LLM Learning

A document explain PyTorch Code

import **torch**

import **torch**.**nn** as **nn**

import **torch**.**nn**.**functional** as **F**

*# Tiny training data (you could replace with your own text)*

text = "hello world"

*# Build vocabulary*

chars = **sorted**(**list**(**set**(text)))

stoi = {ch: i for i, ch in **enumerate**(chars)} *# char → index*

itos = {i: ch for ch, i in stoi.**items**()} *# index → char*

vocab\_size = **len**(chars)

1. **Tokenization**
   * Breaking text into **tokens** (characters for now).
   * stoi → maps tokens to numbers (like giving each word a “name tag” so the model can understand it).
   * itos → reverse map (turn numbers back into tokens).

**Analogy:** You’re at the **mailroom** of the LLM factory. You assign a number to every letter/word so the machines know what they’re handling.

*# Encode & decode helpers*

encode = lambda s: [stoi[c] for c in s]

decode = lambda l: ''.**join**([itos[i] for i in l])

1. **Encoding / Decoding**
   * encode(text) → [numbers] (text → token IDs)
   * decode(ids) → text (token IDs → text)

**Analogy:** You now have a **translator** that converts human language into machine language and back.

data = **torch**.**tensor**(encode(text), dtype=**torch**.long)

1. **Tensors**
   * Token IDs → torch.Tensor
   * This is what the network actually computes with.

**Analogy:** You’ve packaged the mail into **containers** that the assembly line (neural network) can process efficiently.

*# Hyperparameters*

embed\_dim = 16

context\_size = 4 *# how many characters to look back*

num\_heads = 2

num\_layers = 1

*# Simple Transformer Block*

class **TransformerBlock**(**nn**.**Module**):

def **\_\_init\_\_**(self, embed\_dim, num\_heads):

**super**().**\_\_init\_\_**()

self.attn = **nn**.**MultiheadAttention**(embed\_dim, num\_heads, batch\_first=True)

**4. Transformer Block**

**Self-Attention (nn.MultiheadAttention)** → lets tokens **talk to each other** to gather context.

self.ff = **nn**.**Sequential**(

**nn**.**Linear**(embed\_dim, embed\_dim \* 4),

**nn**.**ReLU**(),

**nn**.**Linear**(embed\_dim \* 4, embed\_dim),

)

**Feedforward (nn.Sequential)** → processes each token’s info nonlinearly, like a **processing station** in the factory.

self.ln1 = **nn**.**LayerNorm**(embed\_dim)

self.ln2 = **nn**.**LayerNorm**(embed\_dim)

**LayerNorm (nn.LayerNorm)** → keeps each token vector **stable and well-scaled**, like a **quality control step**.

def **forward**(self, x):

attn\_out, \_ = self.attn(x, x, x, need\_weights=False)

x = self.ln1(x + attn\_out)

ff\_out = self.ff(x)

x = self.ln2(x + ff\_out)

return x

**Residuals (x + F(x))** → lets the token **retain its original info while learning tweaks**, like adding **reinforcements to old plans instead of replacing them entirely**.

**Analogy:** You’ve built the **assembly line for one processing layer**:

* Tokens come in
* Talk to each other (attention/context mixing)
* Get refined (feedforward)
* Standardized (LayerNorm)
* Keep some original info (residuals)

### **What this means**

* At this point, your LLM **can start learning relationships between tokens**.
* Each TransformerBlock is like **a single step in a brain**: it looks at the surrounding context, updates its understanding, and passes the result forward.
* If you stack multiple blocks, your tokens can **understand longer-range relationships** (like connecting ideas across a whole sentence).

### **Analogy for the whole LLM so far**

Imagine your LLM is a **team of editors rewriting a book**:

1. **Tokenization / encode** → everyone gets a list of characters/words to work with, each with a number on it.
2. **Tensors** → the manuscript is now in a format editors can process (digital sheets, not paper).
3. **TransformerBlock** → each editor reads the text, looks at context (attention), makes suggestions (feedforward), and writes back (residual + LayerNorm).
4. **Decoding / itos** → after the editors finish, you can turn the numbers back into readable text.

Right now, your “team” is **ready to start learning patterns**, but you haven’t yet trained it or generated anything.

*# Tiny LLM*

class **TinyLLM**(**nn**.**Module**):

def **\_\_init\_\_**(self, vocab\_size, embed\_dim, context\_size, num\_heads):

**super**().**\_\_init\_\_**()

self.token\_embed = **nn**.**Embedding**(vocab\_size, embed\_dim) *#maps token ids to vectors..... vocab\_size X embed\_dim*

self.pos\_embed = **nn**.**Embedding**(context\_size, embed\_dim) *#learned positional embeddings for the shape of context\_size and embed\_dim*

self.transformer = **TransformerBlock**(embed\_dim, num\_heads) *#Exciting! We call our TransformerBlock Module*

self.lm\_head = **nn**.**Linear**(embed\_dim, vocab\_size) *#Linear is a fully connected layer, with an input vector length of embed\_dim, and adds weight matrix, bias vector: y = xW^T + b*

def **forward**(self, idx):

B, T = idx.shape *#Batch Size and Sequence Length*

tok\_emb = self.token\_embed(idx) *#remember, maps the token id, to the vectors (B, T, E) <- E is the size of the vector representing each token in the model. Like the resolution of the token's brain.*

pos\_emb = self.pos\_embed(**torch**.**arange**(T, device = idx.device)) *#this returns T, E*

x = tok\_emb + pos\_emb

x = self.transformer(x)

logits = self.lm\_head(x)

return logits

*#Positional Embedding size is limited by context\_size. Context\_size must be larger than the max T you ever pass*

*#Weight Tying - best practice to tie token\_embed.weight and lm\_head.weight ..... (use the same matrix)*

5. TinyLLM class explained

1. \_\_init\_\_ – building the model

Maps **token IDs → vectors of size E** (embed\_dim)

Each token gets a “brain vector” the model can work with

Shape: [vocab\_size, embed\_dim]

Learned positional embeddings

Adds information about token position in the sequence

Without this, attention wouldn’t know token order

Shape: [context\_size, embed\_dim]

**Calls the TransformerBlock you built in step 4  
 Handles context mixing + feedforward + residuals**

**Maps the final token embeddings to logits over the vocabulary**

**Shape: [B, T, vocab\_size]**

2. forward – how data flows

idx = input token IDs (batch of sequences)

Converts token IDs → embeddings

Shape: [B, T, E]

Adds **positional info**

Shape: [T, E] → broadcasted across batch

Combine **token + position embeddings**

Shape: [B, T, E]

Tokens pass through TransformerBlock

Output: updated context-aware embeddings [B, T, E]

Linear layer converts embeddings → scores for next-token prediction  
 Shape: [B, T, vocab\_size]

These logits are what you feed to **cross-entropy loss** during training

### **Analogy**

1. **Input tokens** = sticky notes with ID numbers
2. **Token embeddings** = turn sticky notes into “brain vectors”  
   **Positional embeddings** = add a tag showing where each note is in the sequence
3. **TransformerBlock** = team of editors read all notes, mix context, refine ideas
4. **LM head** = editors vote on which token comes next  
   **Output logits** = prediction scores over all possible tokens

*#Model Instantiation and Optimizer*

model = **TinyLLM**(vocab\_size, embed\_dim, context\_size, num\_heads) *#Call the class, using previously defined params*

optimizer = **torch**.**optim**.**AdamW**(model.**parameters**(), lr=1e-2) *#Adam with decoupled weight delay..... common optimizer for transformers. The param lr is very large for anything but toy problems..... you'd want a much smaller LR for real training.....*

*#An optimizer is part of the Model Training Pipeline that updates the model's params so the model learns from data. IE we start with random weights, and they tweak the weights with the optimizer to reduce prediction loss*

*#LR is "Learning Rate" - controls how big each step is when updating weights. Smaller is more accurate, but slower*

*#Moving model to GPU if available*

device = **torch**.**device**("cuda" if **torch**.**cuda**.**is\_available**() else "cpu"); model.**to**(device)

## **Step 6: Model instantiation (TinyLLM + optimizer)**

* **What you did:**
  + Instantiated the model with your parameters:  
    - vocab\_size → how many unique tokens you have
    - embed\_dim → size of each token vector (E)
    - context\_size → maximum sequence length
    - num\_heads → number of attention heads
  + Added an **optimizer (AdamW)** with a learning rate (lr) for training.
* **Purpose:**
  + Now you have a **ready-to-train model** with:  
    - Embeddings for tokens
    - One TransformerBlock
    - Output projection (lm\_head) for predicting next tokens
    - Optimizer for adjusting weights during training
* **Analogy:**
  + You’ve **assembled the editing workshop**:  
    - Input text comes in → encoded to IDs → converted to vectors → passes through editors (TransformerBlock) → generates output → optimizer acts as the **coach** telling the editors how to improve next round.
* **Where you are conceptually:**
  + Your LLM is **built but untrained**.
  + Tokens can flow through embeddings and one TransformerBlock, but the model **doesn’t know anything yet** — it just has the machinery to start learning.

*#TRAIN OUR MODEL!!!*

for epoch in **range**(200): *#train 200 times*

*#get random batch for contiguous slice of length context size.*

start = **torch**.**randint**(0, **len**(data)-context\_size, (1,))

*#input sequence of length context\_size, shape after batch size 1?*

xb = data[start : start **+** context\_size].**unsqueeze**(0) *#input*

*#implementing next-token prediction for each position*

yb = data[start **+** 1 : start **+** context\_size **+** 1].**unsqueeze**(0) *#target*

logits = model(xb) *#feeding data*

*#flatten the batch and time dimensions. cross entropy computes the average loss over these positions*

loss = **F**.**cross\_entropy**(logits.view(-1, vocab\_size), yb.**view**(-1))

*#these update the weights*

optimizer.**zero\_grad**()

loss.**backward**()

optimizer.**step**()

if epoch % 50 == 0:

**print**(f"Epoch {epoch}, loss {loss.**item**():.4f}")

7. Training Loop

**Critical gotcha**

* **Causal masking is missing during training.** Because our TransformerBlock did not use a causal mask, the attention could see future tokens in the slice. If the model can access future tokens during the forward pass, it can trivially copy the next character and the training signal becomes unrealistic. This leads to a “leaky” model that won’t generalize. To fix: add the causal mask inside the transformer block (see earlier) so attention at position t only sees positions <= t.

**Other tips**

* Use batches (B>1) to stabilize training.
* Shuffle and sample diverse sequences; with only "hello world" there’s almost nothing to learn.
* Lower the learning rate for stable training.

Think of your model as a student preparing for a spelling bee:

1. **Forward pass (prediction):** The student looks at part of a word ("hel") and *guesses* the next letter ("p").
2. **Loss calculation (grading):** The teacher checks if the guess is right. Wrong guess? Red mark.  
    The more wrong, the bigger the penalty (loss).
3. **Backward pass (blame assignment):** The student traces back: *“Why did I guess P instead of L? Maybe I was paying too much attention to the first letter, not enough to the second.”* This is backpropagation: figuring out which part of the brain (weights) caused the mistake.
4. **Optimizer step (studying correction):** The student adjusts their memory slightly so next time, they lean a little more towards the correct answer.
5. **Repeat many times:** Like a kid drilling flashcards for hours until their mistakes become rare.

*# Text generation*

context = **torch**.**tensor**([[stoi['h'], stoi['e'], stoi['l'], stoi['l']]], dtype=**torch**.long)

for \_ in **range**(20):

logits = model(context)

probs = **F**.**softmax**(logits[:, -1, :], dim=-1)

next\_id = **torch**.**multinomial**(probs, num\_samples=1)

context = **torch**.**cat**([context, next\_id], dim=1)

**print**("Generated:", decode(context[0].**tolist**()))

8) Text generation

**Sampling choices**

* torch.multinomial(probs, 1) is *stochastic sampling* (good for variety).
* argmax picks the most likely token (greedy) — deterministic, but can be repetitive.
* You can use **temperature** to control randomness
* **Top-k / nucleus (top-p)** sampling are common ways to avoid sampling from very-low-probability tokens.

**probs = F.softmax(logits[:,-1,:], dim=-1)**

A. Converts logits (raw scores) into probabilities.

Ensures they sum to 1.

Example: [0.01, 0.05, 0.9, 0.04].

👉 Analogy: The student converts their confidence scores into percentages:

90% sure it’s "o"

5% chance "a"

4% chance "l"

1% chance "x"

B. Sampling

* Randomly picks the next token according to the probability distribution.
* Not always the top choice — allows creativity and variety

👉 Analogy: Instead of always picking their #1 guess, the student rolls a weighted dice:

* "o" has 90 sides
* "a" has 5 sides
* "l" has 4 sides
* "x" has 1 side  
   They roll → the outcome decides the next letter.

C. Append Prediction

D. Repeat (N Steps)

# **9) Concise list of practical fixes & improvements**

1. **Add causal mask** in TransformerBlock.forward.
2. **During generation** pass only the last context\_size tokens to model.
3. **Tie weights** lm\_head.weight = token\_embed.weight.
4. **Use better tokenization** (BPE/WordPiece) for real text.
5. **Bigger data & batches** — "hello world" is tiny; use more text and batch sampling.
6. **Lower LR** (e.g., 1e-4) and consider a scheduler (warmup + decay).  
   **Use GPU** (model.to('cuda'), move tensors to same device).
7. **Masking dtype** — if attn\_mask errors, convert to float and use -1e9 for masked positions.