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
```

```
--2025-07-31 11:55:23--  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.1'

input.txt.1         100%[===================>]   1.06M  --.-KB/s    in 0.04s

2025-07-31 11:55:24 (26.9 MB/s) - 'input.txt.1' saved [1115394/1115394]
```

```
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()
```

```
%load_ext tensorboard
```

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)

    for param in model.parameters():
      param.grad = None
    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)
```

```
def train_model(model,
                train_loader,
                val_loader,
                loss_fn,
                optimizer,
                scheduler,
                device,
                num_epochs,
                save_dir,
                tb_writer=None):

    best_val_loss = float("inf")
    epochs_times = []

    # Initial evaluation before training
    model.eval()
    with torch.no_grad():
        val_loss = val_epoch(
            val_loader=val_loader,
            tb_writer=tb_writer,
            model=model,
            loss_fn=loss_fn,
            epoch=0,
            device=device
        )
    print(f"Epoch 0: Val Loss {val_loss:.4f}")

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

        model.eval()
        with torch.no_grad():
            val_loss = val_epoch(
                val_loader=val_loader,
                tb_writer=tb_writer,
                model=model,
                loss_fn=loss_fn,
                epoch=epoch,
                device=device
            )

        print(f"Epoch {epoch + 1}: Train Loss {train_loss:.4f}, Val Loss {val_loss:.4f}")

        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": model.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 + 1} with Val Loss {val_loss:.4f}")

    return epochs_times, best_val_loss
```

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
```

```
[' ', 'th', 'ou', ' ', 'd', 'i', 'd', 'st', ' ', 'k', 'il', 'l', ' ', 'ou', 'r', ' ', 'te', 'nd', 'er', ' ', 'b', 'ro',
 ['th', 'ou', ' ', 'd', 'i', 'd', 'st', ' ', 'k', 'il', 'l', ' ', 'ou', 'r', ' ', 'te', 'nd', 'er', ' ', 'b', 'ro', 'th',
```

```
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

```
compile = True
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
base_lr = 0.001
```

```
warmup_steps = 100
save_dir = "checkpoints"
os.makedirs(save_dir, exist_ok=True)
os.makedirs("log_profiler", exist_ok=True)
```

Regular model

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


tinymodel = DecoderOnlySmall(len(vocab), embed_dim, num_heads, seq_length, num_att_layers)
model = tinymodel


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

```
1615500
```

```
total_steps = len(train_loader) * num_epochs
model.to(device)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=base_lr)
scheduler = get_lr_scheduler(optimizer, warmup_steps, total_steps)


epochs_times_nor, best_val_loss = train_model(model,
                                        train_loader,
                                        val_loader,
                                        loss_fn,
                                        optimizer,
                                        scheduler,
                                        device,
                                        num_epochs,
                                        save_dir,
                                        tb_writer)
```

```
        batch 1700 loss (avg so far): 1.8702
        batch 1800 loss (avg so far): 1.8695
        batch 1900 loss (avg so far): 1.8688
        batch 2000 loss (avg so far): 1.8682
        batch 2100 loss (avg so far): 1.8676
        batch 2200 loss (avg so far): 1.8671
        batch 2300 loss (avg so far): 1.8666
        batch 2400 loss (avg so far): 1.8662
        batch 2500 loss (avg so far): 1.8657

    Epoch 4 finished | Loss=1.8654 | Time=79.23s
    Epoch 4: Train Loss 1.8654, Val Loss 4.5822
```

```
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)
  },
  {
    "best_val_loss": best_val_loss
  }
)
```

Compiled default

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


tinymodel = DecoderOnlySmall(len(vocab), embed_dim, num_heads, seq_length, num_att_layers)
tinymodel_compiled = torch.compile(tinymodel)
model_comp = tinymodel_compiled


total_steps = len(train_loader) * num_epochs
model_comp.to(device)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model_comp.parameters(), lr=base_lr)
scheduler = get_lr_scheduler(optimizer, warmup_steps, total_steps)


epochs_times_comp, best_val_loss = train_model(model_comp,
                                        train_loader,
                                        val_loader,
                                        loss_fn,
                                        optimizer,
                                        scheduler,
                                        device,
                                        num_epochs,
                                        save_dir,
                                        tb_writer)
```

```
        batch 100 loss (avg so far): 1.8799
        batch 200 loss (avg so far): 1.8774
        batch 300 loss (avg so far): 1.8756
        batch 400 loss (avg so far): 1.8751
        batch 500 loss (avg so far): 1.8735
        batch 600 loss (avg so far): 1.8733
        batch 700 loss (avg so far): 1.8723
        batch 800 loss (avg so far): 1.8711
        batch 900 loss (avg so far): 1.8701
        batch 1000 loss (avg so far): 1.8690
        batch 1100 loss (avg so far): 1.8679
        batch 1200 loss (avg so far): 1.8668
        batch 1300 loss (avg so far): 1.8658
        batch 1400 loss (avg so far): 1.8648
        batch 1500 loss (avg so far): 1.8639
        batch 1600 loss (avg so far): 1.8633
        batch 1700 loss (avg so far): 1.8626
        batch 1800 loss (avg so far): 1.8618
        batch 1900 loss (avg so far): 1.8613
        batch 2000 loss (avg so far): 1.8606
        batch 2100 loss (avg so far): 1.8600
        batch 2200 loss (avg so far): 1.8594
        batch 2300 loss (avg so far): 1.8590
        batch 2400 loss (avg so far): 1.8586
        batch 2500 loss (avg so far): 1.8581

    Epoch 4 finished | Loss=1.8579 | Time=74.88s
    Epoch 4: Train Loss 1.8579, Val Loss 4.5815
    Best model saved at epoch 4 with Val Loss 4.5815
```

```
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)
  },
  {
    "best_val_loss": best_val_loss
  }
)
```

Compiled reduce-overhead

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


tinymodel = DecoderOnlySmall(len(vocab), embed_dim, num_heads, seq_length, num_att_layers)
tinymodel_compiled = torch.compile(tinymodel, mode="reduce-overhead")
model_over = tinymodel_compiled


total_steps = len(train_loader) * num_epochs
model_over.to(device)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model_over.parameters(), lr=base_lr)
scheduler = get_lr_scheduler(optimizer, warmup_steps, total_steps)


epochs_times_over, best_val_loss = train_model(model_over,
                                        train_loader,
                                        val_loader,
                                        loss_fn,
                                        optimizer,
                                        scheduler,
                                        device,
                                        num_epochs,
                                        save_dir,
                                        tb_writer)
```

```
        batch 1300 loss (avg so far): 1.9741
        batch 1400 loss (avg so far): 1.9716
        batch 1500 loss (avg so far): 1.9692
        batch 1600 loss (avg so far): 1.9668
        batch 1700 loss (avg so far): 1.9645
        batch 1800 loss (avg so far): 1.9622
        batch 1900 loss (avg so far): 1.9599
        batch 2000 loss (avg so far): 1.9577
        batch 2100 loss (avg so far): 1.9555
        batch 2200 loss (avg so far): 1.9534
        batch 2300 loss (avg so far): 1.9512
        batch 2400 loss (avg so far): 1.9492
        batch 2500 loss (avg so far): 1.9472

      Epoch 3 finished | Loss=1.9460 | Time=74.81s
      Epoch 3: Train Loss 1.9460, Val Loss 4.5899
      Best model saved at epoch 3 with Val Loss 4.5899
        batch 100 loss (avg so far): 1.8983
        batch 200 loss (avg so far): 1.8951
        batch 300 loss (avg so far): 1.8931
        batch 400 loss (avg so far): 1.8911
        batch 500 loss (avg so far): 1.8902
        batch 600 loss (avg so far): 1.8889
        batch 700 loss (avg so far): 1.8875
        batch 800 loss (avg so far): 1.8862
        batch 900 loss (avg so far): 1.8850
        batch 1000 loss (avg so far): 1.8837
        batch 1100 loss (avg so far): 1.8827
        batch 1200 loss (avg so far): 1.8816
        batch 1300 loss (avg so far): 1.8805
        batch 1400 loss (avg so far): 1.8794
        batch 1500 loss (avg so far): 1.8787
        batch 1600 loss (avg so far): 1.8780
        batch 1700 loss (avg so far): 1.8773
        batch 1800 loss (avg so far): 1.8764
        batch 1900 loss (avg so far): 1.8758
        batch 2000 loss (avg so far): 1.8753
        batch 2100 loss (avg so far): 1.8747
        batch 2200 loss (avg so far): 1.8742
        batch 2300 loss (avg so far): 1.8738
        batch 2400 loss (avg so far): 1.8733
        batch 2500 loss (avg so far): 1.8729

      Epoch 4 finished | Loss=1.8727 | Time=74.88s
      Epoch 4: Train Loss 1.8727, Val Loss 4.5861
      Best model saved at epoch 4 with Val Loss 4.5861
```

```
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)
  },
  {
    "best_val_loss": best_val_loss
  }
)
```

Compiled max-autotune

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


tinymodel = DecoderOnlySmall(len(vocab), embed_dim, num_heads, seq_length, num_att_layers)
tinymodel_compiled = torch.compile(tinymodel, mode="max-autotune")
model_auto = tinymodel_compiled


total_steps = len(train_loader) * num_epochs
model_auto.to(device)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model_auto.parameters(), lr=base_lr)
scheduler = get_lr_scheduler(optimizer, warmup_steps, total_steps)


epochs_times_auto, best_val_loss = train_model(model_auto,
                                               train_loader,
                                               val_loader,
                                               loss_fn,
                                               optimizer,
                                               scheduler,
                                               device,
```

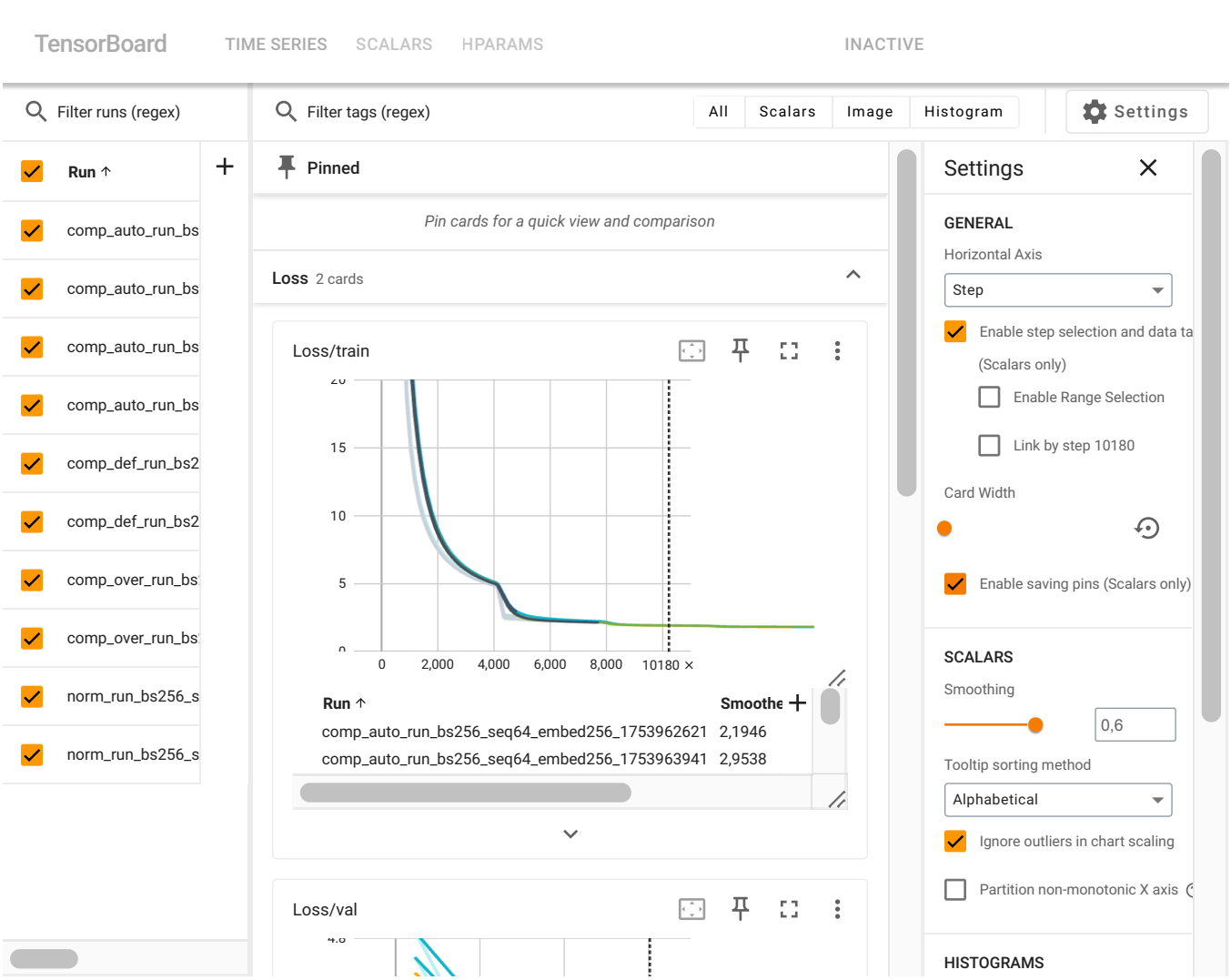
```
                                        num_epochs,
                                        save_dir,
                                        tb_writer)
```

```
    batch 200 loss (avg so far): 2.0186
    batch 300 loss (avg so far): 2.0141
    batch 400 loss (avg so far): 2.0097
    batch 500 loss (avg so far): 2.0060
    batch 600 loss (avg so far): 2.0027
    batch 700 loss (avg so far): 1.9993
    batch 800 loss (avg so far): 1.9957
    batch 900 loss (avg so far): 1.9928
    batch 1000 loss (avg so far): 1.9895
    batch 1100 loss (avg so far): 1.9866
    batch 1200 loss (avg so far): 1.9840
    batch 1300 loss (avg so far): 1.9810
    batch 1400 loss (avg so far): 1.9780
    batch 1500 loss (avg so far): 1.9752
    batch 1600 loss (avg so far): 1.9725
    batch 1700 loss (avg so far): 1.9699
    batch 1800 loss (avg so far): 1.9674
    batch 1900 loss (avg so far): 1.9650
    batch 2000 loss (avg so far): 1.9627
    batch 2100 loss (avg so far): 1.9604
    batch 2200 loss (avg so far): 1.9582
    batch 2300 loss (avg so far): 1.9558
    batch 2400 loss (avg so far): 1.9536
    batch 2500 loss (avg so far): 1.9515

  Epoch 3 finished | Loss=1.9502 | Time=75.81s
  Epoch 3: Train Loss 1.9502, Val Loss 4.5960
  Best model saved at epoch 3 with Val Loss 4.5960
    batch 100 loss (avg so far): 1.8980
    batch 200 loss (avg so far): 1.8953
    batch 300 loss (avg so far): 1.8937
    batch 400 loss (avg so far): 1.8921
    batch 500 loss (avg so far): 1.8907
    batch 600 loss (avg so far): 1.8895
    batch 700 loss (avg so far): 1.8883
    batch 800 loss (avg so far): 1.8869
    batch 900 loss (avg so far): 1.8858
    batch 1000 loss (avg so far): 1.8850
    batch 1100 loss (avg so far): 1.8839
    batch 1200 loss (avg so far): 1.8827
    batch 1300 loss (avg so far): 1.8818
    batch 1400 loss (avg so far): 1.8809
    batch 1500 loss (avg so far): 1.8800
    batch 1600 loss (avg so far): 1.8794
    batch 1700 loss (avg so far): 1.8787
    batch 1800 loss (avg so far): 1.8780
    batch 1900 loss (avg so far): 1.8773
    batch 2000 loss (avg so far): 1.8768
    batch 2100 loss (avg so far): 1.8761
    batch 2200 loss (avg so far): 1.8755
    batch 2300 loss (avg so far): 1.8750
    batch 2400 loss (avg so far): 1.8746
    batch 2500 loss (avg so far): 1.8741

  Epoch 4 finished | Loss=1.8738 | Time=76.16s
  Epoch 4: Train Loss 1.8738, Val Loss 4.5851
  Best model saved at epoch 4 with Val Loss 4.5851
```

```
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)
  },
  {
    "best_val_loss": best_val_loss
  }
)
```

Compare

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




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

```
'k'
```

```
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 = model(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)
```

```
king richard ii:
i have the prince edward iv:
the prince edward iv:
```

Performance

```
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.eval()
    model_comp.eval()
    model_over.eval()
    model_auto.eval()
    inputs = torch.tensor(test_dataset.tokens[-64:].tolist(), device=device).unsqueeze(0)

    with profile(activities=activities) as prof:
      model(inputs)
    with profile(activities=activities) as prof_comp:
      model_comp(inputs)
    with profile(activities=activities) as prof_over:
      model_over(inputs)
    with profile(activities=activities) as prof_auto:
      model_auto(inputs)

    prof.export_chrome_trace("trace.json")
    prof_comp.export_chrome_trace("trace_1.json")
    prof_over.export_chrome_trace("trace_2.json")
    prof_auto.export_chrome_trace("trace_3.json")
    # chrome://tracing
```

```
W0731 12:28:17.089000 2722 torch/_dynamo/convert_frame.py:906] [0/8] torch._dynamo hit config.cache_size_limit (8)
W0731 12:28:17.089000 2722 torch/_dynamo/convert_frame.py:906] [0/8]    function: 'forward' (/tmp/ipython-input-12499907
W0731 12:28:17.089000 2722 torch/_dynamo/convert_frame.py:906] [0/8]    last reason: 0/0: GLOBAL_STATE changed: grad_mod
W0731 12:28:17.089000 2722 torch/_dynamo/convert_frame.py:906] [0/8] To log all recompilation reasons, use TORCH_LOGS="r
W0731 12:28:17.089000 2722 torch/_dynamo/convert_frame.py:906] [0/8] To diagnose recompilation issues, see https://pytor
```

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

```
                          volta_sgemm_32x32_sliced1x4_nn         0.00%       0.000us         0.00%       0.000us       0
                                       aten::_softmax         1.08%      87.851us         1.58%     128.227us      64
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------
Self CPU time total: 8.117ms
Self CUDA time total: 280.087us

-------------------------------------------------------  ------------  ------------  ------------  ------------  ------
                                                   Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU ti
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------
CachingAutotuner.benchmark_all_configs (dynamo_timed...         0.00%       0.000us         0.00%       0.000us       0
                                           aten::fill_         0.06%       1.950ms         0.13%       3.852ms      13
void at::native::vectorized_elementwise_kernel<4, at...         0.00%       0.000us         0.00%       0.000us       0
           _recursive_joint_graph_passes (dynamo_timed)         0.00%       0.000us         0.00%       0.000us       0
                                     CompiledFunction         0.05%       1.550ms        11.13%     335.002ms     335
                                           aten::copy_         0.08%       2.305ms         0.19%       5.845ms      17
                         Memcpy DtoD (Device -> Device)         0.00%       0.000us         0.00%       0.000us       0
        triton_per_fused__softmax_div_eq_masked_fill_1         0.00%       0.000us         0.00%       0.000us       0
                     triton_poi_fused_add_embedding_0         0.00%       0.000us         0.00%       0.000us       0
           triton_poi_fused_relu_threshold_backward_3         0.00%       0.000us         0.00%       0.000us       0
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------
Self CPU time total: 3.010s
Self CUDA time total: 316.491ms

-------------------------------------------------------  ------------  ------------  ------------  ------------  ------
                                                   Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU ti
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------
                                           aten::addmm         6.46%     300.588us         9.67%     450.430us      64
                                volta_sgemm_128x32_tn         0.00%       0.000us         0.00%       0.000us       0
                          volta_sgemm_32x32_sliced1x4_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         1.71%      79.454us         2.19%     101.951us      50
void cublasLt::splitKreduce_kernel<32, 16, int, floa...         0.00%       0.000us         0.00%       0.000us       0
```

```
                                                aten::addmm         5.97%     237.117us         9.30%     369.674us      52
                                     volta_sgemm_128x32_tn         0.00%       0.000us         0.00%       0.000us       0
                                volta_sgemm_32x32_sliced1x4_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         1.44%      57.199us         1.89%      75.144us      37
    void cublasLt::splitKreduce_kernel<32, 16, int, floa...         0.00%       0.000us         0.00%       0.000us       0
                                   aten::native_layer_norm         1.08%      42.917us         2.21%      87.967us      43
    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::add         1.39%      55.305us         2.10%      83.288us      27
    ------------------------------------------------------  ------------  ------------  ------------  ------------  ------
    Self CPU time total: 3.973ms
    Self CUDA time total: 133.471us
```

```
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
```

```
['Self CPU time total: 8.117ms', 'Self CUDA time total: 280.087us']
```

```
epochs_times_nor
```

```
[80.3916072845459, 79.37176394462585, 79.14192152023315, 79.2252516746521]
```

```
epochs_times_comp
```

```
[81.97524499893188, 75.58613777160645, 75.60358476638794, 74.88457536697388]
```

```
epochs_times_over
```

```
[80.73609757423401, 75.96565246582031, 74.8133430480957, 74.87538361549377]
```

```
epochs_times_auto
```

```
[79.39380621910095, 76.50081133842468, 75.81329274177551, 76.16427898406982]
```

Learning

From the profiler we can see that what actually is being done is mainly to optimize cpu time. That is the real bottleneck of the model. So now that we now it the next step to do evidently is to load batch_size. See if now some of the operations are executed in GPU and how much time can we save from training.

```
LOWER_CASE = True
seq_length = 64
old_batch_size = 256
batch_size = 512
embed_dim = 256
num_epochs = 4
num_heads = 1
num_att_layers = 1
base_lr = 0.001
linear_base_lr = base_lr*batch_size/old_batch_size
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
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, prefetch_factor
val_loader= DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=2, pin_memory=True, prefetch_factor=2)
test_loader= DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=2, pin_memory=True, prefetch_factor=


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


tinymodel = DecoderOnlySmall(len(vocab), embed_dim, num_heads, seq_length, num_att_layers)
tinymodel_compiled = torch.compile(tinymodel)
model_comp_b = tinymodel_compiled


total_steps = len(train_loader) * num_epochs
model_comp_b.to(device)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model_comp_b.parameters(), lr=base_lr)
warmup_steps = int(total_steps*0.03)
scheduler = get_lr_scheduler(optimizer, warmup_steps, total_steps)


total_steps, warmup_steps, num_epochs, batch_size, linear_base_lr
```

```
(5120, 153, 4, 512, 0.002)
```

```
epochs_times_comp_b, best_val_loss = train_model(model_comp_b,
                                        train_loader,
                                        val_loader,
                                        loss_fn,
                                        optimizer,
                                        scheduler,
                                        device,
                                        num_epochs,
                                        save_dir,
                                        tb_writer)
```

```
  batch 700 loss (avg so far): 11.0931
  batch 800 loss (avg so far): 10.0912
  batch 900 loss (avg so far): 9.3064
  batch 1000 loss (avg so far): 8.6742
  batch 1100 loss (avg so far): 8.1542
  batch 1200 loss (avg so far): 7.7184

Epoch 1 finished | Loss=7.4175 | Time=74.56s
Epoch 1: Train Loss 7.4175, Val Loss 4.7917
Best model saved at epoch 1 with Val Loss 4.7917
  batch 100 loss (avg so far): 2.8855
  batch 200 loss (avg so far): 2.8791
  batch 300 loss (avg so far): 2.8734
  batch 400 loss (avg so far): 2.8667
  batch 500 loss (avg so far): 2.8612
  batch 600 loss (avg so far): 2.8524
  batch 700 loss (avg so far): 2.8368
  batch 800 loss (avg so far): 2.8157
  batch 900 loss (avg so far): 2.7938
  batch 1000 loss (avg so far): 2.7721
  batch 1100 loss (avg so far): 2.7510
  batch 1200 loss (avg so far): 2.7306

Epoch 2 finished | Loss=2.7145 | Time=74.05s
Epoch 2: Train Loss 2.7145, Val Loss 4.7117
```

```
  batch 700 loss (avg so far): 2.3741
  batch 800 loss (avg so far): 2.3649
  batch 900 loss (avg so far): 2.3564
  batch 1000 loss (avg so far): 2.3486
  batch 1100 loss (avg so far): 2.3412
  batch 1200 loss (avg so far): 2.3344

Epoch 3 finished | Loss=2.3293 | Time=74.45s
Epoch 3: Train Loss 2.3293, Val Loss 4.6619
Best model saved at epoch 3 with Val Loss 4.6619
  batch 100 loss (avg so far): 2.2482
  batch 200 loss (avg so far): 2.2439
  batch 300 loss (avg so far): 2.2415
  batch 400 loss (avg so far): 2.2382
  batch 500 loss (avg so far): 2.2356
  batch 600 loss (avg so far): 2.2334
  batch 700 loss (avg so far): 2.2315
  batch 800 loss (avg so far): 2.2295
  batch 900 loss (avg so far): 2.2282
  batch 1000 loss (avg so far): 2.2271
  batch 1100 loss (avg so far): 2.2259
  batch 1200 loss (avg so far): 2.2249

Epoch 4 finished | Loss=2.2242 | Time=74.51s
Epoch 4: Train Loss 2.2242, Val Loss 4.6548
Best model saved at epoch 4 with Val Loss 4.6548
```

```
print(torch.cuda.memory_summary())
```

```
|===========================================================================|
|                  PyTorch CUDA memory summary, device ID 0                 |
|---------------------------------------------------------------------------|
|            CUDA OOMs: 0            |        cudaMalloc retries: 0         |
|===========================================================================|
|        Metric         | Cur Usage  | Peak Usage | Tot Alloc  | Tot Freed  |
|---------------------------------------------------------------------------|
| Allocated memory      |  86918 KiB | 753268 KiB |  18781 GiB |  18781 GiB |
|       from large pool |  25600 KiB | 693248 KiB |  18357 GiB |  18357 GiB |
|       from small pool |  61318 KiB |  65056 KiB |    424 GiB |    424 GiB |
|---------------------------------------------------------------------------|
| Active memory         |  86918 KiB | 753268 KiB |  18781 GiB |  18781 GiB |
|       from large pool |  25600 KiB | 693248 KiB |  18357 GiB |  18357 GiB |
|       from small pool |  61318 KiB |  65056 KiB |    424 GiB |    424 GiB |
|---------------------------------------------------------------------------|
| Requested memory      |  86910 KiB | 752235 KiB |  18767 GiB |  18767 GiB |
|       from large pool |  25600 KiB | 692224 KiB |  18343 GiB |  18343 GiB |
|       from small pool |  61310 KiB |  65047 KiB |    424 GiB |    423 GiB |
|---------------------------------------------------------------------------|
| GPU reserved memory   |   1234 MiB |   1234 MiB |   2376 MiB |   1142 MiB |
|       from large pool |   1160 MiB |   1160 MiB |   2272 MiB |   1112 MiB |
|       from small pool |     74 MiB |     74 MiB |    104 MiB |     30 MiB |
|---------------------------------------------------------------------------|
| Non-releasable memory |  19578 KiB | 347066 KiB |  10387 GiB |  10387 GiB |
|       from large pool |  15360 KiB | 343040 KiB |   9916 GiB |   9916 GiB |
|       from small pool |   4218 KiB |   8619 KiB |    471 GiB |    471 GiB |
|---------------------------------------------------------------------------|
| Allocations           |     360    |     382    |    3170 K  |    3170 K  |
|       from large pool |       3    |      20    |     983 K  |     983 K  |
|       from small pool |     357    |     378    |    2187 K  |    2187 K  |
|---------------------------------------------------------------------------|
| Active allocs         |     360    |     382    |    3170 K  |    3170 K  |
|       from large pool |       3    |      20    |     983 K  |     983 K  |
|       from small pool |     357    |     378    |    2187 K  |    2187 K  |
|---------------------------------------------------------------------------|
| GPU reserved segments |      63    |      63    |     110    |      47    |
|       from large pool |      26    |      26    |      58    |      32    |
|       from small pool |      37    |      37    |      52    |      15    |
|---------------------------------------------------------------------------|
```