# LLM from zero to Hero

Plan:

1. Build a basic LLM without prebuilt layers or the minimum necessary.

## Basic LLM

1. Get access to data
2. Tokenization

```
!wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
```

```
--2025-07-28 13:49:13--  https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ..
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1115394 (1.1M) [text/plain]
Saving to: ‘input.txt’

input.txt           100%[===================>]   1.06M  --.-KB/s    in 0.03s

2025-07-28 13:49:13 (30.7 MB/s) - ‘input.txt’ saved [1115394/1115394]
```

```
%load_ext tensorboard
```

```
import string
import torch
import math
import os
import numpy as np
import time
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter
from torch.optim.lr_scheduler import LambdaLR
from collections import Counter
from torch.profiler import profile, record_function, ProfilerActivity, schedule
import csv
import time
```

```
LOWER_CASE = True
seq_length = 64
batch_size = 256
embed_dim = 256
num_epochs = 4
num_heads = 1
num_att_layers = 1
```

```
f = open("input.txt")
data = f.read()
```

## Functions and classes

```
def BPE_vocab(data):
  punctuation_set = set(string.punctuation)
  data_sep = word_separator(data=data.lower(), special_char = punctuation_set)
  data_word_char = [list(word) + ["</w>"] for word in data_sep]
  vocab = Counter(tuple(word) for word in data_word_char)
  pairs = Counter()
  for word, freq in vocab.items():
    for i in range(len(word) - 1):
      pair = (word[i], word[i+1])
      pairs[pair] += freq
  vocab = [''.join(word) for word, _ in pairs.most_common(100)] + list(set(data.lower())) + ["</w>"]
  return vocab
```

```
def BPE_enc(data, vocab):
```

```
  i = 0
  data_token = []
  while i < len(data):
    if i + 2 <= len(data) and data[i:i+2] in vocab:
      token = data[i:i+2]
      if token == '\n' and data_token and data_token[-1] == '\n':
        pass
      else:
        data_token.append(token)
      i += 2
    else:
      token = data[i:i+1]
      if token == '\n' and data_token and data_token[-1] == '\n':
        pass
      else:
        data_token.append(token)
      i += 1
  return data_token


def word_separator(data: str, special_char: list[str]) -> list[str]:
  """
  Separate text to words for tokenization
  Args:
    data: text
    speacial_char: special chatacters to break word
  Return:
    list[str]: list of words and special characters
  """
  data_separated = []
  word = ""
  for char in data:
    if char == " " or char == "\n" or char == "\t":
      if word != "":
        data_separated.append(word)
        word = ""
    elif char in special_char:
      data_separated.append(word)
      data_separated.append(char)
    else:
      word += char
  return data_separated


from torch.utils.data import Dataset

class TokenDataset(Dataset):
  def __init__(self, tokens, seq_len):
      self.tokens = tokens      # list or tensor of token IDs
      self.seq_len = seq_len    # length of input sequence

  def __len__(self):
      return len(self.tokens) - self.seq_len

  def __getitem__(self, idx):
      x = self.tokens[idx : idx + self.seq_len]
      y = self.tokens[idx + 1 : idx + self.seq_len + 1]
      return x, y


class PositionalEncoding(nn.Module):
  def __init__(self, d_model, max_len=5000):
    super(PositionalEncoding, self).__init__()

    # Create matrix of shape (max_len, d_model)
    pe = torch.zeros(max_len, d_model)
    position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)  # (max_len, 1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))

    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)

    pe = pe.unsqueeze(0)

    # Register as buffer so it's not a parameter, but saved with the model
    self.register_buffer('pe', pe)

  def forward(self, x):
    """
    x: (batch_size, seq_len, d_model)
    """
    seq_len = x.size(1)
    # Add positional encoding
```

```
        x = x + self.pe[:, :seq_len]
        return x


class Attention(torch.nn.Module):
    def __init__(self, embedding_dim, seq_length):
        super(Attention, self).__init__()
        self.softmax = nn.Softmax(-1)
        self.register_buffer('causal', torch.tril(torch.ones(seq_length, seq_length)))

    def forward(self, q, k, v):
        B, T, _ = q.shape
        x = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(q.size(-1))
        causal_mask = self.causal[:T, :T]
        x = x.masked_fill(causal_mask == 0, float('-inf'))
        x = self.softmax(x)
        x = torch.matmul(x, v)
        return x



class MultiHeadAttention(torch.nn.Module):
    def __init__(self, embedding_dim, num_heads, seq_length):
        super(MultiHeadAttention, self).__init__()
        assert embedding_dim % num_heads == 0, "embedding_dim must be divisible by num_heads"

        self.attention = Attention(embedding_dim, seq_length)

        self.num_heads = num_heads
        self.head_dim = embedding_dim // num_heads

        # Final projection after concatenating heads
        self.out_proj = nn.Linear(embedding_dim, embedding_dim)

        self.q_proj = nn.Linear(embedding_dim, embedding_dim)
        self.k_proj = nn.Linear(embedding_dim, embedding_dim)
        self.v_proj = nn.Linear(embedding_dim, embedding_dim)

    def forward(self, x):
        B, T, C = x.shape # batch_size, seq_length, embed_size

        q = self.q_proj(x)
        k = self.k_proj(x)
        v = self.v_proj(x)

        q = q.view(B, T, self.num_heads, self.head_dim).transpose(1, 2)
        k = k.view(B, T, self.num_heads, self.head_dim).transpose(1, 2)
        v = v.view(B, T, self.num_heads, self.head_dim).transpose(1, 2)

        # We need to merge batch and heads to call your single-head Attention:
        q = q.reshape(B * self.num_heads, T, self.head_dim)
        k = k.reshape(B * self.num_heads, T, self.head_dim)
        v = v.reshape(B * self.num_heads, T, self.head_dim)

        out = self.attention(q, k, v)

        out = out.view(B, self.num_heads, T, self.head_dim).transpose(1, 2)

        out = out.reshape(B, T, C)

        return self.out_proj(out)


class DecoderLayer(torch.nn.Module):
    def __init__(self, embedding_dim, num_head, seq_length):
        super(DecoderLayer, self).__init__()

        self.ff_1 = nn.Linear(embedding_dim, 4*embedding_dim)
        self.ff_2 = nn.Linear(4*embedding_dim, embedding_dim)
        self.m_head_att = MultiHeadAttention(embedding_dim, num_head, seq_length)
        self.layer_norm_1 = nn.LayerNorm(embedding_dim)
        self.layer_norm_2 = nn.LayerNorm(embedding_dim)

        # Activation
        self.relu = nn.ReLU()

    def forward(self, x):
        x_1 = self.m_head_att(x)
        x_2 = self.layer_norm_1(x_1 + x)
        x_3 = self.ff_1(x_2)
        x_4 = self.relu(x_3)
        x_5 = self.ff_2(x_4)
```

```
    x_6 = self.layer_norm_2(x_5 + x_2)
    return x_6


class DecoderOnlySmall(torch.nn.Module):
  def __init__(self, vocab_len, embedding_dim, num_head, seq_length, att_lay):
    super(DecoderOnlySmall, self).__init__()

    self.embed = nn.Embedding(vocab_len, embedding_dim)
    self.linear = nn.Linear(embedding_dim, vocab_len)
    self.DecLayer = DecoderLayer(embedding_dim, num_head, seq_length)

    # Activation
    self.softmax = nn.Softmax(dim=-1)
    self.relu = nn.ReLU()

    # weight tying
    self.linear.weight = self.embed.weight

    # postional encoding
    self.pos_encoding = PositionalEncoding(embedding_dim, max_len=seq_length)

    # Attention laywrs
    self.layers = nn.ModuleList([DecoderLayer(embedding_dim, num_head, seq_length) for _ in range(att_lay)])

  def forward(self, x):
    x_embed = self.embed(x)
    x_embed = self.pos_encoding(x_embed)

    # Decoder layers
    for layer in self.layers:
      x_embed = layer(x_embed)

    x_2 = self.linear(x_embed)
    if self.training:
      return x_2
    else:
      x_3 = self.softmax(x_2)
      return x_3


def train_epoch(epoch_index, tb_writer, training_loader, loss_fn, model, optimizer, scheduler, device):
  running_loss = 0.
  total_batches = 0
  epoch_times = []
  epoch_start = time.time()
  for i, data in enumerate(training_loader):
    inputs, labels = data
    inputs, labels = inputs.to(device), labels.to(device)

    optimizer.zero_grad()
    outputs = model(inputs)
    logits = outputs.view(-1, outputs.size(-1))
    labels = labels.view(-1)
    loss = loss_fn(logits, labels)
    loss.backward()

    optimizer.step()
    scheduler.step()

    running_loss += loss.item()
    total_batches += 1

    if i % 100 == 99:
      avg_loss_so_far = running_loss / total_batches
      print(f'  batch {i+1} loss (avg so far): {avg_loss_so_far:.4f}')
      tb_x = epoch_index * len(training_loader) + i + 1
      tb_writer.add_scalar('Loss/train', avg_loss_so_far, tb_x)

  epoch_time = time.time() - epoch_start
  mean_loss = running_loss / total_batches

  print(f"\nEpoch {epoch_index + 1} finished | Loss={mean_loss:.4f} | Time={epoch_time:.2f}s")

  return mean_loss, epoch_time


def val_epoch(val_loader, tb_writer, model, loss_fn, epoch, device):
    running_loss = 0.
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
```

```
            loss = loss_fn(outputs.view(-1, outputs.size(-1)), labels.view(-1))
            running_loss += loss.item()
    mean_val_loss = running_loss / len(val_loader)
    tb_writer.add_scalar('Loss/val', mean_val_loss, epoch)
    return mean_val_loss


def get_lr_scheduler(optimizer, warmup_steps, total_steps):
  def lr_lambda(current_step):
    if current_step < warmup_steps:
        # linear warmup from 0 to 1
        return float(current_step) / float(max(1, warmup_steps))
    # cosine decay after warmup
    progress = float(current_step - warmup_steps) / float(max(1, total_steps - warmup_steps))
    return 0.5 * (1.0 + math.cos(math.pi * progress))
  return LambdaLR(optimizer, lr_lambda)
```

## Data Preprocessing

```
word_token = False
char_token = False
BPE = True


if LOWER_CASE:
  data = data.lower()
if word_token:
  punctuation_set = set(string.punctuation) # special characters
  data_sep = word_separator(data=data, special_char = punctuation_set)
  vocab = set(data_sep)
if BPE:
  vocab = BPE_vocab(data)
if char_token:
  data_sep = list(data)
  vocab = set(data_sep)

# Vocabulary
word2index = {word: i for i, word in enumerate(sorted(vocab))}
index2word = {i: word for word, i in word2index.items()}

if BPE:
  data_sep = BPE_enc(data, vocab)

# Tokenization
data_token = []
data_token = [word2index[word] for word in data_sep]

# Data Splitting
train = data_token[:int(len(data_token)*0.80)]
val = data_token[int(len(data_token)*0.80):int(len(data_token)*0.90)]
test = data_token[int(len(data_token)*0.90):]

# Create datasets
train_dataset = TokenDataset(torch.tensor(train, dtype=torch.long), seq_length)
val_dataset = TokenDataset(torch.tensor(val, dtype=torch.long), seq_length)
test_dataset = TokenDataset(torch.tensor(test, dtype=torch.long), seq_length)

# Wrap in dataloaders
train_loader= DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=2, pin_memory=True)
val_loader= DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=2, pin_memory=True)
test_loader= DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=2, pin_memory=True)


print(len(vocab))
```

```
140
```

```
for inputs, labels in train_loader:
  print([index2word[inp.item()] for inp in inputs[0]])
  print([index2word[inp.item()] for inp in labels[0]])
  break
```

```
['ne', 'w', 's', ' ', 'we', ' ', 'he', 'ar', '\n', 'is', ' ', 'th', 'at', ' ', 'th', 'e', ' ', 're', 'be', 'l', 's', ' '
['w', 's', ' ', 'we', ' ', 'he', 'ar', '\n', 'is', ' ', 'th', 'at', ' ', 'th', 'e', ' ', 're', 'be', 'l', 's', ' ', 'ha'
```

```
from collections import Counter

token_counts = Counter(data_sep)
most_common = token_counts.most_common(20)  # top 20 tokens
```

```
for token, count in most_common:
    print(f"Token: '{token}'  Count: {count}")
```

```
Token: ' '  Count: 169892
Token: '
'  Count: 32777
Token: 'd'  Count: 24240
Token: 'e'  Count: 23966
Token: 'th'  Count: 23264
Token: 's'  Count: 20161
Token: ','  Count: 19846
Token: 'a'  Count: 15161
Token: 'y'  Count: 15154
Token: 't'  Count: 14647
Token: 'r'  Count: 14359
Token: 'u'  Count: 14054
Token: 'm'  Count: 13951
Token: 'g'  Count: 13830
Token: 'i'  Count: 12680
Token: 'o'  Count: 11432
Token: 'in'  Count: 10337
Token: ':'  Count: 10316
Token: 'an'  Count: 10212
Token: 'p'  Count: 9974
```

## Model Training

```
save_dir = "checkpoints"
os.makedirs(save_dir, exist_ok=True)
os.makedirs("log_profiler", exist_ok=True)
tinymodel = DecoderOnlySmall(len(vocab), embed_dim, num_heads, seq_length, num_att_layers)
```

```
model_parameters = filter(lambda p: p.requires_grad, tinymodel.parameters())
params = sum([np.prod(p.size()) for p in model_parameters])
print(params)
```

```
1615500
```

```
custom_run_name = f"run_bs{batch_size}_seq{seq_length}_embed{embed_dim}_{int(time.time())}"
tb_writer = SummaryWriter(log_dir=f"runs/{custom_run_name}")
```

```
base_lr = 0.001
warmup_steps = 100
total_steps = len(train_loader) * num_epochs
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
tinymodel.to(device)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(tinymodel.parameters(), lr=base_lr)
scheduler = get_lr_scheduler(optimizer, warmup_steps, total_steps)
best_val_loss = float("inf")
epochs_times = []

tinymodel.eval()
with torch.no_grad():
  val_loss = val_epoch(val_loader=val_loader,
                       tb_writer=tb_writer,
                       model=tinymodel,
                       loss_fn=loss_fn,
                       epoch=0,
                       device=device)
  print(f"Epoch {0}: Val Loss {val_loss}")

for epoch in range(num_epochs):
  tinymodel.train()
  train_loss, epoch_times = train_epoch(epoch_index=epoch,
                          tb_writer=tb_writer,
                          training_loader=train_loader,
                          loss_fn=loss_fn,
                          model=tinymodel,
                          optimizer=optimizer,
                          scheduler=scheduler,
                          device=device)
  epochs_times.append(epoch_times)

  tinymodel.eval()
  with torch.no_grad():
    val_loss = val_epoch(val_loader=val_loader,
```

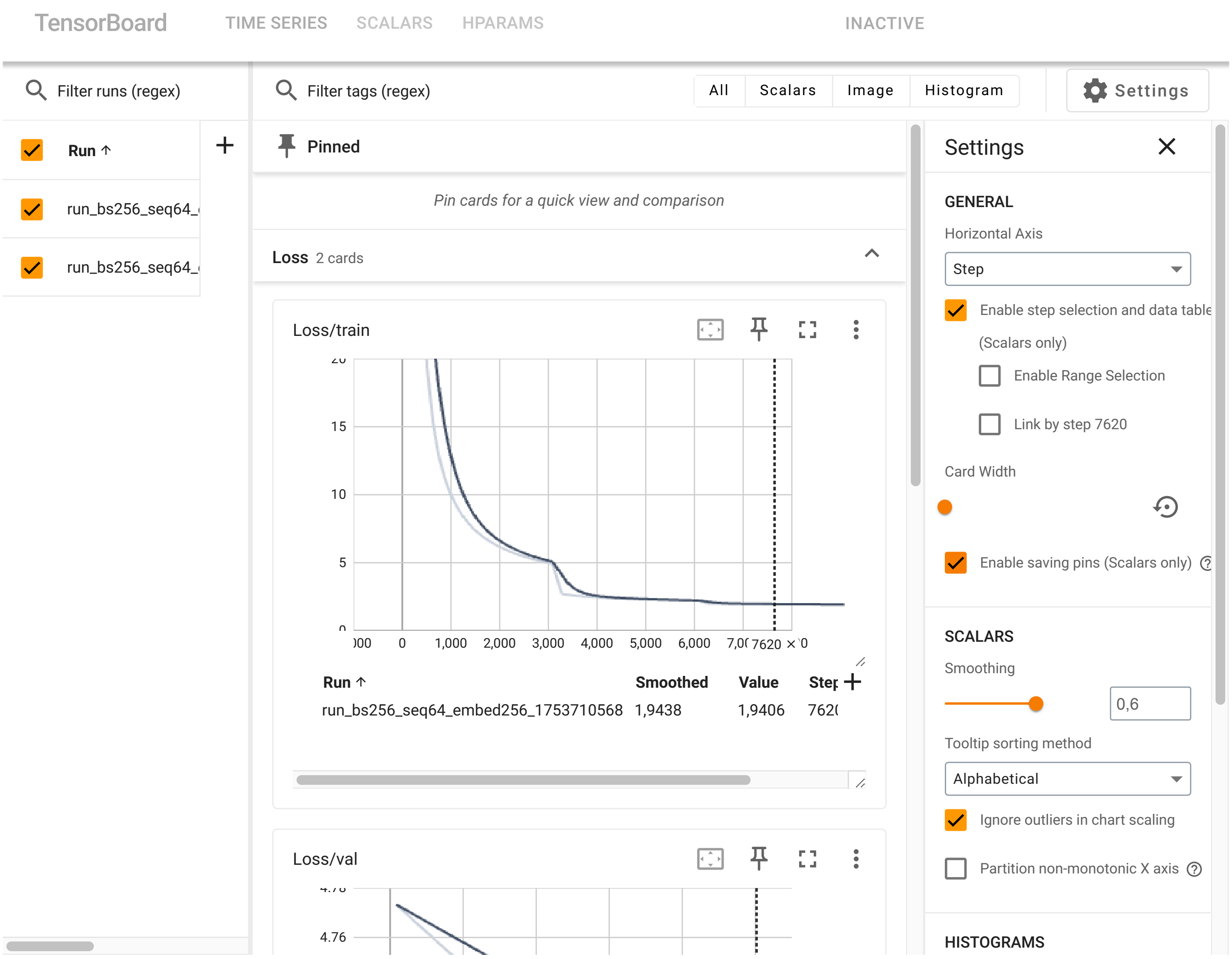
```
                    tb_writer=tb_writer,
                    model=tinymodel,
                    loss_fn=loss_fn,
                    epoch=epoch,
                    device=device)
    print(f"Epoch {epoch + 1}: Train Loss {train_loss}, Val Loss {val_loss}")

  # Save model
  if val_loss < best_val_loss:
    best_val_loss = val_loss
    checkpoint_path = os.path.join(save_dir, f"best_model_val_{val_loss:.4f}.pt")
    torch.save({
        "epoch": epoch,
        "model_state_dict": tinymodel.state_dict(),
        "optimizer_state_dict": optimizer.state_dict(),
        "train_loss": train_loss,
        "val_loss": val_loss
    }, checkpoint_path)
    print(f"Best model saved at epoch {epoch} with Val Loss {val_loss:.4f}")
```

```
  batch 100 loss (avg so far): 2.0073
  batch 200 loss (avg so far): 2.0027
  batch 300 loss (avg so far): 1.9990
  batch 400 loss (avg so far): 1.9957
  batch 500 loss (avg so far): 1.9920
  batch 600 loss (avg so far): 1.9889
  batch 700 loss (avg so far): 1.9856
  batch 800 loss (avg so far): 1.9823
  batch 900 loss (avg so far): 1.9796
  batch 1000 loss (avg so far): 1.9769
  batch 1100 loss (avg so far): 1.9740
  batch 1200 loss (avg so far): 1.9715
  batch 1300 loss (avg so far): 1.9689
  batch 1400 loss (avg so far): 1.9663
  batch 1500 loss (avg so far): 1.9636
  batch 1600 loss (avg so far): 1.9610
  batch 1700 loss (avg so far): 1.9585
  batch 1800 loss (avg so far): 1.9561
  batch 1900 loss (avg so far): 1.9538
  batch 2000 loss (avg so far): 1.9515
  batch 2100 loss (avg so far): 1.9491
  batch 2200 loss (avg so far): 1.9469
  batch 2300 loss (avg so far): 1.9448
  batch 2400 loss (avg so far): 1.9426
  batch 2500 loss (avg so far): 1.9406

Epoch 3 finished | Loss=1.9395 | Time=76.25s
Epoch 3: Train Loss 1.9394881067797543, Val Loss 4.58490207195282
Best model saved at epoch 2 with Val Loss 4.5849
  batch 100 loss (avg so far): 1.8851
  batch 200 loss (avg so far): 1.8853
  batch 300 loss (avg so far): 1.8843
  batch 400 loss (avg so far): 1.8839
  batch 500 loss (avg so far): 1.8828
  batch 600 loss (avg so far): 1.8813
  batch 700 loss (avg so far): 1.8801
  batch 800 loss (avg so far): 1.8789
  batch 900 loss (avg so far): 1.8778
  batch 1000 loss (avg so far): 1.8768
  batch 1100 loss (avg so far): 1.8758
  batch 1200 loss (avg so far): 1.8751
  batch 1300 loss (avg so far): 1.8739
  batch 1400 loss (avg so far): 1.8731
  batch 1500 loss (avg so far): 1.8722
  batch 1600 loss (avg so far): 1.8714
  batch 1700 loss (avg so far): 1.8707
  batch 1800 loss (avg so far): 1.8700
  batch 1900 loss (avg so far): 1.8695
  batch 2000 loss (avg so far): 1.8690
  batch 2100 loss (avg so far): 1.8684
  batch 2200 loss (avg so far): 1.8678
  batch 2300 loss (avg so far): 1.8672
  batch 2400 loss (avg so far): 1.8668
  batch 2500 loss (avg so far): 1.8663

Epoch 4 finished | Loss=1.8659 | Time=76.66s
Epoch 4: Train Loss 1.8659167950507254, Val Loss 4.584011560678482
Best model saved at epoch 3 with Val Loss 4.5840
```

```
tb_writer.add_hparams(
  {
    "batch_size": batch_size,
    "seq_length": seq_length,
    "lr": 0.001,
    "embed_dim": embed_dim,
    "num_layers": num_att_layers,
    "vocab_size": len(vocab)
```

```
    },
    {
      "train_loss": train_loss,
      "val_loss": val_loss
    }
)
```

```
%tensorboard --logdir runs
```




```
tinymodel.eval()
with torch.no_grad():
    out = tinymodel(test_dataset.tokens[-64:].unsqueeze(0).to(device))
index2word[torch.argmax(out[:, -1, :]).item()]
```

```
'g'
```

```
pred_length = 50
pred = []
generated_tokens = test_dataset.tokens[-64:].tolist()

for _ in range(pred_length):
    inp = torch.tensor(generated_tokens[-64:], device=device).unsqueeze(0)
    with torch.no_grad():
        out = tinymodel(inp)
    next_token_id = torch.argmax(out[:, -1, :], dim=-1).item()
    generated_tokens.append(next_token_id)
    pred.append(index2word[next_token_id])


pretty_text = ''.join(pred)
print(pretty_text)
```

```
gloucester:
the god the prince the god the prince the god the pri
```

```
activities = [ProfilerActivity.CPU]
if torch.cuda.is_available():
    device = "cuda"
    activities += [ProfilerActivity.CUDA]
elif torch.xpu.is_available():
    device = "xpu"
    activities += [ProfilerActivity.XPU]
else:
    print(
        "Neither CUDA nor XPU devices are available to demonstrate profiling on acceleration devices"
    )
    import sys

    sys.exit(0)

model = tinymodel.to(device)
inputs = torch.tensor(test_dataset.tokens[-64:].tolist(), device=device).unsqueeze(0)

with profile(activities=activities) as prof:
    model(inputs)

prof.export_chrome_trace("trace.json")
# chrome://tracing


sort_by_keyword = "self_" + device + "_time_total"
print(prof.key_averages().table(sort_by=sort_by_keyword, row_limit=10))
```

```
-------------------------------------------------------  ------------  ------------  ------------  ------------  -------
                                                   Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU tim
-------------------------------------------------------  ------------  ------------  ------------  ------------  -------
                                            aten::addmm         7.44%     383.041us        12.22%     629.446us      89.
                         volta_sgemm_32x32_sliced1x4_tn         0.00%       0.000us         0.00%       0.000us       0.
                                  volta_sgemm_128x32_tn         0.00%       0.000us         0.00%       0.000us       0.
                         volta_sgemm_64x32_sliced1x4_tn         0.00%       0.000us         0.00%       0.000us       0.
                                              aten::bmm         1.90%      97.845us         2.41%     124.234us      62.
void cublasLt::splitKreduce_kernel<32, 16, int, floa...         0.00%       0.000us         0.00%       0.000us       0.
                                aten::native_layer_norm         1.29%      66.683us         2.73%     140.829us      70.
void at::native::(anonymous namespace)::vectorized_l...         0.00%       0.000us         0.00%       0.000us       0.
                         volta_sgemm_32x32_sliced1x4_nn         0.00%       0.000us         0.00%       0.000us       0.
                                         aten::_softmax         1.01%      52.152us         1.49%      76.813us      38.
-------------------------------------------------------  ------------  ------------  ------------  ------------  -------
Self CPU time total: 5.151ms
Self CUDA time total: 280.274us
```

```
import statistics
summary_lines = [
    line for line in prof.key_averages().table(sort_by=sort_by_keyword, row_limit=10).splitlines() if line.startswith("Self
]
summary_lines.append(statistics.mean(epochs_times))


summary_lines
```

```
['Self CPU time total: 5.151ms',
 'Self CUDA time total: 280.274us',
 76.39216899871826]
```

Empieza a programar o a crear código con IA.