ec2-user/anaconda3/envs/python3/lib/python3.10/site-packages/torch/uti
ls/checkpoint.py:464: UserWarning: torch.utils.checkpoint: the use_reentrant
parameter should be passed explicitly. In version 2.4 we will raise an excep
tion if use_reentrant is not passed. use_reentrant=False is recommended, but
if you need to preserve the current default behavior, you can pass use_reent
rant=True. Refer to docs for more details on the differences between the two
variants.
  warnings.warn(
```

Plot loss curves

In [210…

```
# plt.figure(figsize=(10, 5))
# plt.plot(np.array(train_losses), label='Train Loss')
# plt.plot(np.array(val_losses), label='Validation Loss')
# plt.xlabel('Batch')
# plt.ylabel('Loss')
# plt.title('Loss curves')
# plt.legend()
# plt.show()
```

Evaluate on test data loader

In [211…

```
model_path = os.getcwd() + '/classifier.pth'

model = BinaryClassification(peft_model)
model.load_state_dict(torch.load(model_path))
model.to(device)
```

```
Out[211… BinaryClassification(
  (base_model): PeftModelForCausalLM(
    (base_model): LoraModel(
      (model): MistralForCausalLM(
        (model): MistralModel(
          (embed_tokens): Embedding(32000, 4096)
          (layers): ModuleList(
            (0-31): 32 x MistralDecoderLayer(
              (self_attn): MistralSdpaAttention(
                (q_proj): lora.Linear4bit(
                  (base_layer): Linear4bit(in_features=4096, out_features=4
096, bias=False)
                  (lora_dropout): ModuleDict(
                    (default): Dropout(p=0.01, inplace=False)
                  )
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=4096, out_features=32, bi
as=False)
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=32, out_features=4096, bi
as=False)
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                )
                (k_proj): lora.Linear4bit(
                  (base_layer): Linear4bit(in_features=4096, out_features=1
024, bias=False)
                  (lora_dropout): ModuleDict(
                    (default): Dropout(p=0.01, inplace=False)
                  )
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=4096, out_features=32, bi
as=False)
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=32, out_features=1024, bi
as=False)
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                )
                (v_proj): lora.Linear4bit(
                  (base_layer): Linear4bit(in_features=4096, out_features=1
024, bias=False)
                  (lora_dropout): ModuleDict(
                    (default): Dropout(p=0.01, inplace=False)
                  )
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=4096, out_features=32, bi
as=False)
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=32, out_features=1024, bi
as=False)
```

```
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                )
                (o_proj): Linear4bit(in_features=4096, out_features=4096, b
ias=False)
                (rotary_emb): MistralRotaryEmbedding()
              )
              (mlp): MistralMLP(
                (gate_proj): Linear4bit(in_features=4096, out_features=1433
6, bias=False)
                (up_proj): Linear4bit(in_features=4096, out_features=14336,
bias=False)
                (down_proj): Linear4bit(in_features=14336, out_features=409
6, bias=False)
                (act_fn): SiLU()
              )
              (input_layernorm): MistralRMSNorm()
              (post_attention_layernorm): MistralRMSNorm()
            )
          )
          (norm): MistralRMSNorm()
        )
        (lm_head): Linear(in_features=4096, out_features=1, bias=False)
      )
    )
  )
)
```

In [212…

```
test_accuracy, avg_test_loss = validate(model, test_loader)
print(f"Avg. test loss: {avg_test_loss:.2f}, Test Accuracy: {test_accuracy*1
```

```
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
huggingface/tokenizers: The current process just got forked, after paralleli
sm has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tr
ue | false)
```

```
iteration 0
iteration: 0, validation loss:  9.630246495362371e-06
iteration 1
iteration 2
iteration 3
iteration 4
iteration 5
iteration 6
iteration 7
iteration 8
iteration 9
iteration 10
iteration 11
iteration 12
iteration 13
iteration 14
iteration 15
iteration 16
iteration 17
iteration 18
iteration 19
iteration 20
iteration 21
iteration 22
iteration 23
iteration 24
iteration 25
iteration 26
iteration 27
iteration 28
iteration 29
Avg. test loss: 0.01, Test Accuracy: 99.73%
```

Some notes and observations from Training Loop:

1.

- When the `probs tensor` contains all `1s`, it means that the model is predicting the positive class with very high confidence for all the samples in the batch.
- In the `BinaryClassification` class, the forward method takes the input `x` and passes it through the `base_model` to obtain the outputs. The `outputs.logits` tensor contains the `unnormalized log probabilities (logits)` for each token in the sequence.
- The line `probs = self.sigmoid(outputs.logits[:, -1, :])` applies the `sigmoid` activation function to the logits of the last token in each sequence. The sigmoid function maps the logits to probabilities between `0` and `1`. If probs is all `1s`, it suggests that the model is extremely confident in predicting the positive class for all the samples.

This could happen due to several reasons:

`Overfitting:` The model may have overfit to the training data, learning to predict the positive class with high confidence for all the samples, even if they don't truly belong to the positive class. Overfitting can occur when the model is too complex relative to the amount of training data or when the training process continues for too long.

`Imbalanced dataset:` If the training data is heavily imbalanced, with a significantly higher number of positive samples compared to negative samples, the model may learn to predict the positive class by default. This can lead to high probabilities for the positive class, even for negative samples.

`Insufficient regularization:` Regularization techniques, such as dropout or weight decay, help prevent overfitting by introducing noise or constraints during training. If the model lacks proper regularization, it may become overconfident in its predictions.

2.

`Vanishing gradients problem commonly observed with sigmoid activation`

```
- The vanishing gradient problem is a significant issue when
using the sigmoid activation function in deep neural networks
due to the multiplicative nature of backpropagation and the
small derivative values of the sigmoid function, especially
in the saturated regions.

- The intuition behind the saturated region is that when the
output of the sigmoid function becomes saturated (i.e.,
remains constant), it means that the input to the sigmoid
function is either a large positive or negative value. In
such cases, even if the weights are updated, the output of
the sigmoid function will remain almost the same, as it is
already saturated at either 0 or 1.
```

3.

`Interpreting gradients`

```
- Gradient Magnitude: Large gradients (e.g., > 1) might
indicate unstable learning or potential exploding gradients.
Very small gradients (e.g., < 1e-5) could suggest vanishing
gradients or that the model has converged. Moderate, non-zero
gradients typically indicate active learning. Gradients
should generally be non-zero but not too large (e.g., between
1e-5 and 1).


- Gradient Direction:
    - Positive gradients mean the loss increases as the
parameter increases.
```

```
    - Negative gradients mean the loss decreases as the
parameter increases.


- Gradient Variability:
    - If gradients vary significantly between batches, it
might indicate high variance in your data or unstable
learning.
    - Consistent gradients across batches suggest stable
learning.

- LoRA Architecture: LoRA adds small, trainable rank
decomposition matrices to the original model layers,
typically attention layers. The original model parameters
remain frozen, and only these new LoRA parameters are
trained.

- Gradient Flow: Gradients will only flow through the LoRA
parameters, not the original model parameters. This means
you'll only see non-zero gradients for these new parameters.
```

Deploy to AWS SageMaker

- https://sagemaker-examples.readthedocs.io/en/latest/frameworks/pytorch/get_started_mnist_deploy.html
- https://github.com/aws/amazon-sagemaker-examples/blob/main/sagemaker-python-sdk/pytorch_batch_inference/sagemaker_batch_inference_torchserve.ipynb
- https://sagemaker.readthedocs.io/en/stable/frameworks/pytorch/using_pytorch.html#de pytorch-models
- https://github.com/aws/amazon-sagemaker-examples/blob/main/sagemaker_batch_transform/pytorch_mnist_batch_transform/pyto mnist-batch-transform.ipynb

Instance type: `g4dn.xlarge`, 1 NVIDIA T4 Tensor Core GPU, 4 vCPUs: https://aws.amazon.com/ec2/instance-types/g4/

In [ ]:
```
import boto3
import sagemaker
from sagemaker.pytorch import PyTorchModel
from sagemaker.serializers import JSONSerializer
from sagemaker.deserializers import JSONDeserialize
```

In [ ]:
```
# Step 2: Upload the model to S3
def upload_to_s3(bucket_name, model_name):
    s3_client = boto3.client('s3')
    s3_client.upload_file('model.pth', bucket_name, f'{model_name}/model.pth
    print(f"Model uploaded to s3://{bucket_name}/{model_name}/model.pth")
```

```python
# Step 3: Create a SageMaker model and deploy to an endpoint
def deploy_to_sagemaker(bucket_name, model_name, role_arn):
    sagemaker_session = sagemaker.Session()

    # Create PyTorch model
    pytorch_model = PyTorchModel(
        model_data=f's3://{bucket_name}/{model_name}/model.pth',
        role=role_arn,
        entry_point='inference.py',  # create this file
        framework_version='1.8.1',  # Adjust as needed
        py_version='py3',
        predictor_cls=sagemaker.predictor.Predictor,
        serializer=JSONSerializer(),
        deserializer=JSONDeserializer()
    )

    # Deploy the model to an endpoint
    predictor = pytorch_model.deploy(
        initial_instance_count=1,
        instance_type='ml.m5.large',  # Adjust as needed
        endpoint_name=f'{model_name}-endpoint'
    )

    print(f"Model deployed to endpoint: {predictor.endpoint_name}")
    return predictor
```

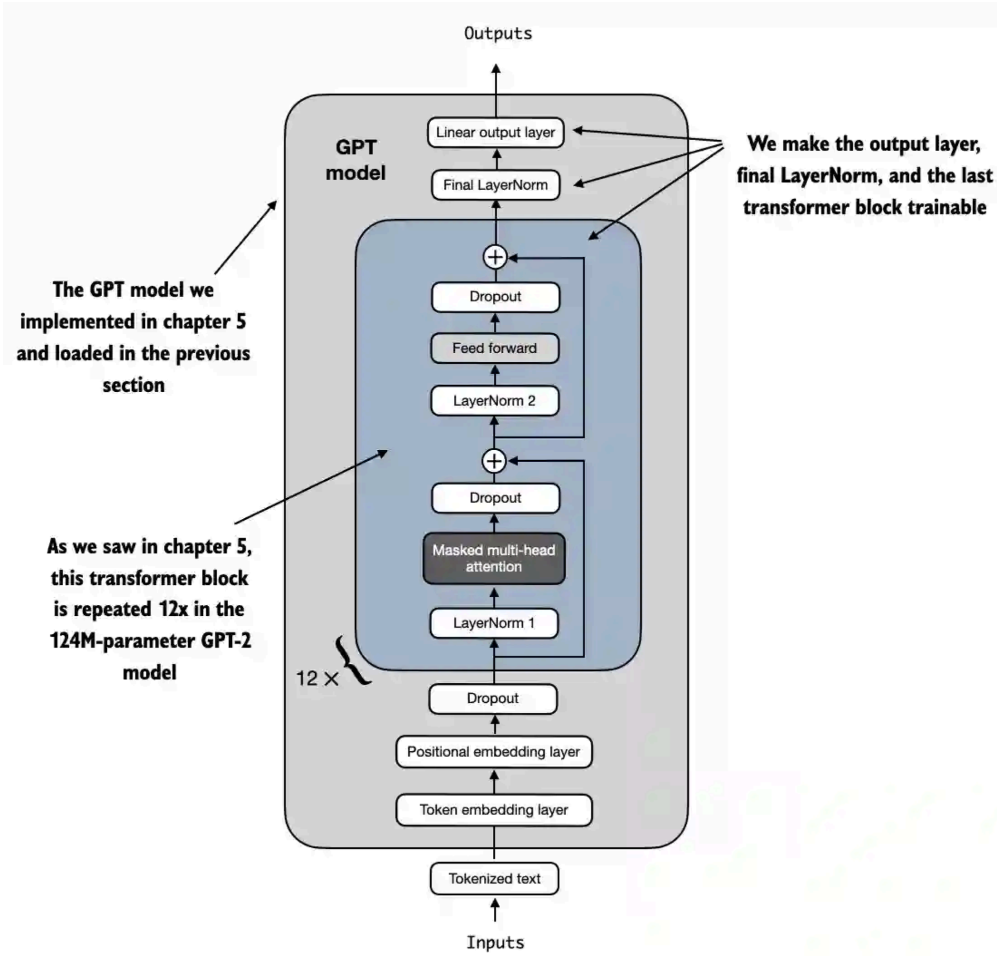
In [ ]:

```python
# Main execution
if __name__ == "__main__":
    package_model()

    bucket_name = 'your-s3-bucket-name'
    model_name = 'your-model-name'
    role_arn = 'your-sagemaker-role-arn'

    upload_to_s3(bucket_name, model_name)
    predictor = deploy_to_sagemaker(bucket_name, model_name, role_arn)

    # Test the endpoint
    test_data = {"input": [1.0, 2.0, 3.0, 4.0]}  # Adjust based on your mode
    result = predictor.predict(test_data)
    print("Prediction result:", result)
```

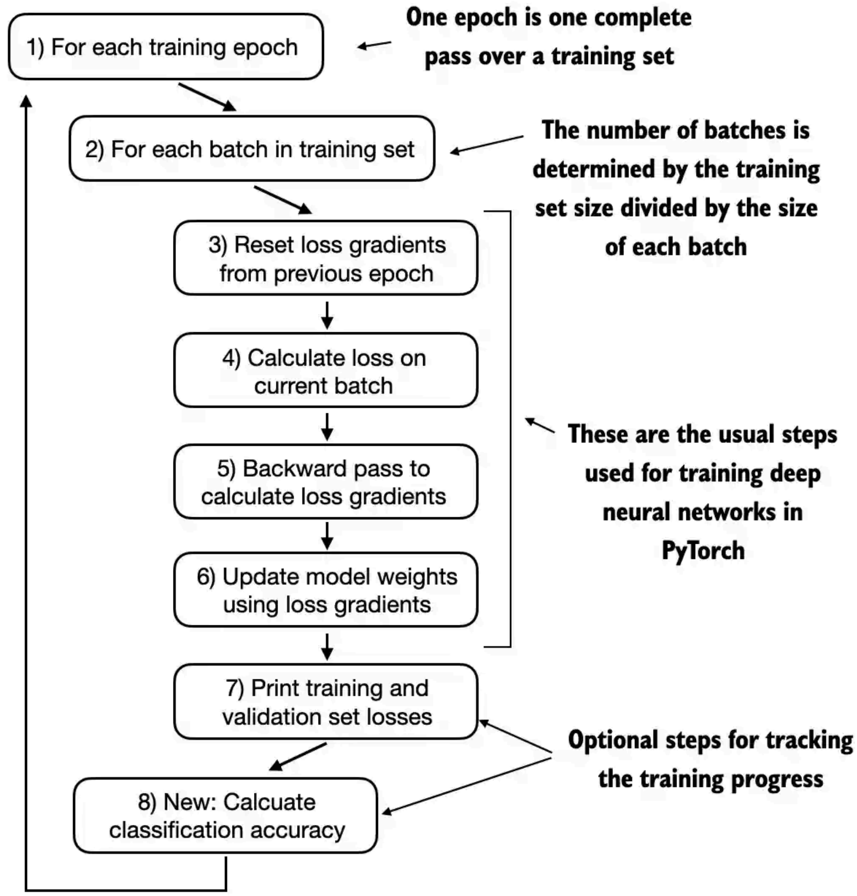
In [ ]:

While, finetuning only the last layer could also be considered a parameter-efficient finetuning technique, techniques such as prefix tuning, adapters, and low-rank adaptation, all of which “modify” multiple layers, namely `attention layers` and `feed-forward` to achieve much better predictive performance (at a low cost).

So, we can also make the last transformer block and the final `LayerNorm` module connecting the last transformer block to the output layer trainable.



4. Fine-tuning Training Loop



Before explaining the loss calculation, let's have a brief look at how the model outputs are turned into class labels