HW0: Training GPT from Scratch with Parameter-Free Optimization and Various Positional Embedding Methods

Aneesh Muppidi
Harvard SEAS
Harvard College
Cambridge, MA, USA
aneeshmuppidi@college.harvard.edu

Abstract

In this HW0, we implement a simple transformer model trained on the tinyshakespeare dataset, closely following Andrej Karpathy's tutorial , "Let's build GPT: from scratch, in code." **The first notable exploration is demonstrating how a parameter-free optimizer, TRAC, which we recently developed for the reinforcement learning setting, can also help mitigate the sensitivity of the learning rate while training GPT. Secondly, we explore the effects of different positional embedding methods, including learned, sinusoidal, Fourier, and rotary positional embeddings (RoPE), on the model's training performance.** Code for the experiments can be found here.

1 Training with Parameter-Free Optimizers

Modern deep learning optimizers, such as Adam and AdamW (used in this tutorial), rely heavily on the learning rate as a key hyperparameter. However, the learning process can be quite sensitive to the choice of this parameter Cutkosky et al. (2023); Cutkosky (2019), as illustrated in Figure 1.

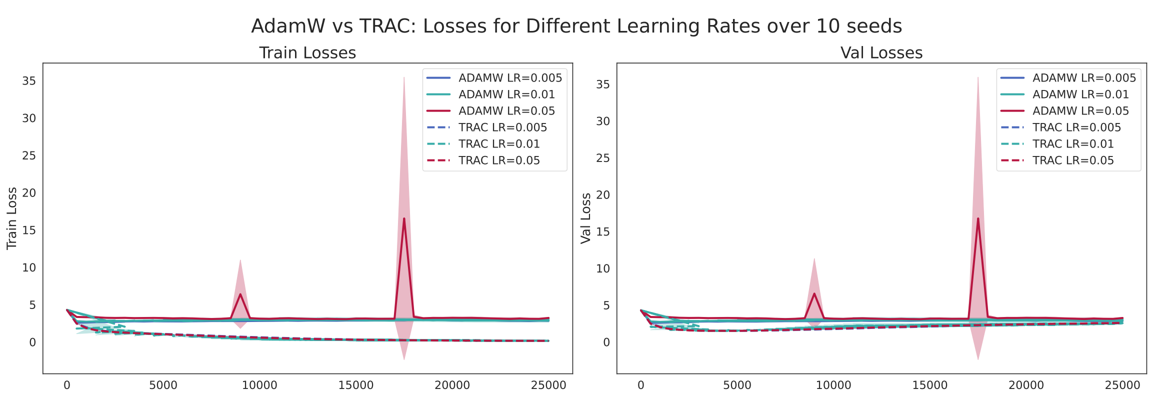

Figure 1: Training and validation losses with AdamW are sensitive to different learning rates, particularly at a learning rate of 0.05. TRAC helps stabilize AdamW, leading to consistent loss convergence regardless of the learning rate used.

To address this sensitivity, learning rate-free optimizers have been developed. Our recent work, TRAC Muppidi et al. (2024), introduces a parameter-free optimizer based on the theory of Online

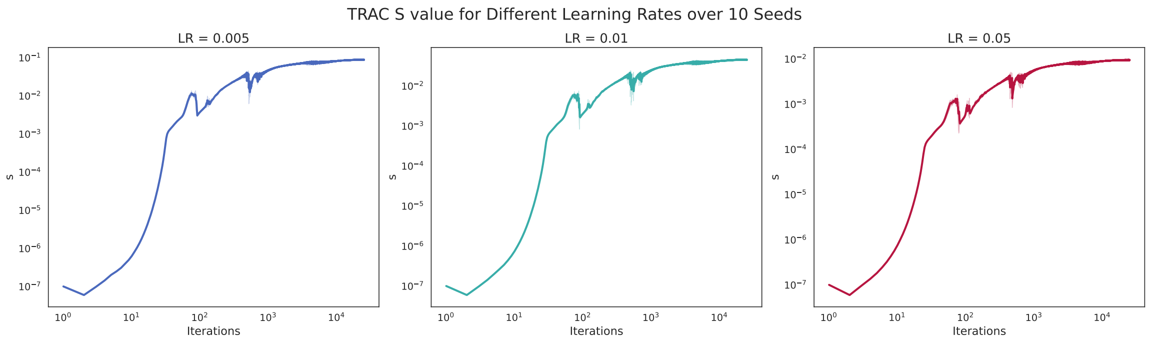

Figure 2: Average scaling values chosen by TRAC during training for three different AdamW learning rates.

Convex Optimization Orabona (2023), with a unique connection to adaptive regularization. While TRAC has demonstrated significant improvements in adversarial reinforcement learning settings Dohare et al. (2021), its effectiveness in supervised deep learning tasks, such as GPT training, has not been explored.

At a high level, TRAC operates by wrapping around a base optimizer (like AdamW) and applying a scaling factor s to adjust the updates from the base optimizer in an online data-dependent manner. Both theoretically and empirically, TRAC is shown to be insensitive to the base optimizer's learning rate. More information can be found here.

In our first experiment, we train a GPT model from scratch using the same architectural hyperparameters as outlined in the YouTube tutorial. We conduct training runs with 10 different random seeds across three learning rates using AdamW as the baseline optimizer. We then also run the same 10 seeds across the three learning rates but with TRAC as the meta optimizer.

As shown in Figure 1, **the training and validation losses are sensitive to the learning rate when using AdamW alone**. However, when using **TRAC, we see more consistent performance across these learning rates, leading to better loss convergences.**

When plotting the average scaling value for TRAC across the three learning rates (Figure 2), we also see that the scaling value converges to quite a small value.

2 Implementing and Training with Different Positional Embedding Methods

We were particularly interested in exploring the effects of different positional embedding methods on model performance. The original YouTube tutorial uses learned positional embeddings. In our experiments, we implemented additional methods, including sinusoidal positional embeddings (as described in Section 3.5 of the "Attention is All You Need" paper Vaswani et al. (2023)), Rotary Position Embeddings (RoPE) Su et al. (2023), and Fourier Feature embeddings (similar to Li et al. (2021).

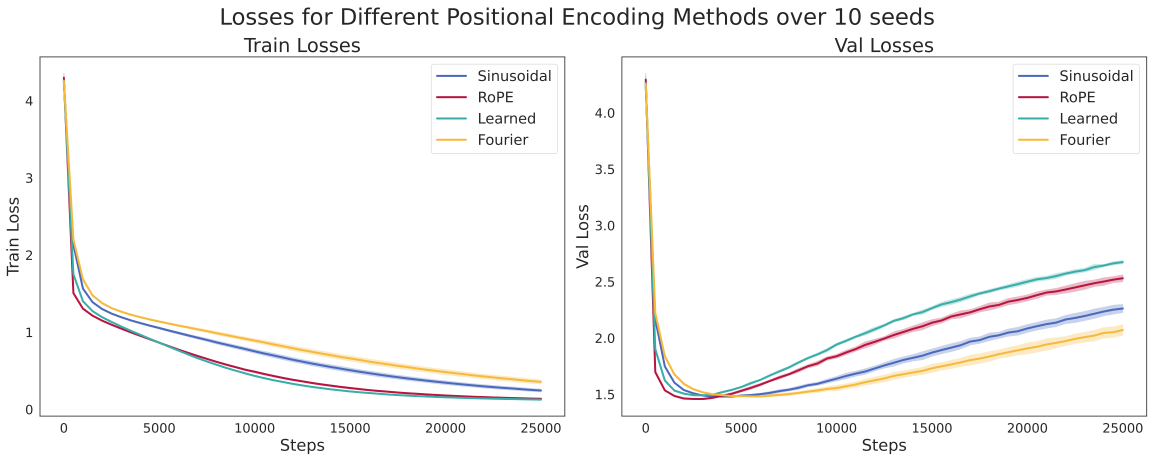

Figure 3: Training and Validation losses of Sinusoidal, RoPE, Learned, and Fourier Feature positional embeddings

Although our architecture is not scaled to a level where we can draw statistically significant conclusions, Figure 3 shows that after running experiments with 10 different seeds for each method, RoPE and learned positional embeddings performed the best.

References

Ashok Cutkosky. Combining online learning guarantees. In *Conference on Learning Theory*, pp. 895–913. PMLR, 2019.

Ashok Cutkosky, Aaron Defazio, and Harsh Mehta. Mechanic: A learning rate tuner. *Advances in Neural Information Processing Systems*, 36, 2023.

Shibhansh Dohare, Richard S Sutton, and A Rupam Mahmood. Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv preprint arXiv:2108.06325*, 2021.

Yang Li, Si Si, Gang Li, Cho-Jui Hsieh, and Samy Bengio. Learnable fourier features for multi-dimensional spatial positional encoding, 2021. URL https://arxiv.org/abs/2106.02795.

Aneesh Muppidi, Zhiyu Zhang, and Heng Yang. Fast trac: A parameter-free optimizer for lifelong reinforcement learning, 2024. URL https://arxiv.org/abs/2405.16642.

Francesco Orabona. A modern introduction to online learning. *arXiv preprint arXiv:1912.13213*, 2023.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

3 Code

The github repo for this code can be found here, but is also here in the pdf below:

Aneesh Muppidi
aneeshmuppidi@college.harvard.edu

Harvard John A. Paulson
School of Engineering
and Applied Sciences

```python
import argparse
import math
import torch
import torch.nn as nn
import os
from torch.nn import functional as F
from pathlib import Path
from trac_optimizer import start_trac

BATCH_SIZE = 64
BLOCK_SIZE = 256
MAX_ITERS = 25000
EVAL_INTERVAL = 500
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
EVAL_ITERS = 200
N_EMBD = 384
N_HEAD = 6
N_LAYER = 6
DROPOUT = 0.2

class SinusoidalPositionEmbedding(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, length):
        position = torch.arange(length, dtype=torch.float32).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, self.dim, 2).float() * (-math.log(10000.0) / self.dim))
        embeddings = torch.zeros(length, self.dim)
        embeddings[:, 0::2] = torch.sin(position * div_term)
        embeddings[:, 1::2] = torch.cos(position * div_term)
        return embeddings.to(DEVICE)

class FourierFeatureEmbeddings(nn.Module):
    def __init__(self, n_embd, max_seq_len):
        super().__init__()
        self.n_embd = n_embd
        self.max_seq_len = max_seq_len
        self.num_ff = n_embd // 2
        self.freq_bands = 2.0 ** torch.linspace(0., math.log2(self.num_ff) - 1, self.num_ff)

    def forward(self, positions):
        seq_len = positions.shape[0]
        pos_expanded = positions.unsqueeze(1)
        freq_expanded = self.freq_bands.unsqueeze(0).to(positions.device)
        angles = pos_expanded * freq_expanded
        fourier_features = torch.cat([torch.sin(angles), torch.cos(angles)], dim=-1)
        return fourier_features

class Head(nn.Module):
    def __init__(self, head_size, pos_encoding):
        super().__init__()
        self.key = nn.Linear(N_EMBD, head_size, bias=False)
        self.query = nn.Linear(N_EMBD, head_size, bias=False)
        self.value = nn.Linear(N_EMBD, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(BLOCK_SIZE, BLOCK_SIZE)))
        self.dropout = nn.Dropout(DROPOUT)
        self.head_size = head_size
        self.pos_encoding = pos_encoding

    def forward(self, x):
        B, T, C = x.shape
        k = self.key(x)
        q = self.query(x)
        v = self.value(x)

```

Aneesh Muppidi
aneeshmuppidi@college.harvard.edu

Harvard John A. Paulson School of Engineering and Applied Sciences

```python
        if self.pos_encoding == 'rope':
            sinusoidal_pos = self.get_sinusoidal_embeddings(T, self.head_size)
            q, k = self.apply_rotary_position_embeddings(sinusoidal_pos, q, k)

        wei = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)
        out = wei @ v
        return out

    def apply_rotary_position_embeddings(self, sinusoidal_pos, q, k):
        sin, cos = sinusoidal_pos.chunk(2, dim=-1)
        q_rot = torch.stack((-q[..., 1::2], q[..., ::2]), dim=-1)
        k_rot = torch.stack((-k[..., 1::2], k[..., ::2]), dim=-1)
        q_rot = torch.reshape(q_rot, q.shape[:-1] + (q.shape[-1]//2, 2)) * torch.stack((cos, sin), dim=-1)
        k_rot = torch.reshape(k_rot, k.shape[:-1] + (k.shape[-1]//2, 2)) * torch.stack((cos, sin), dim=-1)
        q_rot = torch.reshape(q_rot, q.shape)
        k_rot = torch.reshape(k_rot, k.shape)
        return q_rot, k_rot

    def get_sinusoidal_embeddings(self, n_positions, dim):
        position = torch.arange(n_positions, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, dim, 2).float() * (-math.log(10000.0) / dim))
        sinusoidal_emb = torch.zeros((n_positions, dim))
        sinusoidal_emb[:, 0::2] = torch.sin(position * div_term)
        sinusoidal_emb[:, 1::2] = torch.cos(position * div_term)
        return sinusoidal_emb.to(self.key.weight.device)

class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads, head_size, pos_encoding):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size, pos_encoding) for _ in range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, N_EMBD)
        self.dropout = nn.Dropout(DROPOUT)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out

class FeedForward(nn.Module):
    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(DROPOUT),
        )

    def forward(self, x):
        return self.net(x)

class Block(nn.Module):
    def __init__(self, n_embd, n_head, pos_encoding):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size, pos_encoding)
        self.ffwd = FeedForward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
```

Aneesh Muppidi
aneeshmuppidi@college.harvard.edu

Harvard John A. Paulson
School of Engineering
and Applied Sciences

```
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x

class GPTLanguageModel(nn.Module):
    def __init__(self, vocab_size, pos_encoding):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, N_EMBD)
        self.pos_encoding = pos_encoding

        if pos_encoding == 'learned':
            self.position_embedding_table = nn.Embedding(BLOCK_SIZE, N_EMBD)
        elif pos_encoding == 'sinusoidal':
            self.position_embedding = SinusoidalPositionEmbedding(N_EMBD)
        elif pos_encoding == 'fourier':
            self.position_embedding = FourierFeatureEmbeddings(N_EMBD, BLOCK_SIZE)

        self.blocks = nn.Sequential(*[Block(N_EMBD, N_HEAD, pos_encoding) for _ in range(N_LAYER)])
        self.ln_f = nn.LayerNorm(N_EMBD)
        self.lm_head = nn.Linear(N_EMBD, vocab_size)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        tok_emb = self.token_embedding_table(idx)

        if self.pos_encoding == 'learned':
            pos_emb = self.position_embedding_table(torch.arange(T, device=DEVICE))
            x = tok_emb + pos_emb
        elif self.pos_encoding == 'sinusoidal':
            pos_emb = self.position_embedding(T)
            x = tok_emb + pos_emb
        elif self.pos_encoding == 'fourier':
            pos_emb = self.position_embedding(torch.arange(T, device=DEVICE))
            x = tok_emb + pos_emb
        else:  # 'rope' or no positional encoding
            x = tok_emb

        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -BLOCK_SIZE:]
            logits, _ = self(idx_cond)
            logits = logits[:, -1, :]
            probs = F.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
```

Aneesh Muppidi
aneeshmuppidi@college.harvard.edu

Harvard John A. Paulson
School of Engineering
and Applied Sciences

```
            idx = torch.cat((idx, idx_next), dim=1)
        return idx

def get_batch(split, train_data, val_data):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - BLOCK_SIZE, (BATCH_SIZE,))
    x = torch.stack([data[i:i+BLOCK_SIZE] for i in ix])
    y = torch.stack([data[i+1:i+BLOCK_SIZE+1] for i in ix])
    return x.to(DEVICE), y.to(DEVICE)

@torch.no_grad()
def estimate_loss(model, train_data, val_data):
    model.eval()

    def evaluate_split(split):
        losses = torch.zeros(EVAL_ITERS, device=DEVICE)
        for k in range(EVAL_ITERS):
            X, Y = get_batch(split, train_data, val_data)
            _, loss = model(X, Y)
            losses[k] = loss.item()
        return losses.mean().item()

    results = {split: evaluate_split(split) for split in ['train', 'val']}

    model.train()
    return results

def train(model, train_data, val_data, log_file, learning_rate, optimizer, trac_log_file):
    if optimizer == "adamw":
        optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
    elif optimizer == "trac":
        optimizer = start_trac(log_file=trac_log_file, Base=torch.optim.AdamW)(model.parameters(), lr=learning_rate)
    for iter in range(MAX_ITERS):
        if iter % EVAL_INTERVAL == 0 or iter == MAX_ITERS - 1:
            losses = estimate_loss(model, train_data, val_data)
            with open(log_file, 'a') as f:
                f.write(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}\n")
        xb, yb = get_batch('train', train_data, val_data)
        _, loss = model(xb, yb)
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()

def main(args):
    torch.manual_seed(args.seed)

    dir = f"/n/home04/amuppidi/nanoGPT/ng-video-lecture/logs/{args.optimizer}/{args.lr}"
    Path(dir).mkdir(parents=True, exist_ok=True)
    log_file = f"{dir}/{args.seed}_log.txt"
    trac_log_file = f"{dir}/{args.seed}_trac_log.txt"
    # clear everything from log
    with open(log_file, 'w') as f:
        f.write('')

    with open(args.input_file, 'r', encoding='utf-8') as f:
        text = f.read()

    chars = sorted(list(set(text)))
    vocab_size = len(chars)
    stoi = {ch: i for i, ch in enumerate(chars)}
    itos = {i: ch for i, ch in enumerate(chars)}
    encode = lambda s: [stoi[c] for c in s]
    decode = lambda l: ''.join([itos[i] for i in l])

```

Aneesh Muppidi
aneeshmuppidi@college.harvard.edu


```python
    data = torch.tensor(encode(text), dtype=torch.long)
    n = int(0.9 * len(data))
    train_data = data[:n]
    val_data = data[n:]

    model = GPTLanguageModel(vocab_size, args.pos_encoding).to(DEVICE)

    if args.train:
        train(model, train_data, val_data, log_file, args.lr, args.optimizer, trac_log_file)

    context = torch.zeros((1, 1), dtype=torch.long, device=DEVICE)
    generated_text = decode(model.generate(context, max_new_tokens=500)[0].tolist())
    print("Generated text:")
    print(generated_text)
    if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="GPT Language Model with various positional encodings")
    parser.add_argument("--input_file", type=str, default="/n/home04/amuppidi/nanoGPT/ng-video-lecture/input.txt", help="Path to the input text file")
    parser.add_argument("--seed", type=int, default=1337, help="Random seed for reproducibility")
    parser.add_argument("--lr", type=float, default=3e-4, help="Learning rate")
    parser.add_argument("--optimizer", type=str, default="adamw", choices=["adamw", "trac"], help="Optimizer to use")
    parser.add_argument("--pos_encoding", type=str, default="learned", choices=["learned", "sinusoidal", "fourier", "rope"], help="Type of positional encoding to use")
    parser.add_argument("--train", action="store_true", help="Train the model")
    args = parser.parse_args()
    main(args)
```

Listing 1: Code for HW0 and this report.