**Implementing LoRA from Scratch**

How to implement LoRA from scratch and some practical tips

[[Martin Dittgen](https://medium.com/@martin.p.dittgen?source=post_page-----20f838b046f1--------------------------------)](https://medium.com/@martin.p.dittgen?source=post_page-----20f838b046f1--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----20f838b046f1--------------------------------)](https://towardsdatascience.com/?source=post_page-----20f838b046f1--------------------------------)

[Martin Dittgen](https://medium.com/@martin.p.dittgen?source=post_page-----20f838b046f1--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F733d27d88c18&operation=register&redirect=https%3A%2F%2Ftowardsdatascience.com%2Fimplementing-lora-from-scratch-20f838b046f1&user=Martin+Dittgen&userId=733d27d88c18&source=post_page-733d27d88c18----20f838b046f1---------------------post_header-----------)

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----20f838b046f1--------------------------------)

·

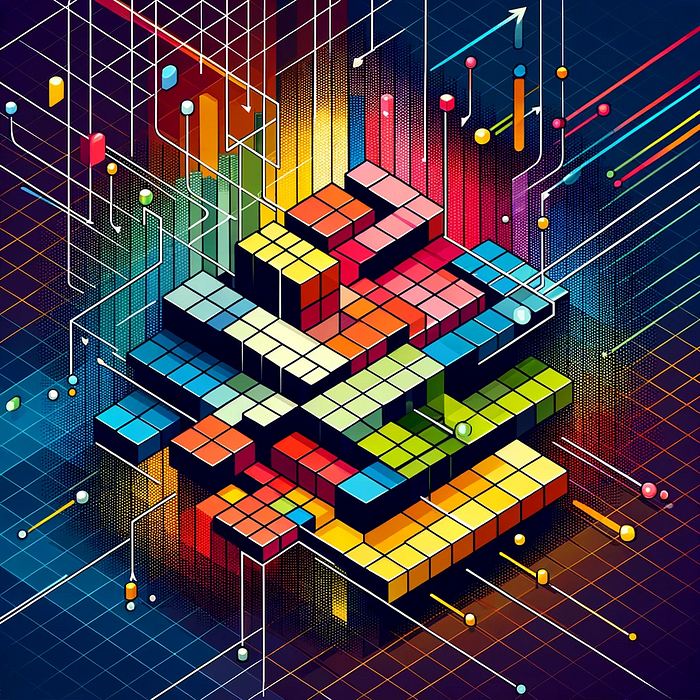
17 min read

·

6 days ago

410

3



*Abstract artistic representation of LoRA, created by DALLE*

In this blog post, I will show you how to implement LoRA from scratch.

LoRA, an acronym for *Low-Rank Adaptation* or *Low-Rank Adaptors*, offers an efficient and lightweight method for fine-tuning pre-existing language models. This includes masked language models like *BERT* and *RoBERTa*, as well as causal (or chatbot) models such as *GPT*, *Llama*, and *Mistral*.

One of the main advantages of low-rank adaptors is their efficiency. By utilizing fewer parameters, LoRAs significantly lower computational complexity and memory usage. This allows us to train large models on consumer-grade GPUs and effortlessly distribute our compact (in terms of megabytes) LoRAs to others.

Additionally, LoRAs can improve generalization performance. By constraining the model complexity, they help prevent overfitting, especially in scenarios with limited training data. This results in more resilient models that excel with new, unseen data, or at the very least, retain the knowledge from their initial training tasks.

Furthermore, low-rank adaptors can be seamlessly integrated into existing neural network architectures. This integration allows for fine-tuning and adaptation of pre-trained models with minimal additional training cost, making them highly suitable for transfer learning applications.

We’ll start by delving into how LoRA functions, then I’ll demonstrate how you can develop it from scratch for a *RoBERTa* model, followed by benchmarking our implementation using the *GLUE* and *SQuAD* benchmarks along with a discussion on general tips and improvements.

**How LoRA works**

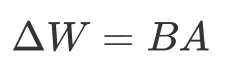
The basic idea of LoRA is to keep the pre-trained matrices (i.e. parameters of the original model) frozen (i.e. in a fixed state) and only add a small delta to the original matrix, which has fewer parameters than the original matrix.

For example consider the matrix *W*, which could either be the parameters of a fully connected layer or one of the matrices from the self-attention mechanism of a transformer:



Obviously, if ***W****-orig* had dimensions *n×m* and we would just initialize a new delta matrix with the same dimensions to fine-tune on we would have gained nothing; quite to the contrary we would have doubled the parameters.

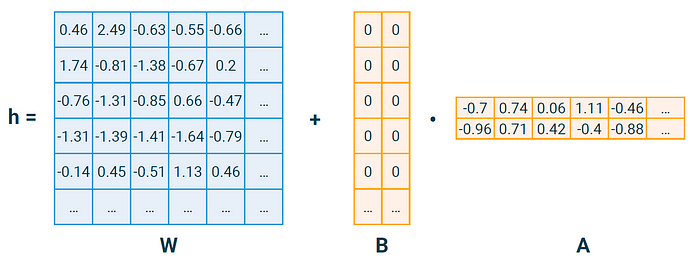
The trick is to make **ΔW** less *“dimensional”* than the original matrix, by constructing it via matrix multiplication from lower dimensional matrices *B* and *A*.



Where we first define a rank *r*, to be significantly smaller than the base matrix dimensions *r≪n*and *r≪m.* Then matrix ***B***is *n×r* and matrix***A*** is *r×m*. Multiplying them yields a matrix with the same dimensions of ***W***, but constructed from a much lower parameter count.

Obviously we want our delta to be zero at the start of the training, such that the fine-tuning starts just like the original model. Therefore ***B*** is often initialized as all zeros and ***A*** is initialized as random (usually normally distributed) values.

For example, this might look like this:

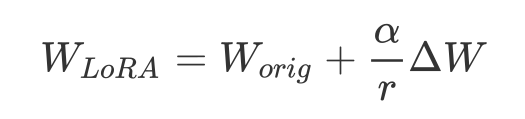


*A figure with an example of how LoRA might look for an actual matrix*

Imagine a situation where our base-dimensionality is 1024 and we chose a LoRA rank *r* of 4 then:

* ***W*** has 1024 \* 1024 ≈ 1 Million parameters
* ***A*** & ***B*** have r \* 1024 = 4 \* 1024 ≈ 4k parameters each, yielding 8k in total
* Thus we only have to train 0.8% of the parameters to update our matrix with LoRA

A little aside, in the LoRA paper they weigh the delta matrix with an alpha parameter:



If you just set your *α* to the first *r* you experiment with and fine-tune the learning rate you can generally change the *r* parameter later without having to fine-tune the learning rate again (at least approximately). While we can overlook this detail in our implementation, it’s a common feature in many other LoRA libraries, such as Hugging Face’s PEFT.

**Implementing LoRA**

For our implementation we want to stick closely to the original LoRA paper. There they tested which matrices of a transformer you actually have to replace. They found that, when comparing different strategies on a GPT-3 fine-tune task, it was sufficient to only adapt the self-attention mechanism’s query and value vectors.

Note that many people ignore this assessment nowadays and allow each matrix to be fine-tuned, no matter the task or model (see QLoRA paper).

Our implementation here will be done in PyTorch, but should be easily adaptable to different frameworks.

For this blogpost, I simplified the code a bit, such that it should be easier to read, while still showing the essential elements. The full code and some trained LoRA weights can be found here: <https://github.com/Montinger/Transformer-Workbench>.

**Reimplementing the self-attention model**

The model we wish to adapt is the RoBERTa model from Huggingface. The most straightforward way is to just re-wrap the original self-attention mechanism RobertaSelfAttention. The new class LoraRobertaSelfAttention will then initialize the LoRA matrices. All the B matrices will be initialized with zeros and all the A matrices with random numbers from a normal distribution.

class LoraRobertaSelfAttention(RobertaSelfAttention):  
 """  
 Extends RobertaSelfAttention with LoRA (Low-Rank Adaptation) matrices.  
 LoRA enhances efficiency by only updating the query and value matrices.  
 This class adds LoRA matrices and applies LoRA logic in the forward method.  
  
 Parameters:  
 - r (int): Rank for LoRA matrices.  
 - config: Configuration of the Roberta Model.  
 """  
 def \_\_init\_\_(self, r=8, \*args, \*\*kwargs):  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
 d = self.all\_head\_size  
  
 # Initialize LoRA matrices for query and value  
 self.lora\_query\_matrix\_B = nn.Parameter(torch.zeros(d, r))  
 self.lora\_query\_matrix\_A = nn.Parameter(torch.randn(r, d))  
 self.lora\_value\_matrix\_B = nn.Parameter(torch.zeros(d, r))  
 self.lora\_value\_matrix\_A = nn.Parameter(torch.randn(r, d))

Given these matrices, we now define new class methods lora\_query and lora\_value. These calculate the ***ΔW*** matrix, i.e. ***BA,*** and add it to the original matrix, which we call from the original methods query and value.

class LoraRobertaSelfAttention(RobertaSelfAttention):  
 # ...  
  
 def lora\_query(self, x):  
 """  
 Applies LoRA to the query component. Computes a modified query output by adding   
 the LoRA adaptation to the standard query output. Requires the regular linear layer   
 to be frozen before training.  
 """  
 lora\_query\_weights = torch.matmul(self.lora\_query\_matrix\_B, self.lora\_query\_matrix\_A)  
 return self.query(x) + F.linear(x, lora\_query\_weights)  
  
 def lora\_value(self, x):  
 """  
 Applies LoRA to the value component. Computes a modified value output by adding   
 the LoRA adaptation to the standard value output. Requires the regular linear layer   
 to be frozen before training.  
 """  
 lora\_value\_weights = torch.matmul(self.lora\_value\_matrix\_B, self.lora\_value\_matrix\_A)  
 return self.value(x) + F.linear(x, lora\_value\_weights)

Now the ugly part: To use the methods we have to overwrite the original forward function of the RobertaSelfAttention. Though this is a bit hard-coded (see the discussion on improvements later), it is quite simple. First, we copy the original forward code from <https://github.com/huggingface/transformers/blob/main/src/transformers/models/roberta/modeling_roberta.py>. Second we replace each call to query by lora\_query and each call to value to lora\_value. The function then looks like this:

class LoraRobertaSelfAttention(RobertaSelfAttention):  
 # ...  
 def forward(self, hidden\_states, \*args, \*\*kwargs):  
 """Copied from  
https://github.com/huggingface/transformers/blob/main/src/transformers/models/roberta/modeling\_roberta.py  
 but replaced the query and value calls with calls to the  
 lora\_query and lora\_value functions.  
 We will just sketch of how to adjust this here.   
 Change every call to self.value and self.query in the actual version.  
 """  
 # original code for query:  
 ## mixed\_query\_layer = self.query(hidden\_states)  
 # updated query for LoRA:  
 mixed\_query\_layer = self.lora\_query(hidden\_states)  
  
 # The key has no LoRA, thus leave these calls unchanged  
 key\_layer = self.transpose\_for\_scores(self.key(hidden\_states))  
  
 # original code for value:  
 ## value\_layer = self.transpose\_for\_scores(self.value(hidden\_states))  
 # updated value for LoRA:  
 value\_layer = self.transpose\_for\_scores(self.lora\_value(hidden\_states))  
   
 # ... (rest of the forward code, unchanged)

Tada, there we have it: Our implementation of our LoRA-self-attention. Now the only task that remains is to swap out the attention modules in the original RoBERTa model.

**Replacing the modules**

Ok great, we have replaced the self-attention with our own implementation; but how do we get this new class into the old RoBERTa model? Essentially we have to loop over each named component of the RoBERTa model, check whether it is of the class RobertaSelfAttention, and if yes replace it by LoraRobertaSelfAttention, while making sure that the original weight matrices are retained.

In order to achieve this we will write a new wrapper function that can do this replacement. Additionally, we also want to add the functionality for fine-tuning the RoBERTa model on some actual tasks later

class LoraWrapperRoberta(nn.Module):  
 def \_\_init\_\_(self, task\_type, num\_classes=None, dropout\_rate=0.1, model\_id="roberta-large",  
 lora\_rank=8, train\_biases=True, train\_embedding=False, train\_layer\_norms=True):  
 """  
 A wrapper for RoBERTa with Low-Rank Adaptation (LoRA) for various NLP tasks.  
 - task\_type: Type of NLP task ('glue', 'squad\_v1', 'squad\_v2').  
 - num\_classes: Number of classes for classification (varies with task).  
 - dropout\_rate: Dropout rate in the model.  
 - model\_id: Pre-trained RoBERTa model ID.  
 - lora\_rank: Rank for LoRA adaptation.  
 - train\_biases, train\_embedding, train\_layer\_norms:   
 Flags whether to keep certain parameters trainable   
 after initializing LoRA.  
   
 Example:  
 model = LoraWrapperRoberta(task\_type='glue')  
 """  
 super().\_\_init\_\_()  
 # 1. Initialize the base model with parameters  
 self.model\_id = model\_id  
 self.tokenizer = RobertaTokenizer.from\_pretrained(model\_id)  
 self.model = RobertaModel.from\_pretrained(model\_id)  
 self.model\_config = self.model.config  
  
 # 2. Add the layer for the benchmark tasks  
 d\_model = self.model\_config.hidden\_size  
 self.finetune\_head\_norm = nn.LayerNorm(d\_model)  
 self.finetune\_head\_dropout = nn.Dropout(dropout\_rate)  
 self.finetune\_head\_classifier = nn.Linear(d\_model, num\_classes)  
  
 # 3. Set up the LoRA model for training  
 self.replace\_multihead\_attention()  
 self.freeze\_parameters\_except\_lora\_and\_bias()

As you can see we call two helper methods in the initialization:

1. self.replace\_multihead\_attention: This replaces the attention of all neural network parts by our previously written LoraRobertaSelfAttention
2. self.freeze\_parameters\_except\_lora\_and\_bias: This will freeze all of the main parameters for the training, such that the gradients and optimizer steps are only applied to the LoRA parameters and the other bias and layer norm parameters we want to keep trainable.

class LoraWrapperRoberta(nn.Module):  
 # ...  
  
 def replace\_multihead\_attention\_recursion(self, model):  
 """  
 Replaces RobertaSelfAttention with LoraRobertaSelfAttention in the model.  
 This method applies the replacement recursively to all sub-components.  
  
 Parameters  
 ----------  
 model : nn.Module  
 The PyTorch module or model to be modified.  
 """  
 for name, module in model.named\_children():  
 if isinstance(module, RobertaSelfAttention):  
 # Replace RobertaSelfAttention with LoraRobertaSelfAttention  
 new\_layer = LoraRobertaSelfAttention(r=self.lora\_rank, config=self.model\_config)  
 new\_layer.load\_state\_dict(module.state\_dict(), strict=False)  
 setattr(model, name, new\_layer)  
 else:  
 # Recursive call for child modules  
 self.replace\_multihead\_attention\_recursion(module)

We have to loop recursively through all the model parts, as in PyTorch parts of the network can (and in fact are for RoBERTa) packed into a separate PyTorch module.

Now we have to freeze all the parameters we don’t want to train any longer:

class LoraWrapperRoberta(nn.Module):  
 # ...  
  
 def freeze\_parameters\_except\_lora\_and\_bias(self):  
 """  
 Freezes all model parameters except for specific layers and types based on the configuration.  
 Parameters in LoRA layers, the finetune head, bias parameters, embeddings, and layer norms   
 can be set as trainable based on class settings.  
 """  
 for name, param in self.model.named\_parameters():  
 is\_trainable = (  
 "lora\_" in name or  
 "finetune\_head\_" in name or  
 (self.train\_biases and "bias" in name) or  
 (self.train\_embeddings and "embeddings" in name) or  
 (self.train\_layer\_norms and "LayerNorm" in name)  
 )  
 param.requires\_grad = is\_trainable

Additionally, we have to implement the forward methods to account for the tasks we will fine-tune on as well as two methods to save and load the LoRA weights, such that we can load the adapters of a previously trained model.

Cliffhanger: There is a way, that would have made the code much nicer and easy to generalize to other network architectures (as ours is pretty hard coded to the RoBERTa model). Can you think what this might be? You have time to ponder this question until we discuss it in the *Possible Improvements* section below. But until then: Let's test on some benchmarks if our implementation actually works.

**Benchmarking the results with GLUE and SQuAD**

Our implementation is now ready to be evaluated using the GLUE (General Language Understanding Evaluation) and SQuAD (*Stanford Question Answering* Dataset) benchmarks.

The GLUE benchmark, a suite of eight diverse NLP tasks, gauges a language model’s comprehensive understanding abilities. It includes challenges like sentiment analysis, textual entailment, and sentence similarity, offering a robust measure of a model’s linguistic adaptability and proficiency.

SQuAD, on the other hand, focuses on assessing question-answering models. It involves extracting answers from Wikipedia passages, where the model identifies the relevant text span. SQuAD v2, a more advanced version, introduces unanswerable questions, adding complexity and mirroring real-life situations where models must recognize when text lacks an answer.

Note that for the following benchmark, I did not tune any hyperparameters, did not do multiple runes (especially the smaller GLUE datasets are prone to stochastic noise), did not do any early stopping, and did not start from a fine-tune on a previous GLUE task (as is often done to decrease the variability of the small dataset noise and prevent overfitting).

All runs:

* Started from a freshly initialized LoRA injection with rank 8 into the RoBERTa-base model
* The training is done for exactly 6 epochs for each task, without any early stopping.
* During the first 2 epochs the learning rate was linearly scaled up to the maximum value, and then linearly decayed towards zero over the remaining 4 epochs.
* The maximum learning rate for all tasks was 5e-4.
* The batch size for all tasks was 16

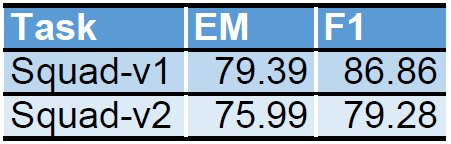
The RoBERTa-base model has 124.6 million parameters. With the LoRA parameters, the biases, and layer norms we only have 420 thousand unfrozen parameters to train. This means we essentially train on only 0.34% of the original parameters.

The number of parameters introduced by LoRA for these specific tasks is remarkably minimal, amounting to just 1.7 MB of actual disk size. You can find the trained LoRAs in the Git repo in the *Output* folder.

Post-training, we reloaded the LoRA parameters, reapplied them, and tested performance on each task’s validation set. Below are the results:



Performance on GLUE Benchmarks using LoRA



Performance on SQuAD Datasets using LoRA

Likely these results could be greatly improved with some hyperparameter fine-tuning. Nevertheless, it clearly proves that our LoRA implementation is working and our injected low-rank matrices are learning.

**Possible Improvements**

Reflecting on our implementation, one might wonder: “Could there have been a more efficient, generalizable (i.e. transferable to other network architectures) approach than recoding the self-attention class and performing complex replacements?”

Indeed we could have simply implemented a wrapper around the pytorch nn.Linear function and be more specific on which layers we want to replace with it, via checking their names. Similarly, you could write wrappers around most base pytorch layers and be able to quickly adapt LoRA to new network architectures. To give a quick sketch of how this could be done:

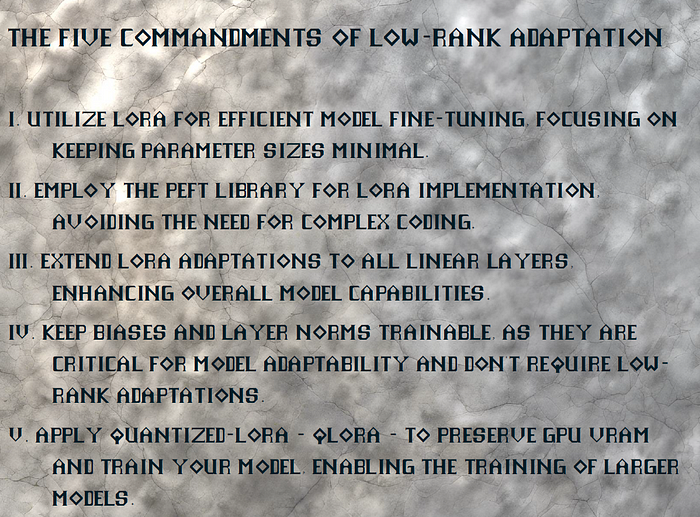
class LoraLinear(nn.Linear):  
 """  
 Extends a PyTorch linear layer with Low-Rank Adaptation (LoRA).  
 LoRA adds two matrices to the layer, allowing for efficient training of large models.  
 """  
 def \_\_init\_\_(self, in\_features, out\_features, r=8, \*args, \*\*kwargs):  
 super().\_\_init\_\_(in\_features, out\_features, \*args, \*\*kwargs)  
  
 # Initialize LoRA matrices  
 self.lora\_matrix\_B = nn.Parameter(torch.zeros(out\_features, r))  
 self.lora\_matrix\_A = nn.Parameter(torch.randn(r, in\_features))  
   
 # Freeze the original weight matrix  
 self.weight.requires\_grad = False  
  
 def forward(self, x: Tensor) -> Tensor:  
 # Compute LoRA weight adjustment  
 lora\_weights = torch.matmul(self.lora\_matrix\_B, self.lora\_matrix\_A)  
 # Apply the original and LoRA-adjusted linear transformations  
 return super().forward(x) + F.linear(x, lora\_weights)

This is actually (close to) the way the huggingface PEFT (Parameter-Efficient Fine-Tuning) library implements LoRA. For any practical application, where you are not trying to learn, I strongly recommend using it, instead of coding your own.

Also it became a rather common practice to inject LoRA into all linear layers as well (i.e. all matrices of the self-attention and the two linear layers for the fully connected forward network). It is usually a good idea to keep the biases and layer-norms trainable, in addition to the LoRA parameters. As they already are small you won’t need a low-rank injection for them.

Quantizing the original matrix weights to conserve GPU VRAM is also advisable, facilitating the training of larger models on a given GPU. This can be efficiently done using the bits-and-bytes library, now fully integrated with Hugging Face (see references).

Summarizing, here are the Five Commandments of Low-Rank Adaptation in a serious setting:



*The Five Commandments of Low-Rank Adaptation*

If you find the inscribed stone tablet hard to read, here they are again in plain text:

*The Five Commandments of Low-Rank Adaptation*

*1. Utilize LoRA for efficient model fine-tuning, focusing on keeping parameter sizes minimal.  
2. Employ the PEFT library for LoRA implementation, avoiding the need for complex coding.  
3. Extend LoRA adaptations to all linear layers, enhancing overall model capabilities.  
4. Keep biases and layer norms trainable, as they are critical for model adaptability and don’t require low-rank adaptations.  
5. Apply Quantized-LoRA — QLoRA — to preserve GPU VRAM and train your model, enabling the training of larger models.*

Remember, training with QLoRA may be a bit slower than LoRA, as it involves de-quantizing matrices during each multiplication. For instance, when fine-tuning something massive like Llama-7B, QLoRA requires about 75% less VRAM but is roughly 40% slower compared to standard LoRA. For more insights, check out the blogposts I linked in the references.

**A Step-by-Step Guide to PEFT Implementation**

Let's look at how to actually obey our commandments and implement a better version via PEFT.

First off, let’s load our model in a quantized manner. Thanks to the bitsandbytes integration with the Huggingface transformers library (introduced in May 2023), this is a breeze.

We have to specify a configuration file and then load the model directly from huggingface with this quantization. Generally, it is best to use the *AutoModel* objects from transformers. It is difficult to load a quantized model as a submodule of a larger, newly defined, nn.module object. You should generally work with the raw models from huggingface and thus import directly an AutoModelForSequenceClassification for the GLUE tasks and AutoModelForQuestionAnswering for the SQuAD benchmarks. In the configuration we can also specify which parameters not to quantize: Here we have to register the classification or qa-output heads, as we want to train these in full, i.e. without LoRA, as these were newly initialized for the fine-tuning and were never part of the pre-trained base model.

import bitsandbytes as bnb  
from transformers import AutoModel, AutoModelForSequenceClassification, BitsAndBytesConfig  
  
# Configuration to load a quantized model  
bnb\_config = BitsAndBytesConfig(  
 load\_in\_4bit=True, # Enable 4-bit loading  
 bnb\_4bit\_quant\_type="nf4",  
 bnb\_4bit\_compute\_dtype=torch.bfloat16,  
 llm\_int8\_skip\_modules=['classifier', 'qa\_outputs'], # Skip these for quantization  
)  
  
# Load the model from Huggingface with quantization  
model = AutoModelForSequenceClassification.from\_pretrained('roberta-base',  
 torch\_dtype="auto", quantization\_config=bnb\_config)

You can verify the 4-bit loading by inspecting the model’s modules and parameter data types:

# Verify 4-bit loading  
print("Verifying 4-bit elements (Linear4bit) in the attention layer:")  
print(model.roberta.encoder.layer[4].attention)  
  
print("Checking for uint8 data type:")  
print(model.roberta.encoder.layer[4].attention.self.query.weight.dtype)

Now on to inject the LoRA parameters with PEFT. Note that the PEFT library is much more flexible, also when working with custom models or other convoluted structures, so as long as you are only doing LoRA instead of QLoRA (quantization is usually the tricky part).

The PEFT library targets the modules to replace via their names; thus we have to take a look at the models model.named\_parameters(). Here is how this looks for the non-quantized roberta-base model.

Module Parameters  
---------------------------------------------------------- ------------  
roberta.embeddings.word\_embeddings.weight 38\_603\_520  
roberta.embeddings.position\_embeddings.weight 394\_752  
roberta.embeddings.token\_type\_embeddings.weight 768  
roberta.embeddings.LayerNorm.weight 768  
roberta.embeddings.LayerNorm.bias 768  
roberta.encoder.layer.0.attention.self.query.weight 589\_824  
roberta.encoder.layer.0.attention.self.query.bias 768  
roberta.encoder.layer.0.attention.self.key.weight 589\_824  
roberta.encoder.layer.0.attention.self.key.bias 768  
roberta.encoder.layer.0.attention.self.value.weight 589\_824  
roberta.encoder.layer.0.attention.self.value.bias 768  
roberta.encoder.layer.0.attention.output.dense.weight 589\_824  
roberta.encoder.layer.0.attention.output.dense.bias 768  
roberta.encoder.layer.0.attention.output.LayerNorm.weight 768  
roberta.encoder.layer.0.attention.output.LayerNorm.bias 768  
roberta.encoder.layer.0.intermediate.dense.weight 2\_359\_296  
roberta.encoder.layer.0.intermediate.dense.bias 3\_072  
roberta.encoder.layer.0.output.dense.weight 2\_359\_296  
roberta.encoder.layer.0.output.dense.bias 768  
roberta.encoder.layer.0.output.LayerNorm.weight 768  
roberta.encoder.layer.0.output.LayerNorm.bias 768  
roberta.encoder.layer.1.attention.self.query.weight 589\_824  
...  
roberta.encoder.layer.11.output.LayerNorm.bias 768  
classifier.dense.weight 589\_824  
classifier.dense.bias 768  
classifier.out\_proj.weight 1\_536  
classifier.out\_proj.bias 2  
---------------------------------------------------------- ------------  
TOTAL 124\_647\_170

We can then specify the LoRA targets to select for these strings. The check is if it contains the specified substring in its full name. Thus writing query and value is equivalent to our from-scratch implementation above. For the dense layers we have to be a bit more careful as the classifier also has a dense output. If we wish to fine-tune the other dense layers we have to be more specific via intermediate.dense and output.dense.

All parameters that were not injected with LoRA parameters are automatically frozen, i.e. will not receive any gradient updates. If there are any layers we want to train in their original form we can specify them by passing a list to the modules\_to\_save parameters of the Lora-Config. In our case, we want to add the LayerNorm here and the fine-tune heads for GLUE and SQuAD. Note that not each element of the lists has to match something. We can simply add the classifier and qa\_outputs to this list and then have a single configuration file that will work correctly for both tasks.

For the bias parameters you can use the convenient configuration parameter bias. You can specify either *all* to retrain all biases of all modules, *lora\_only* to only train the injected ones, or *none* to keep all biases constant during training.

The following example injects a LoRA with rank 2. We specify the alpha parameters with the 8 above, as this was the rank we tried first and should allow us to keep the original learning rate from our from-scratch example.

import peft  
  
# Config for the LoRA Injection via PEFT  
peft\_config = peft.LoraConfig(  
 r=2, # rank dimension of the LoRA injected matrices  
 lora\_alpha=8, # parameter for scaling, use 8 here to make it comparable with our own implementation  
 target\_modules=['query', 'key', 'value', 'intermediate.dense', 'output.dense'], # be precise about dense because classifier has dense too  
 modules\_to\_save=["LayerNorm", "classifier", "qa\_outputs"], # Retrain the layer norm; classifier is the fine-tune head; qa\_outputs is for SQuAD  
 lora\_dropout=0.1, # dropout probability for layers  
 bias="all", # none, all, or lora\_only  
)  
  
model = peft.get\_peft\_model(model, peft\_config)

Remember, specifying more modules for LoRA injections might increase VRAM requirements. If you encounter VRAM limitations, consider reducing the number of target modules or the LoRA rank.

For training, especially with QLoRA, choose an optimizer that’s compatible with quantized matrices. Replace your standard torch optimizer with a bitsandbytes variant like so:

import torch  
import bitsandbytes as bnb  
  
# replace this  
optimizer = torch.optim.AdamW(args here)  
# with this  
optimizer = bnb.optim.AdamW8bit(same args here)

You can then train this model like before, without having to explicitly worry about QLoRA during training.

Once training is complete, the process for saving and reloading your model is straightforward. Use model.save\_pretrained to save your model, specifying the desired filename. The PEFT library will automatically create a directory at this location, where it stores the model weights and a configuration file. This file includes essential details like the base model and LoRA configuration parameters.

To reload the model, utilize peft.AutoPeftModel.from\_pretrained, passing the directory path as an argument. A crucial point to remember is that the LoRA configuration currently does not retain the number of classes for which AutoModelForSequenceClassification was initialized. When using from\_pretrained, you need to manually input this class number as an additional parameter. Failing to do so will result in an error.

The reloaded model will comprise the original base model with the LoRA adapters applied. Should you decide to integrate the LoRA adapters permanently into the base model matrices, simply execute model.merge\_and\_unload().

For a more hands-on understanding and detailed instructions, have a look at the GitHub repository. There, you’ll find two notebooks titled *Train-QLoRA-with-PEFT.ipynb* and *Load-LoRA-Weights-PEFT.ipynb*, providing a step-by-step example for training and loading models with PEFT.

**Conclusion**

“We shall not cease from exploration, and the end of all our exploring will be to arrive where we started and know the place for the first time.”

— from “Little Gidding” by T.S. Eliot

This journey has taken us from a straightforward, albeit hard-coded, LoRA implementation to a deeper understanding of low-rank adaptors, their practical implementation, and benchmark testing.

We explored an alternative, more efficient implementation strategy and delved into the elegance of existing libraries like PEFT for LoRA integration.

Our adventure concludes with practical guidelines for employing LoRA, encapsulated in the ‘Five Commandments,’ ensuring efficient and effective use of this technique in real-world applications and a step-by-step guide on how to implement them in practice.

**References**

*All images, unless otherwise noted, are by the author.*

* Original LoRA paper: <https://arxiv.org/pdf/2106.09685.pdf>
* QLoRA paper: <https://arxiv.org/abs/2305.14314>
* Sentdex Guide on QLoRA finetuning: <https://www.youtube.com/watch?v=J_3hDqSvpmg>
* Blogpost about LoRA fine-tuning on Llama: <https://www.anyscale.com/blog/fine-tuning-llms-lora-or-full-parameter-an-in-depth-analysis-with-llama-2>
* bitsandbytes Hugging Face integration: <https://huggingface.co/blog/4bit-transformers-bitsandbytes>
* LoRA training insights: <https://lightning.ai/pages/community/lora-insights/>
* Expected VRAM savings LoRA vs QLoRA when fine-tuning a Llama model: <https://cloud.google.com/vertex-ai/docs/model-garden/lora-qlora>
* The font I used for the stone slab text, in case you want to create your own: <https://www.fontspace.com/sharp-objects-nbp-font-f14469>