**Building a Million-Parameter LLM from Scratch Using Python**

A Step-by-Step Guide to Replicating LLaMA Architecture

[[](https://medium.com/@fareedkhandev?source=post_page-----f612398f06c2--------------------------------)](https://medium.com/@fareedkhandev?source=post_page-----f612398f06c2--------------------------------)

[[Level Up Coding](https://levelup.gitconnected.com/?source=post_page-----f612398f06c2--------------------------------)](https://levelup.gitconnected.com/?source=post_page-----f612398f06c2--------------------------------)

[Fareed Khan](https://medium.com/@fareedkhandev?source=post_page-----f612398f06c2--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fb856005e5ecd&operation=register&redirect=https%3A%2F%2Flevelup.gitconnected.com%2Fbuilding-a-million-parameter-llm-from-scratch-using-python-f612398f06c2&user=Fareed+Khan&userId=b856005e5ecd&source=post_page-b856005e5ecd----f612398f06c2---------------------post_header-----------)

Published in

[Level Up Coding](https://levelup.gitconnected.com/?source=post_page-----f612398f06c2--------------------------------)

·

25 min read

·

6 days ago

429

4



Image from GoogleDeepMind (Open Source available on pexels)

Making your own Large Language Model (LLM) is a cool thing that many big companies like Google, Twitter, and Facebook are doing. They release different versions of these models, like 7 billion, 13 billion, or 70 billion. Even smaller communities are doing it too. You might have read blogs or watched videos on creating your own LLM, but they usually talk a lot about theory and not so much about the actual steps and code.

In this blog, I’ll try to make an LLM with only 2.3 million parameters, and the interesting part is we won’t need a fancy GPU for it. We’ll follow a [LLaMA 1 Paper](https://arxiv.org/abs/2302.13971" \t "_blank) Approach to guide us. Don’t worry; we’ll keep it simple and use a basic dataset so you can see how easy it is to create your own million-parameter LLM.

**Table of Contents**

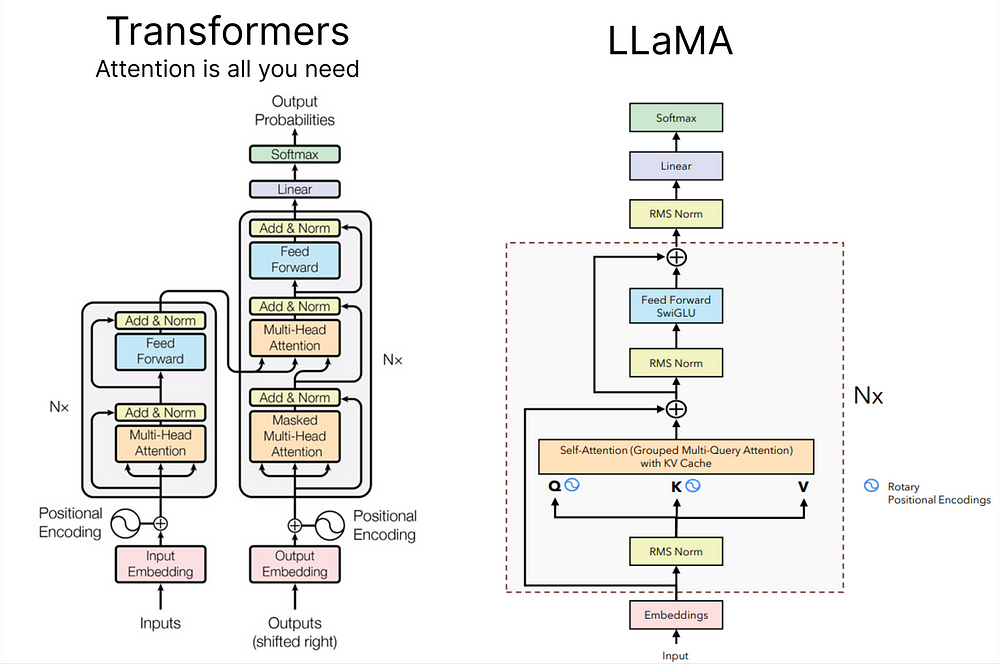
· [Prerequisites](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#c1c5)  
· [Understanding the Transformer Architecture of LLaMA](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#f469)  
∘ [Pre-normalization Using RMSNorm](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#2c34)  
∘ [SwiGLU Activation Function](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2" \l "c371)  
∘ [Rotary Embeddings (RoPE)](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#a65c)  
· [Setting the Stage](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#cf9a)  
· [Data Preprocessing](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#3430)  
· [Evaluation Strategy](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#0ba9)  
· [Setting Up a Base Neural Network Model](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#fbc7)  
· [Replicating LLaMA Architecture](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#2bcd)  
∘ [RMSNorm for pre-normalization](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2" \l "bafd)  
∘ [Rotary Embeddings](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#4753)  
∘ [SwiGLU activation function](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2" \l "8180)  
· [Experimenting with hyperparameters](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#f7d6)  
· [Saving Your Language Model (LLM)](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#21f1)  
· [Conclusion](https://levelup.gitconnected.com/building-a-million-parameter-llm-from-scratch-using-python-f612398f06c2#2365)

**Prerequisites**

Make sure you have a basic understanding of object-oriented programming (**OOP**) and neural networks (**NN**). Familiarity with **PyTorch** will also be helpful in coding.

**Understanding the Transformer Architecture of LLaMA**

Before diving into creating our own LLM using the LLaMA approach, it’s essential to understand the architecture of LLaMA. Below is a comparison diagram between the vanilla transformer and LLaMA.



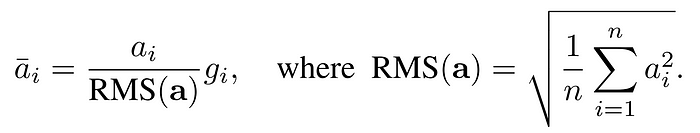
*Difference between Transformers and Llama architecture (Llama architecture by*[***Umar Jamil***](https://github.com/hkproj)*)*

In case you’re not familiar with the vanilla transformer architecture, you can read [this blog](https://medium.com/@fareedkhandev/understanding-transformers-a-step-by-step-math-example-part-1-a7809015150a) for a basic guide.

Let’s look into the essential concepts of LLaMA with a bit more detail:

**Pre-normalization Using RMSNorm:**

In the LLaMA approach, a technique called RMSNorm is employed for normalizing the input of each transformer sub-layer. This method is inspired by GPT-3 and is designed to optimize the computational cost associated with Layer Normalization. RMSNorm provides similar performance to LayerNorm but reduces the running time significantly (by 7%∼64%).

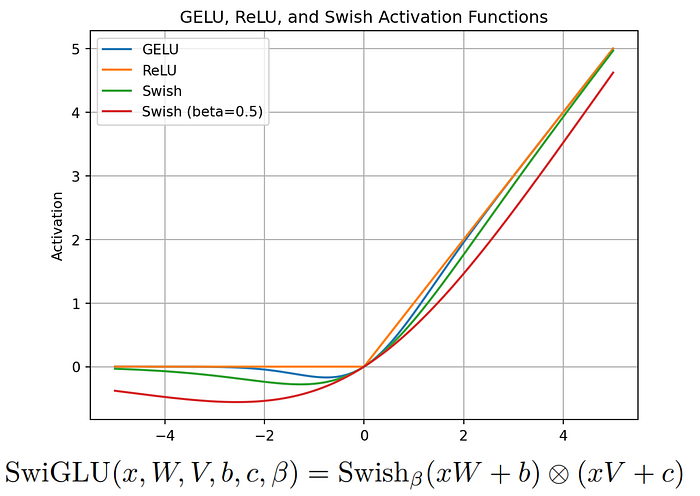


Root Mean Square Layer Normalization Paper (<https://arxiv.org/abs/1910.07467>)

It achieves this by emphasizing re-scaling invariance and regulating the summed inputs based on the root mean square (RMS) statistic. The primary motivation is to simplify LayerNorm by removing the mean statistic. Interested readers can explore the detailed implementation of RMSNorm [here](https://github.com/bzhangGo/rmsnorm/blob/master/rmsnorm_torch.py).

**SwiGLU Activation Function:**

LLaMA introduces the SwiGLU activation function, drawing inspiration from PaLM. To understand SwiGLU, it’s essential to first grasp the Swish activation function. SwiGLU extends Swish and involves a custom layer with a dense network to split and multiply input activations.



SwiGLU: GLU Variants Improve Transformer (<https://kikaben.com/swiglu-2020/>)

The aim is to enhance the expressive power of the model by introducing a more sophisticated activation function. Further details on SwiGLU can be found in the associated [paper](https://arxiv.org/pdf/2002.05202v1.pdf).

**Rotary Embeddings (RoPE):**

Rotary Embeddings, or RoPE, is a type of position embedding used in LLaMA. It encodes absolute positional information using a rotation matrix and naturally includes explicit relative position dependency in self-attention formulations. RoPE offers advantages such as scalability to various sequence lengths and decaying inter-token dependency with increasing relative distances.

This is achieved by encoding relative positions through multiplication with a rotation matrix, resulting in decayed relative distances — a desirable feature for natural language encoding. Those interested in the mathematical details can refer to the [RoPE paper](https://arxiv.org/pdf/2104.09864v4.pdf" \t "_blank).

In addition to these concepts, the LLaMA paper introduces other significant approaches, including the use of the **AdamW optimizer** with specific parameters, efficient implementations such as the causal [multi-head attention operator](https://facebookresearch.github.io/xformers/components/mha.html) available in the xformers library, and manually implemented backward functions for transformer layers to optimize computation during backward passes.

A special acknowledgment and thanks to [Anush Kumar](https://akgeni.medium.com/) for providing an in-depth explanation of each crucial aspect of LLaMA.

**Setting the Stage**

We’ll be working with a range of Python libraries throughout this project, so let’s import them:

# PyTorch for implementing LLM (No GPU)  
import torch  
  
# Neural network modules and functions from PyTorch  
from torch import nn  
from torch.nn import functional as F  
  
# NumPy for numerical operations  
import numpy as np  
  
# Matplotlib for plotting Loss etc.  
from matplotlib import pyplot as plt  
  
# Time module for tracking execution time  
import time  
  
# Pandas for data manipulation and analysis  
import pandas as pd  
  
# urllib for handling URL requests (Downloading Dataset)  
import urllib.request

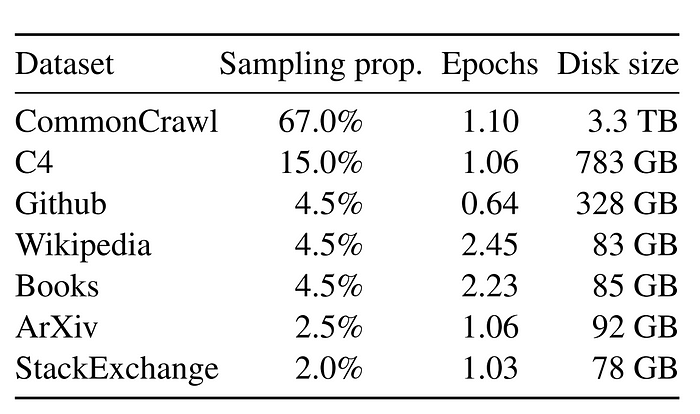
Furthermore, I’m creating a configuration object that stores model parameters.

# Configuration object for model parameters  
MASTER\_CONFIG = {  
 # Adding parameters later  
}

This approach maintains flexibility, allowing for the addition of more parameters as needed in the future.

**Data Preprocessing**

In the original LLaMA paper, diverse open-source datasets were employed to train and evaluate the model.



<https://research.facebook.com/publications/llama-open-and-efficient-foundation-language-models/>

Unfortunately, utilizing extensive datasets may be impractical for smaller projects. Therefore, for our implementation, we’ll take a more modest approach by creating a dramatically scaled-down version of LLaMA.

Given the constraints of not having access to vast amounts of data, we will focus on training a simplified version of LLaMA using the TinyShakespeare dataset. This open source dataset, available [here](https://github.com/karpathy/char-rnn/blob/master/data/tinyshakespeare/input.txt), contains approximately 40,000 lines of text from various Shakespearean works. This choice is influenced by the [Makemore series by Karpathy](https://www.youtube.com/playlist?list=PLAqhIrjkxbuWI23v9cThsA9GvCAUhRvKZ" \t "_blank), which provides valuable insights into training language models.

While LLaMA was trained on an extensive dataset comprising **1.4 trillion** tokens, our dataset, TinyShakespeare, containing around **1 million characters**.

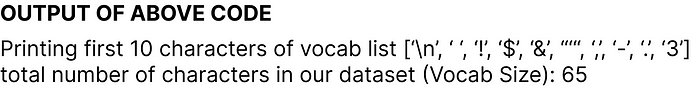
First, let’s obtain our dataset by downloading it:

# The URL of the raw text file on GitHub  
url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt"  
  
# The file name for local storage  
file\_name = "tinyshakespeare.txt"  
  
# Execute the download  
urllib.request.urlretrieve(url, file\_name)

This Python script fetches the tinyshakespeare dataset from the specified URL and saves it locally with the filename **“tinyshakespeare.txt.”**

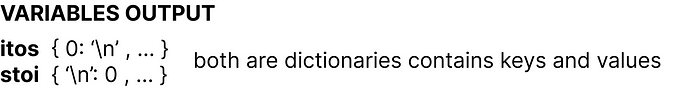
Next, let’s determine the vocabulary size, which represents the unique number of characters in our dataset. Here’s the code snippet:

# Read the content of the dataset  
lines = open("tinyshakespeare.txt", 'r').read()  
  
# Create a sorted list of unique characters in the dataset  
vocab = sorted(list(set(lines)))  
  
# Display the first 10 characters in the vocabulary list  
print('Printing the first 10 characters of the vocab list:', vocab[:10])  
  
# Output the total number of characters in our dataset (Vocabulary Size)  
print('Total number of characters in our dataset (Vocabulary Size):', len(vocab))



Now, we’re creating mappings between integers to characters (**itos**) and characters to integers (**stoi**). Here’s the code:

# Mapping integers to characters (itos)  
itos = {i: ch for i, ch in enumerate(vocab)}  
  
# Mapping characters to integers (stoi)  
stoi = {ch: i for i, ch in enumerate(vocab)}



In the original LLaMA paper, the [SentencePiece byte-pair encoding tokenizer](https://github.com/google/sentencepiece" \t "_blank) from Google was used. However, for simplicity, we’ll opt for a basic character-level tokenizer. Let’s create encode and decode functions that we’ll later apply to our dataset:

# Encode function: Converts a string to a list of integers using the mapping stoi  
def encode(s):  
 return [stoi[ch] for ch in s]  
  
# Decode function: Converts a list of integers back to a string using the mapping itos  
def decode(l):  
 return ''.join([itos[i] for i in l])  
  
# Example: Encode the string "hello" and then decode the result  
decode(encode("morning"))

The final line will output morning confirms the proper functionality of the encode and decode functions.

We are now converting our dataset into a torch tensor, specifying its data type for further operations using **PyTorch**:

# Convert the dataset into a torch tensor with specified data type (dtype)  
dataset = torch.tensor(encode(lines), dtype=torch.int8)  
  
# Display the shape of the resulting tensor  
print(dataset.shape)

The output istorch.Size([1115394]) indicates that our dataset contains approximately **one million tokens**. It's worth noting that this is significantly smaller than the LLaMA dataset, which consists of **1.4 trillion tokens**.

We’ll create a function responsible for splitting our dataset into training, validation, or test sets. In machine learning or deep learning projects, such splits are crucial for developing and evaluating models, and the same principle applies here in replicating a Large Language Model (LLM) approach:

# Function to get batches for training, validation, or testing  
def get\_batches(data, split, batch\_size, context\_window, config=MASTER\_CONFIG):  
 # Split the dataset into training, validation, and test sets  
 train = data[:int(.8 \* len(data))]  
 val = data[int(.8 \* len(data)): int(.9 \* len(data))]  
 test = data[int(.9 \* len(data)):]  
  
 # Determine which split to use  
 batch\_data = train  
 if split == 'val':  
 batch\_data = val  
 if split == 'test':  
 batch\_data = test  
  
 # Pick random starting points within the data  
 ix = torch.randint(0, batch\_data.size(0) - context\_window - 1, (batch\_size,))  
  
 # Create input sequences (x) and corresponding target sequences (y)  
 x = torch.stack([batch\_data[i:i+context\_window] for i in ix]).long()  
 y = torch.stack([batch\_data[i+1:i+context\_window+1] for i in ix]).long()  
  
 return x, y

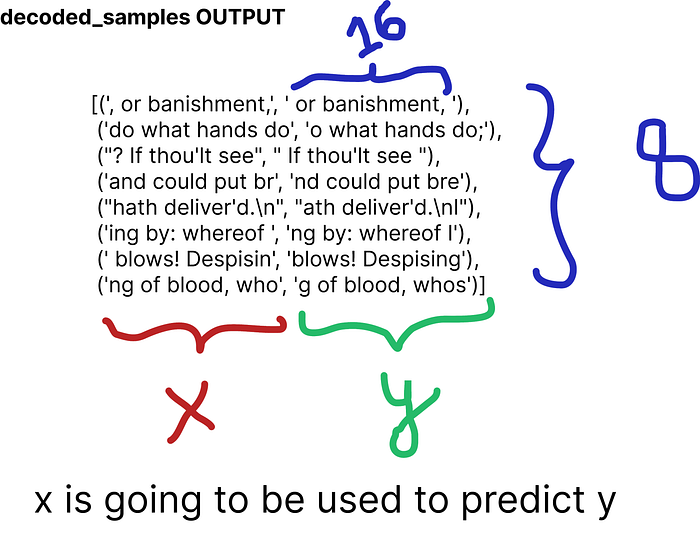
Now that our splitting function is defined, let’s establish two parameters crucial for this process:

# Update the MASTER\_CONFIG with batch\_size and context\_window parameters  
MASTER\_CONFIG.update({  
 'batch\_size': 8, # Number of batches to be processed at each random split  
 'context\_window': 16 # Number of characters in each input (x) and target (y) sequence of each batch  
})

batch\_size determines how many batches are processed at each random split, while context\_window specifies the number of characters in each input (x) and target (y) sequence of each batch.

Let’s print a random sample from the train split of batch 8 and context window 16 from our dataset:

# Obtain batches for training using the specified batch size and context window  
xs, ys = get\_batches(dataset, 'train', MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'])  
  
# Decode the sequences to obtain the corresponding text representations  
decoded\_samples = [(decode(xs[i].tolist()), decode(ys[i].tolist())) for i in range(len(xs))]  
  
# Print the random sample  
print(decoded\_samples)



**Evaluation Strategy**

Now, we are set to create a function dedicated to evaluating our self-created LLaMA architecture. The reason for doing this before defining the actual model approach is to enable continuous evaluation during the training process.

@torch.no\_grad() # Don't compute gradients for this function  
def evaluate\_loss(model, config=MASTER\_CONFIG):  
 # Placeholder for the evaluation results  
 out = {}  
   
 # Set the model to evaluation mode  
 model.eval()  
  
 # Iterate through training and validation splits  
 for split in ["train", "val"]:  
 # Placeholder for individual losses  
 losses = []  
  
 # Generate 10 batches for evaluation  
 for \_ in range(10):  
 # Get input sequences (xb) and target sequences (yb)  
 xb, yb = get\_batches(dataset, split, config['batch\_size'], config['context\_window'])  
   
 # Perform model inference and calculate the loss  
 \_, loss = model(xb, yb)  
   
 # Append the loss to the list  
 losses.append(loss.item())  
  
 # Calculate the mean loss for the split and store it in the output dictionary  
 out[split] = np.mean(losses)  
   
 # Set the model back to training mode  
 model.train()  
   
 return out

We have used the **loss** as a metric to assess the performance of the model during training iterations. Our function iterates through the training and validation splits, computes the mean loss over 10 batches for each split, and finally returns the results. The model is then set back to training mode with model.train().

**Setting Up a Base Neural Network Model**

We’re building a basic neural network that we’ll improve later using LLaMA techniques.

# Definition of a basic neural network class  
class SimpleBrokenModel(nn.Module):  
 def \_\_init\_\_(self, config=MASTER\_CONFIG):  
 super().\_\_init\_\_()  
 self.config = config  
  
 # Embedding layer to convert character indices to vectors (vocab size: 65)  
 self.embedding = nn.Embedding(config['vocab\_size'], config['d\_model'])  
  
 # Linear layers for modeling relationships between features  
 # (to be updated with SwiGLU activation function as in LLaMA)  
 self.linear = nn.Sequential(  
 nn.Linear(config['d\_model'], config['d\_model']),  
 nn.ReLU(), # Currently using ReLU, will be replaced with SwiGLU as in LLaMA  
 nn.Linear(config['d\_model'], config['vocab\_size']),  
 )  
  
 # Print the total number of model parameters  
 print("Model parameters:", sum([m.numel() for m in self.parameters()]))

In the current architecture, the embedding layer has a vocabulary size of 65, representing the characters in our dataset. As this serves as our base model, we are using **ReLU**as the activation function in the linear layers; however, this will later be replaced with SwiGLU, as used in LLaMA.

To create a forward pass for our base model, we must define a forward function within our NN model.

# Definition of a basic neural network class  
class SimpleBrokenModel(nn.Module):  
 def \_\_init\_\_(self, config=MASTER\_CONFIG):  
  
 # Rest of the code   
 ...   
  
 # Forward pass function for the base model  
 def forward(self, idx, targets=None):  
 # Embedding layer converts character indices to vectors  
 x = self.embedding(idx)  
   
 # Linear layers for modeling relationships between features  
 a = self.linear(x)  
   
 # Apply softmax activation to obtain probability distribution  
 logits = F.softmax(a, dim=-1)  
  
 # If targets are provided, calculate and return the cross-entropy loss  
 if targets is not None:  
 # Reshape logits and targets for cross-entropy calculation  
 loss = F.cross\_entropy(logits.view(-1, self.config['vocab\_size']), targets.view(-1))  
 return logits, loss  
  
 # If targets are not provided, return the logits  
 else:  
 return logits  
  
 # Print the total number of model parameters  
 print("Model parameters:", sum([m.numel() for m in self.parameters()]))

This forward pass function takes character indices (idx) as input, applies the embedding layer, passes the result through linear layers, applies a softmax activation to obtain a probability distribution (logits). If targets are provided, it calculates the cross-entropy loss and returns both logits and loss. If targets are not provided, it returns only the logits.

To instantiate this model, we can directly invoke the class and print the total number of parameters in our Simple Neural Network Model. We’ve set the dimension of our linear layers to 128, specifying this value in our config object:

# Update MASTER\_CONFIG with the dimension of linear layers (128)  
MASTER\_CONFIG.update({  
 'd\_model': 128,  
})  
  
# Instantiate the SimpleBrokenModel using the updated MASTER\_CONFIG  
model = SimpleBrokenModel(MASTER\_CONFIG)  
  
# Print the total number of parameters in the model  
print("Total number of parameters in the Simple Neural Network Model:", sum([m.numel() for m in model.parameters()]))



Our Simple Neural Network Model comprises approximately 33,000 parameters.

Similarly, to compute logits and loss, we only need to feed our split dataset into our model:

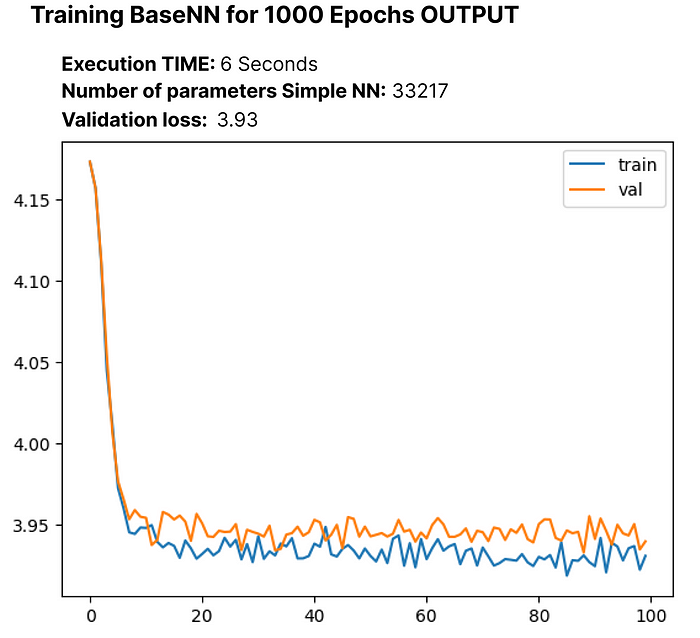
# Obtain batches for training using the specified batch size and context window  
xs, ys = get\_batches(dataset, 'train', MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'])  
  
# Calculate logits and loss using the model  
logits, loss = model(xs, ys)

To train our base model and note its performance, we need to specify some parameters. We are training for a total of 1000 epochs. Increasing the batch size to 32 from 8, and set the log\_interval to 10, indicating that the code will print or log information about the training progress every 10 batches. For optimization, we’ll use the Adam optimizer.

# Update MASTER\_CONFIG with training parameters  
MASTER\_CONFIG.update({  
 'epochs': 1000, # Number of training epochs  
 'log\_interval': 10, # Log information every 10 batches during training  
 'batch\_size': 32, # Increase batch size to 32  
})  
  
# Instantiate the SimpleBrokenModel with updated configuration  
model = SimpleBrokenModel(MASTER\_CONFIG)  
  
# Define the Adam optimizer for model parameters  
optimizer = torch.optim.Adam(  
 model.parameters(), # Pass the model parameters to the optimizer  
)

Let’s execute the training process and capture the loss from our base model, including the total number of parameters. **Additionally, each line is commented for clarity**:

# Function to perform training  
def train(model, optimizer, scheduler=None, config=MASTER\_CONFIG, print\_logs=False):  
 # Placeholder for storing losses  
 losses = []  
   
 # Start tracking time  
 start\_time = time.time()  
  
 # Iterate through epochs  
 for epoch in range(config['epochs']):  
 # Zero out gradients  
 optimizer.zero\_grad()  
  
 # Obtain batches for training  
 xs, ys = get\_batches(dataset, 'train', config['batch\_size'], config['context\_window'])  
  
 # Forward pass through the model to calculate logits and loss  
 logits, loss = model(xs, targets=ys)  
  
 # Backward pass and optimization step  
 loss.backward()  
 optimizer.step()  
  
 # If a learning rate scheduler is provided, adjust the learning rate  
 if scheduler:  
 scheduler.step()  
  
 # Log progress every specified interval  
 if epoch % config['log\_interval'] == 0:  
 # Calculate batch time  
 batch\_time = time.time() - start\_time  
   
 # Evaluate loss on validation set  
 x = evaluate\_loss(model)  
   
 # Store the validation loss  
 losses += [x]  
   
 # Print progress logs if specified  
 if print\_logs:  
 print(f"Epoch {epoch} | val loss {x['val']:.3f} | Time {batch\_time:.3f} | ETA in seconds {batch\_time \* (config['epochs'] - epoch)/config['log\_interval'] :.3f}")  
   
 # Reset the timer  
 start\_time = time.time()  
  
 # Print learning rate if a scheduler is provided  
 if scheduler:  
 print("lr: ", scheduler.get\_lr())  
  
 # Print the final validation loss  
 print("Validation loss: ", losses[-1]['val'])  
   
 # Plot the training and validation loss curves  
 return pd.DataFrame(losses).plot()  
  
# Execute the training process  
train(model, optimizer)



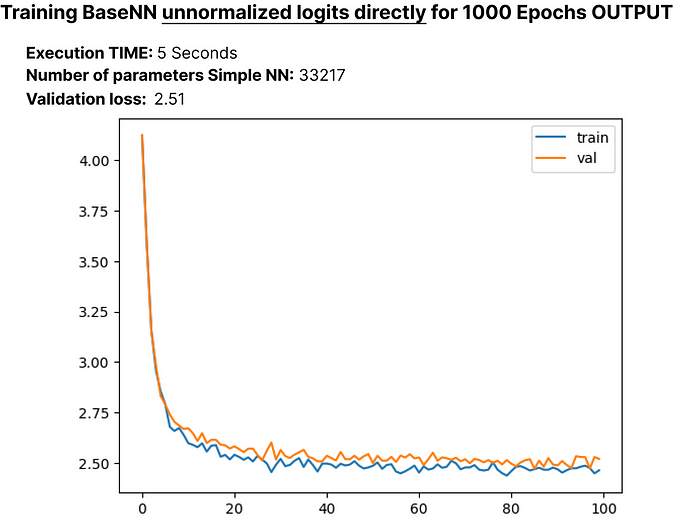
The initial cross-entropy loss before training stands at 4.17, and after 1000 epochs, it reduces to 3.93. In this context, cross-entropy reflects the likelihood of selecting the incorrect word.

Our model incorporates a softmax layer on the logits, which transforms a vector of numbers into a probability distribution. Let’s use the built-in F.cross\_entropy function, we need to directly pass in the [unnormalized logits](https://pytorch.org/docs/stable/generated/torch.nn.functional.cross_entropy.html). Consequently, we will modify our model accordingly.

# Modified SimpleModel class without softmax layer  
class SimpleModel(nn.Module):  
 def \_\_init\_\_(self, config):  
   
 # Rest of the code  
 ...  
  
 def forward(self, idx, targets=None):  
 # Embedding layer converts character indices to vectors  
 x = self.embedding(idx)  
   
 # Linear layers for modeling relationships between features  
 logits = self.linear(x)  
  
 # If targets are provided, calculate and return the cross-entropy loss  
 if targets is not None:  
  
 # Rest of the code  
 ...

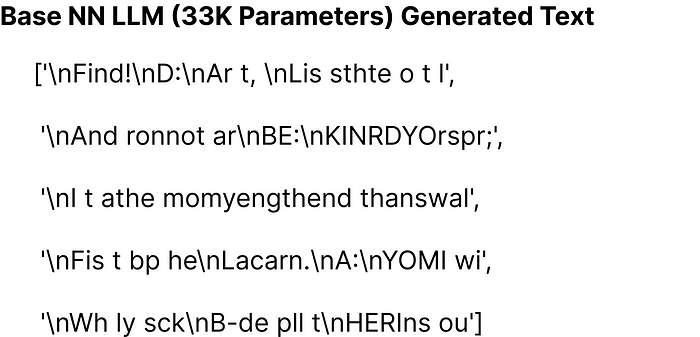
Let’s recreate the updated SimpleModel and train it for 1000 epochs to observe any changes:

# Create the updated SimpleModel  
model = SimpleModel(MASTER\_CONFIG)  
  
# Obtain batches for training  
xs, ys = get\_batches(dataset, 'train', MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'])  
  
# Calculate logits and loss using the model  
logits, loss = model(xs, ys)  
  
# Define the Adam optimizer for model parameters  
optimizer = torch.optim.Adam(model.parameters())  
  
# Train the model for 100 epochs  
train(model, optimizer)



After reducing the loss to 2.51, let’s explore how our language model with approximately **33,000 parameters** generates text during inferencing. We’ll create a ‘generate’ function, which we’ll later use when replicating LLaMA:

# Generate function for text generation using the trained model  
def generate(model, config=MASTER\_CONFIG, max\_new\_tokens=30):  
 idx = torch.zeros(5, 1).long()  
 for \_ in range(max\_new\_tokens):  
 # Call the model  
 logits = model(idx[:, -config['context\_window']:])  
 last\_time\_step\_logits = logits[  
 :, -1, :  
 ] # all the batches (1), last time step, all the logits  
 p = F.softmax(last\_time\_step\_logits, dim=-1) # softmax to get probabilities  
 idx\_next = torch.multinomial(  
 p, num\_samples=1  
 ) # sample from the distribution to get the next token  
 idx = torch.cat([idx, idx\_next], dim=-1) # append to the sequence  
 return [decode(x) for x in idx.tolist()]  
  
# Generate text using the trained model  
generate(model)



The generated text doesn’t look great with our basic model of around 33K parameters. However, now that we’ve laid the groundwork with this simple model, we’ll move on to constructing the LLaMA architecture in the next section.

**Replicating LLaMA Architecture**

In the earlier part of the blog, we covered essential concepts, and now, we’ll integrate these concepts into our base model. LLaMA introduces three architectural modifications to the original Transformer:

1. RMSNorm for pre-normalization
2. Rotary embeddings
3. SwiGLU activation function

We’ll incorporate each of these modifications one by one into our base model, iterating and building upon them.

**RMSNorm for pre-normalization:**

We are defining an RMSNorm function with the following functionalities:

class RMSNorm(nn.Module):  
 def \_\_init\_\_(self, layer\_shape, eps=1e-8, bias=False):  
 super(RMSNorm, self).\_\_init\_\_()  
  
 # Registering a learnable parameter 'scale' as a parameter of the module  
 self.register\_parameter("scale", nn.Parameter(torch.ones(layer\_shape)))  
  
 def forward(self, x):  
 """  
 Assumes shape is (batch, seq\_len, d\_model)  
 """  
 # Calculating the Frobenius norm, RMS = 1/sqrt(N) \* Frobenius norm  
 ff\_rms = torch.linalg.norm(x, dim=(1,2)) \* x[0].numel() \*\* -.5  
  
 # Normalizing the input tensor 'x' with respect to RMS  
 raw = x / ff\_rms.unsqueeze(-1).unsqueeze(-1)  
  
 # Scaling the normalized tensor using the learnable parameter 'scale'  
 return self.scale[:x.shape[1], :].unsqueeze(0) \* raw

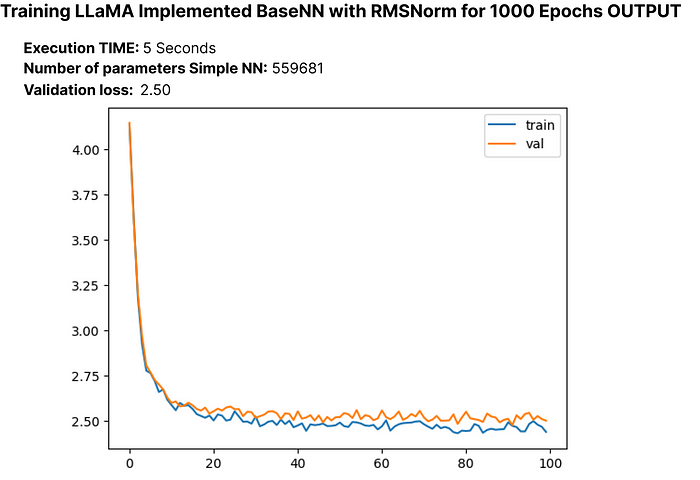
we define the RMSNorm class. During initialization, it registers a scale parameter. In the forward pass, it calculates the **Frobenius norm** of the input tensor and then normalizes the tensor. Finally, the tensor is scaled by the registered scale parameter. This function is designed for use in LLaMA to replace the LayerNorm operation.

Now it’s time to incorporate the first implementation concept of LLaMA, which is RMSNorm, into our simple NN model. Here’s the updated code:

# Define the SimpleModel\_RMS with RMSNorm  
class SimpleModel\_RMS(nn.Module):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_()  
 self.config = config  
  
 # Embedding layer to convert character indices to vectors  
 self.embedding = nn.Embedding(config['vocab\_size'], config['d\_model'])  
  
 # RMSNorm layer for pre-normalization  
 self.rms = RMSNorm((config['context\_window'], config['d\_model']))  
  
 # Linear layers for modeling relationships between features  
 self.linear = nn.Sequential(  
 # Rest of the code  
 ...  
 )  
  
 # Print the total number of model parameters  
 print("Model parameters:", sum([m.numel() for m in self.parameters()]))  
  
 def forward(self, idx, targets=None):  
 # Embedding layer converts character indices to vectors  
 x = self.embedding(idx)  
  
 # RMSNorm pre-normalization  
 x = self.rms(x)  
  
 # Linear layers for modeling relationships between features  
 logits = self.linear(x)  
  
 if targets is not None:  
  
 # Rest of the code  
 ...

Let’s execute the modified NN model with RMSNorm and observe the updated number of parameters in the model, along with the loss:

# Create an instance of SimpleModel\_RMS  
model = SimpleModel\_RMS(MASTER\_CONFIG)  
  
# Obtain batches for training  
xs, ys = get\_batches(dataset, 'train', MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'])  
  
# Calculate logits and loss using the model  
logits, loss = model(xs, ys)  
  
# Define the Adam optimizer for model parameters  
optimizer = torch.optim.Adam(model.parameters())  
  
# Train the model  
train(model, optimizer)



The validation loss experiences a small decrease, and the parameters of our updated LLM now total approximately 55,000.

**Rotary Embeddings:**

Next, we will implement rotary positional embeddings. In RoPE, the authors suggest embedding the position of a token in a sequence by rotating the embedding, applying a different rotation at each position. Let’s create a function that mimics the actual paper implementation of RoPE:

def get\_rotary\_matrix(context\_window, embedding\_dim):  
 # Initialize a tensor for the rotary matrix with zeros  
 R = torch.zeros((context\_window, embedding\_dim, embedding\_dim), requires\_grad=False)  
   
 # Loop through each position in the context window  
 for position in range(context\_window):  
 # Loop through each dimension in the embedding  
 for i in range(embedding\_dim // 2):  
 # Calculate the rotation angle (theta) based on the position and embedding dimension  
 theta = 10000. \*\* (-2. \* (i - 1) / embedding\_dim)  
 # Calculate the rotated matrix elements using sine and cosine functions  
 m\_theta = position \* theta  
 R[position, 2 \* i, 2 \* i] = np.cos(m\_theta)  
 R[position, 2 \* i, 2 \* i + 1] = -np.sin(m\_theta)  
 R[position, 2 \* i + 1, 2 \* i] = np.sin(m\_theta)  
 R[position, 2 \* i + 1, 2 \* i + 1] = np.cos(m\_theta)  
 return R

we generate a rotary matrix based on the specified context window and embedding dimension, following the proposed RoPE implementation.

As you may be familiar with the architecture of transformers, which involves attention heads, we similarly need to create attention heads when replicating LLaMA. To start, let’s first create a single **masked attention head** using the get\_rotary\_matrix function we previously developed for rotary embeddings. **Additionally, each line is commented for clarity**:

class RoPEAttentionHead(nn.Module):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_()  
 self.config = config  
 # Linear transformation for query  
 self.w\_q = nn.Linear(config['d\_model'], config['d\_model'], bias=False)  
 # Linear transformation for key  
 self.w\_k = nn.Linear(config['d\_model'], config['d\_model'], bias=False)  
 # Linear transformation for value  
 self.w\_v = nn.Linear(config['d\_model'], config['d\_model'], bias=False)  
 # Obtain rotary matrix for positional embeddings  
 self.R = get\_rotary\_matrix(config['context\_window'], config['d\_model'])  
  
 def get\_rotary\_matrix(context\_window, embedding\_dim):  
 # Generate rotational matrix for RoPE  
 R = torch.zeros((context\_window, embedding\_dim, embedding\_dim), requires\_grad=False)  
 for position in range(context\_window):  
 for i in range(embedding\_dim//2):  
   
 # Rest of the code  
 ...  
  
 return R  
  
 def forward(self, x, return\_attn\_weights=False):  
 # x: input tensor of shape (batch, sequence length, dimension)  
  
 b, m, d = x.shape # batch size, sequence length, dimension  
  
 # Linear transformations for Q, K, and V  
 q = self.w\_q(x)  
 k = self.w\_k(x)  
 v = self.w\_v(x)  
  
 # Rotate Q and K using the RoPE matrix  
 q\_rotated = (torch.bmm(q.transpose(0, 1), self.R[:m])).transpose(0, 1)  
 k\_rotated = (torch.bmm(k.transpose(0, 1), self.R[:m])).transpose(0, 1)  
  
 # Perform scaled dot-product attention  
 activations = F.scaled\_dot\_product\_attention(  
 q\_rotated, k\_rotated, v, dropout\_p=0.1, is\_causal=True  
 )  
  
 if return\_attn\_weights:  
 # Create a causal attention mask  
 attn\_mask = torch.tril(torch.ones((m, m)), diagonal=0)  
 # Calculate attention weights and add causal mask  
 attn\_weights = torch.bmm(q\_rotated, k\_rotated.transpose(1, 2)) / np.sqrt(d) + attn\_mask  
 attn\_weights = F.softmax(attn\_weights, dim=-1)  
 return activations, attn\_weights  
  
 return activations

Now that we have a single masked attention head that returns attention weights, the next step is to create a multi-Head attention mechanism.

class RoPEMaskedMultiheadAttention(nn.Module):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_()  
 self.config = config  
 # Create a list of RoPEMaskedAttentionHead instances as attention heads  
 self.heads = nn.ModuleList([  
 RoPEMaskedAttentionHead(config) for \_ in range(config['n\_heads'])  
 ])  
 self.linear = nn.Linear(config['n\_heads'] \* config['d\_model'], config['d\_model']) # Linear layer after concatenating heads  
 self.dropout = nn.Dropout(.1) # Dropout layer  
  
 def forward(self, x):  
 # x: input tensor of shape (batch, sequence length, dimension)  
  
 # Process each attention head and concatenate the results  
 heads = [h(x) for h in self.heads]  
 x = torch.cat(heads, dim=-1)  
   
 # Apply linear transformation to the concatenated output  
 x = self.linear(x)  
   
 # Apply dropout  
 x = self.dropout(x)  
 return x

The original paper used 32 heads for their smaller 7b LLM variation, but due to constraints, we’ll use 8 heads for our approach.

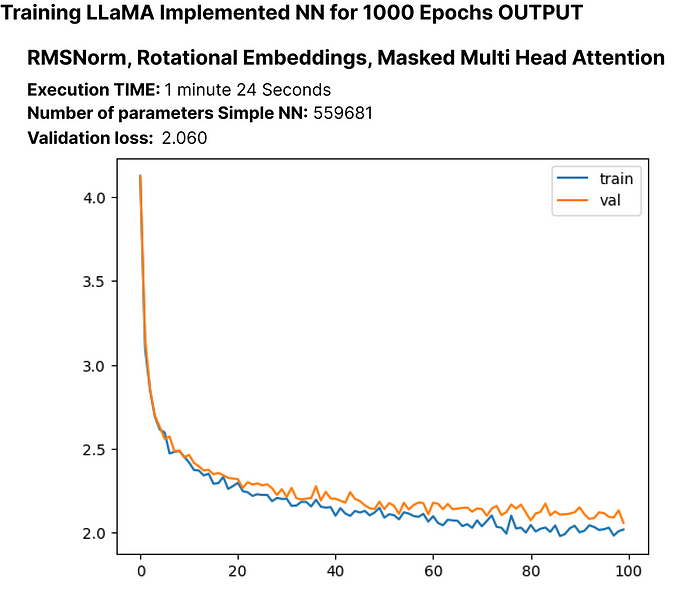
# Update the master configuration with the number of attention heads  
MASTER\_CONFIG.update({  
 'n\_heads': 8,  
})

Now that we’ve implemented Rotational Embedding and Multi-head Attention, let’s re-write our RMSNorm neural network model with the updated code. We’ll test its performance, compute the loss, and check the number of parameters. We’ll refer to this updated model as **“RopeModel”**

class RopeModel(nn.Module):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_()  
 self.config = config  
  
 # Embedding layer for input tokens  
 self.embedding = nn.Embedding(config['vocab\_size'], config['d\_model'])  
   
 # RMSNorm layer for pre-normalization  
 self.rms = RMSNorm((config['context\_window'], config['d\_model']))  
   
 # RoPEMaskedMultiheadAttention layer  
 self.rope\_attention = RoPEMaskedMultiheadAttention(config)  
  
 # Linear layer followed by ReLU activation  
 self.linear = nn.Sequential(  
 nn.Linear(config['d\_model'], config['d\_model']),  
 nn.ReLU(),  
 )  
  
 # Final linear layer for prediction  
 self.last\_linear = nn.Linear(config['d\_model'], config['vocab\_size'])  
  
 print("model params:", sum([m.numel() for m in self.parameters()]))  
  
 def forward(self, idx, targets=None):  
 # idx: input indices  
 x = self.embedding(idx)  
  
 # One block of attention  
 x = self.rms(x) # RMS pre-normalization  
 x = x + self.rope\_attention(x)  
  
 x = self.rms(x) # RMS pre-normalization  
 x = x + self.linear(x)  
  
 logits = self.last\_linear(x)  
  
 if targets is not None:  
 loss = F.cross\_entropy(logits.view(-1, self.config['vocab\_size']), targets.view(-1))  
 return logits, loss  
  
 else:  
 return logits

Let’s execute the modified NN model with RMSNorm, Rotational Embeddings and Masked Multi Head Attentions to observe the updated number of parameters in the model, along with the loss:

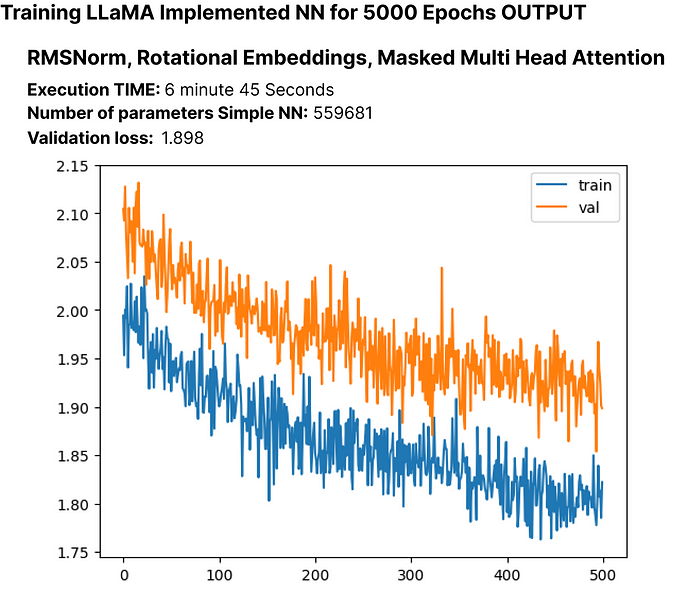
# Create an instance of RopeModel (RMSNorm, RoPE, Multi-Head)  
model = RopeModel(MASTER\_CONFIG)  
  
# Obtain batches for training  
xs, ys = get\_batches(dataset, 'train', MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'])  
  
# Calculate logits and loss using the model  
logits, loss = model(xs, ys)  
  
# Define the Adam optimizer for model parameters  
optimizer = torch.optim.Adam(model.parameters())  
  
# Train the model  
train(model, optimizer)



The validation loss experiences a small decrease again, and the parameters of our updated LLM now total approximately 55,000.

Let’s train the model for more epochs to see if the loss of our recreated LLaMA LLM continues to decrease or not.

# Updating training configuration with more epochs and a logging interval  
MASTER\_CONFIG.update({  
 "epochs": 5000,  
 "log\_interval": 10,  
})  
  
# Training the model with the updated configuration  
train(model, optimizer)



The validation loss continues to decrease, suggesting that training for more epochs could lead to further loss reduction, though not significantly.

**SwiGLU activation function:**

As mentioned before, the creators of LLaMA use SwiGLU instead of ReLU, so we’ll be implementing SwiGLU equation in our code.



<https://arxiv.org/pdf/2002.05202v1.pdf>

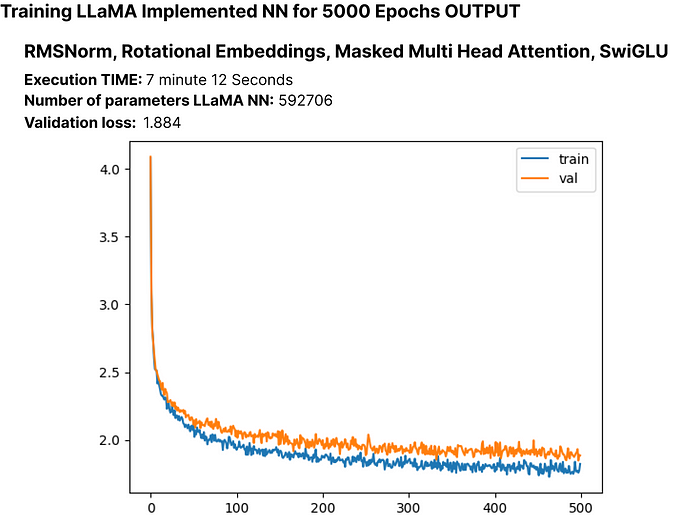
class SwiGLU(nn.Module):  
 """ Paper Link -> https://arxiv.org/pdf/2002.05202v1.pdf """  
 def \_\_init\_\_(self, size):  
 super().\_\_init\_\_()  
 self.config = config # Configuration information  
 self.linear\_gate = nn.Linear(size, size) # Linear transformation for the gating mechanism  
 self.linear = nn.Linear(size, size) # Linear transformation for the main branch  
 self.beta = torch.randn(1, requires\_grad=True) # Random initialization of the beta parameter  
  
 # Using nn.Parameter for beta to ensure it's recognized as a learnable parameter  
 self.beta = nn.Parameter(torch.ones(1))  
 self.register\_parameter("beta", self.beta)  
  
 def forward(self, x):  
 # Swish-Gated Linear Unit computation  
 swish\_gate = self.linear\_gate(x) \* torch.sigmoid(self.beta \* self.linear\_gate(x))  
 out = swish\_gate \* self.linear(x) # Element-wise multiplication of the gate and main branch  
 return out

After implementing the SwiGLU equation in python, we need to integrate it into our modified LLaMA language model (**RopeModel**).

class RopeModel(nn.Module):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_()  
 self.config = config  
  
 # Embedding layer for input tokens  
 self.embedding = nn.Embedding(config['vocab\_size'], config['d\_model'])  
   
 # RMSNorm layer for pre-normalization  
 self.rms = RMSNorm((config['context\_window'], config['d\_model']))  
   
 # Multi-head attention layer with RoPE (Rotary Positional Embeddings)  
 self.rope\_attention = RoPEMaskedMultiheadAttention(config)  
  
 # Linear layer followed by SwiGLU activation  
 self.linear = nn.Sequential(  
 nn.Linear(config['d\_model'], config['d\_model']),  
 SwiGLU(config['d\_model']), # Adding SwiGLU activation  
 )  
  
 # Output linear layer  
 self.last\_linear = nn.Linear(config['d\_model'], config['vocab\_size'])  
  
 # Printing total model parameters  
 print("model params:", sum([m.numel() for m in self.parameters()]))  
  
 def forward(self, idx, targets=None):  
 x = self.embedding(idx)  
  
 # One block of attention  
 x = self.rms(x) # RMS pre-normalization  
 x = x + self.rope\_attention(x)  
  
 x = self.rms(x) # RMS pre-normalization  
 x = x + self.linear(x) # Applying SwiGLU activation  
  
 logits = self.last\_linear(x)  
  
 if targets is not None:  
 # Calculate cross-entropy loss if targets are provided  
 loss = F.cross\_entropy(logits.view(-1, self.config['vocab\_size']), targets.view(-1))  
 return logits, loss  
  
 else:  
 return logits

Let’s execute the modified NN model with RMSNorm, Rotational Embeddings, Masked Multi Head Attentions and SwiGLU to observe the updated number of parameters in the model, along with the loss:

# Create an instance of RopeModel (RMSNorm, RoPE, Multi-Head, SwiGLU)  
model = RopeModel(MASTER\_CONFIG)  
  
# Obtain batches for training  
xs, ys = get\_batches(dataset, 'train', MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'])  
  
# Calculate logits and loss using the model  
logits, loss = model(xs, ys)  
  
# Define the Adam optimizer for model parameters  
optimizer = torch.optim.Adam(model.parameters())  
  
# Train the model  
train(model, optimizer)



Once again the validation loss experiences a small decrease, and the parameters of our updated LLM now total approximately 60,000.

So far, we have successfully implemented the key components of the paper, namely RMSNorm, RoPE, and SwiGLU. We observed that these implementations led to a minimal decrease in the loss.

Now we will add layers to our LLaMA to examine its impact on the loss. The original paper used 32 layers for the 7b version, but we will use only 4 layers. Let’s adjust our model settings accordingly.

# Update model configurations for the number of layers  
MASTER\_CONFIG.update({  
 'n\_layers': 4, # Set the number of layers to 4  
})

Let’s start by creating a single layer to understand its impact.

# add RMSNorm and residual connection  
class LlamaBlock(nn.Module):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_()  
 self.config = config  
  
 # RMSNorm layer  
 self.rms = RMSNorm((config['context\_window'], config['d\_model']))  
  
 # RoPE Masked Multihead Attention layer  
 self.attention = RoPEMaskedMultiheadAttention(config)  
  
 # Feedforward layer with SwiGLU activation  
 self.feedforward = nn.Sequential(  
 nn.Linear(config['d\_model'], config['d\_model']),  
 SwiGLU(config['d\_model']),  
 )  
  
 def forward(self, x):  
 # one block of attention  
 x = self.rms(x) # RMS pre-normalization  
 x = x + self.attention(x) # residual connection  
  
 x = self.rms(x) # RMS pre-normalization  
 x = x + self.feedforward(x) # residual connection  
 return x

Create an instance of the LlamaBlock class and applies it to a random tensor.

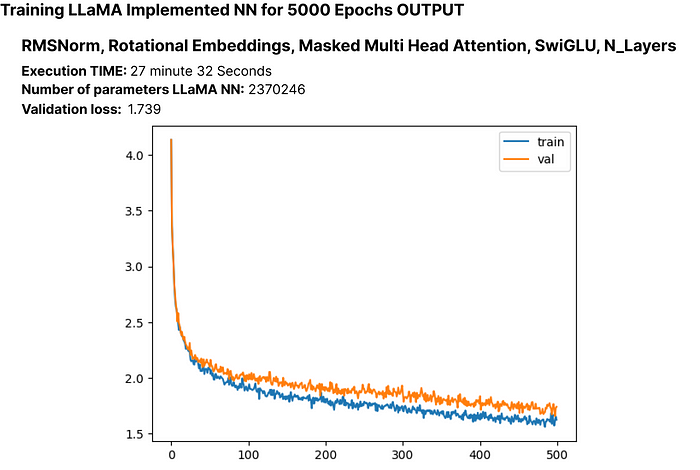
# Create an instance of the LlamaBlock class with the provided configuration  
block = LlamaBlock(MASTER\_CONFIG)  
  
# Generate a random tensor with the specified batch size, context window, and model dimension  
random\_input = torch.randn(MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'], MASTER\_CONFIG['d\_model'])  
  
# Apply the LlamaBlock to the random input tensor  
output = block(random\_input)

Having successfully created a single layer, we can now use it to construct multiple layers. Additionally, we will rename our model class from **“ropemodel”** to **“Llama”** as we have replicated every component of the LLaMA language model.

class Llama(nn.Module):  
 def \_\_init\_\_(self, config):  
 super().\_\_init\_\_()  
 self.config = config  
 # Embedding layer for token representations  
 self.embeddings = nn.Embedding(config['vocab\_size'], config['d\_model'])  
 # Sequential block of LlamaBlocks based on the specified number of layers  
 self.llama\_blocks = nn.Sequential(  
 OrderedDict([(f"llama\_{i}", LlamaBlock(config)) for i in range(config['n\_layers'])])  
 )  
 # Feedforward network (FFN) for final output  
 self.ffn = nn.Sequential(  
 nn.Linear(config['d\_model'], config['d\_model']),  
 SwiGLU(config['d\_model']),  
 nn.Linear(config['d\_model'], config['vocab\_size']),  
 )  
  
 # Print total number of parameters in the model  
 print("model params:", sum([m.numel() for m in self.parameters()]))  
  
 def forward(self, idx, targets=None):  
 # Input token indices are passed through the embedding layer  
 x = self.embeddings(idx)  
 # Process the input through the LlamaBlocks  
 x = self.llama\_blocks(x)  
 # Pass the processed input through the final FFN for output logits  
 logits = self.ffn(x)  
  
 # If targets are not provided, return only the logits  
 if targets is None:  
 return logits  
 # If targets are provided, compute and return the cross-entropy loss  
 else:  
 loss = F.cross\_entropy(logits.view(-1, self.config['vocab\_size']), targets.view(-1))  
 return logits, loss

Let’s execute the modified LLaMA model with RMSNorm, Rotational Embeddings, Masked Multi Head Attentions, SwiGLU and N\_layers to observe the updated number of parameters in the model, along with the loss:

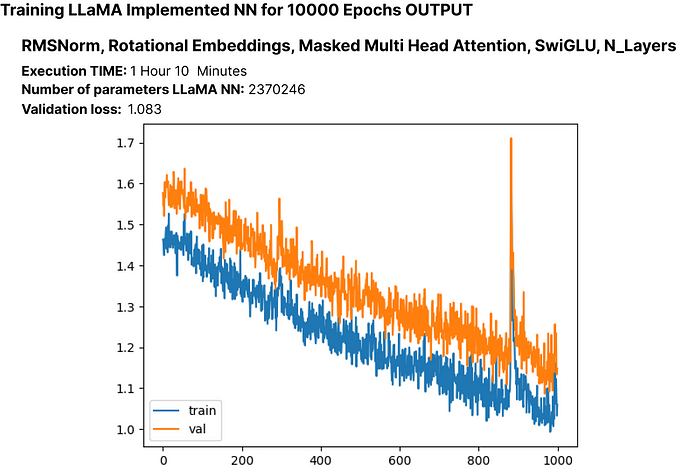
# Create an instance of RopeModel (RMSNorm, RoPE, Multi-Head, SwiGLU, N\_layers)  
llama = Llama(MASTER\_CONFIG)  
  
# Obtain batches for training  
xs, ys = get\_batches(dataset, 'train', MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'])  
  
# Calculate logits and loss using the model  
logits, loss = llama(xs, ys)  
  
# Define the Adam optimizer for model parameters  
optimizer = torch.optim.Adam(llama.parameters())  
  
# Train the model  
train(llama, optimizer)



While there’s a possibility of overfitting, it’s crucial to explore whether extending the number of epochs leads to a further reduction in loss. Additionally, note that our current LLM has over 2 million parameters.

Let’s train it for higher number of epochs.

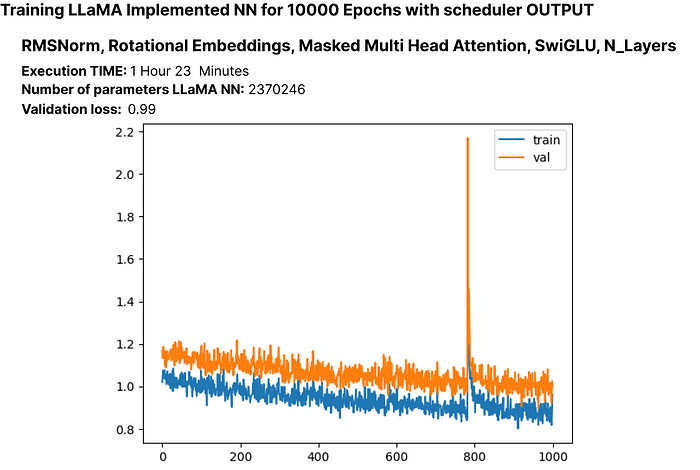
# Update the number of epochs in the configuration  
MASTER\_CONFIG.update({  
 'epochs': 10000,  
})  
# Train the LLaMA model for the specified number of epochs  
train(llama, optimizer, scheduler=None, config=MASTER\_CONFIG)



The loss here is 1.08, we can achieve even more lower loss without encountering significant overfitting. This suggests the model is performing well.

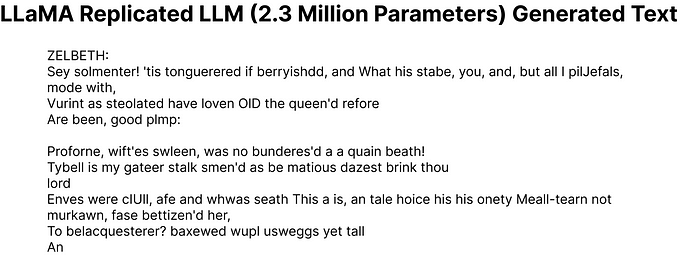
Let’s train the model once more, this time incorporating a scheduler

# Training the model again, scheduler for better optimization.  
train(llama, optimizer, config=MASTER\_CONFIG)



Up until now, we’ve successfully implemented a scaled-down version of the LLaMA architecture on our custom dataset. Now, let’s examine the generated output from our 2 million-parameter Language Model.

# Generate text using the trained LLM (llama) with a maximum of 500 tokens  
generated\_text = generate(llama, MASTER\_CONFIG, 500)[0]  
print(generated\_text)



Even though some generated words may not be perfect English, our LLM with just 2 million parameters has shown a basic understanding of the English language.

Now, let’s see how well our model performs on the test set.

# Get batches from the test set  
xs, ys = get\_batches(dataset, 'test', MASTER\_CONFIG['batch\_size'], MASTER\_CONFIG['context\_window'])  
  
# Pass the test data through the LLaMA model  
logits, loss = llama(xs, ys)  
  
# Print the loss on the test set  
print(loss)

The computed loss on the test set is approximately 1.236.

A simple way to check for changes in the generated output is to run training for a large number of epochs and observe the results.

**Experimenting with hyperparameters**

Hyperparameter tuning is a crucial step in training neural networks. In the original Llama paper, the authors utilized the Cosine Annealing learning schedule. However, in our experimentation, it didn’t perform well. Here’s an example of experimenting with hyperparameters using a different learning schedule:

# Update configuration  
MASTER\_CONFIG.update({  
 "epochs": 1000  
})  
  
# Create Llama model with Cosine Annealing learning schedule  
llama\_with\_cosine = Llama(MASTER\_CONFIG)  
  
# Define Adam optimizer with specific hyperparameters  
llama\_optimizer = torch.optim.Adam(  
 llama.parameters(),  
 betas=(.9, .95),  
 weight\_decay=.1,  
 eps=1e-9,  
 lr=1e-3  
)  
  
# Define Cosine Annealing learning rate scheduler  
scheduler = torch.optim.lr\_scheduler.CosineAnnealingLR(llama\_optimizer, 300, eta\_min=1e-5)  
  
# Train the Llama model with the specified optimizer and scheduler  
train(llama\_with\_cosine, llama\_optimizer, scheduler=scheduler)

**Saving Your Language Model (LLM)**

You can save your entire LLM or just the parameters using the following:

# Save the entire model  
torch.save(llama, 'llama\_model.pth')  
  
# If you want to save only the model parameters  
torch.save(llama.state\_dict(), 'llama\_model\_params.pth')

To save your PyTorch model for Hugging Face’s Transformers library, you can use the save\_pretrained method. Here's an example:

from transformers import GPT2LMHeadModel, GPT2Config  
  
# Assuming Llama is your PyTorch model  
llama\_config = GPT2Config.from\_dict(MASTER\_CONFIG)  
llama\_transformers = GPT2LMHeadModel(config=llama\_config)  
llama\_transformers.load\_state\_dict(llama.state\_dict())  
  
# Specify the directory where you want to save the model  
output\_dir = "llama\_model\_transformers"  
  
# Save the model and configuration  
llama\_transformers.save\_pretrained(output\_dir)

GPT2Config is used to create a configuration object compatible with GPT-2. Then, a GPT2LMHeadModel is created and loaded with the weights from your Llama model. Finally, save\_pretrained is called to save both the model and configuration in the specified directory.

You can then load the model using the Transformers library:

from transformers import GPT2LMHeadModel, GPT2Config  
  
# Specify the directory where the model was saved  
output\_dir = "llama\_model\_transformers"  
  
# Load the model and configuration  
llama\_transformers = GPT2LMHeadModel.from\_pretrained(output\_dir)

**Conclusion**

In this blog, we’ve walked through a step-by-step process on how to implement the LLaMA approach to build your own small Language Model (LLM). As a suggestion, consider expanding your model to around 15 million parameters, as smaller models in the range of 10M to 20M tend to comprehend English better. Once your LLM becomes proficient in language, you can fine-tune it for specific use cases.

I hope this comprehensive blog has provided you with insights on replicating a paper to create your personalized LLM.

Thanks for reading this extensive post!