motor_gpt_dev

January 19, 2026

[1]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import random
import os, glob, math
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```python
device = "cuda" if torch.cuda.is_available() else "cpu"
print("device:", device)
```

```
device: cuda
```

0.0.1 Attention

[3]:
```python
class SelfAttentionHead(nn.Module):
    def __init__(self, head_dim, embed_dim, traj_size, dropout=0.0):
        super().__init__()
        # linear projections for Q, V, K
        self.key = nn.Linear(embed_dim, head_dim, bias=False)
        self.query = nn.Linear(embed_dim, head_dim, bias=False)
        self.value = nn.Linear(embed_dim, head_dim, bias=False)
        mask = torch.tril(torch.ones(traj_size, traj_size)).view(1, traj_size,␣
↪traj_size)
        self.atten_drop = nn.Dropout(dropout)
        self.resid_drop = nn.Dropout(dropout)
        self.register_buffer("mask", mask)

    def forward(self, x):
        B, T, C = x.shape

        q = self.query(x) #(B, T, H)
        k = self.key(x)
        v = self.value(x)

        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))  # (B,␣
↪T, T)
        att = att.masked_fill(self.mask[:,:T,:T]==0, float('-inf'))
```

```
        att = F.softmax(att, dim=-1)  # (B, T, T)
        att = self.atten_drop(att)

        out = att @ v  # (B, T, H)
        out = self.resid_drop(out)

        return out
```

[4]:
```
# test the SelfAttentionHead
head = SelfAttentionHead(head_dim=16, embed_dim=32, traj_size=8)
x = torch.randn(4, 8, 32)  # (B, T, C)
out = head(x)
print(out.shape)
```

```
torch.Size([4, 8, 16])
```

[5]:
```
class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads, embed_dim, head_dim, traj_size, dropout=0.0):
        super().__init__()
        self.heads = nn.ModuleList([SelfAttentionHead(head_dim, embed_dim,␣
↪traj_size, dropout) for _ in range(num_heads)])
        self.proj = nn.Linear(num_heads * head_dim, embed_dim, bias=False)
        self.drop = nn.Dropout(dropout)

    def forward(self, x):
        multi_head_out = [h(x) for h in self.heads]  # list of (B, T, head_size)
        multi_head_concat = torch.cat(multi_head_out, dim=-1) # (B, T,␣
↪num_heads * head_size)

        out = self.drop(self.proj(multi_head_concat))  # (B, T, embed_dim)

        return out
```

[6]:
```
mha = MultiHeadAttention(num_heads=4, embed_dim=32, head_dim=8, traj_size=8,␣
↪dropout=0.0)
x = torch.randn(4, 8, 32)
out = mha(x)
print(out.shape)
```

```
torch.Size([4, 8, 32])
```

0.0.2 Transformer Block

[7]:
```
class FeedForward(nn.Module):
    def __init__(self, embed_dim, expansion=4, dropout=0.0):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(embed_dim, expansion*embed_dim),
            nn.GELU(),
            nn.Linear(expansion*embed_dim, embed_dim),
            nn.Dropout(dropout),
        )
    def forward(self, x): return self.net(x)

class Block(nn.Module):
    def __init__(self, embed_dim, n_head, block_size, mlp_expansion=4,␣
 ↪dropout=0.0):
        super().__init__()
        assert embed_dim % n_head == 0
        head_size = embed_dim // n_head
        self.ln1 = nn.LayerNorm(embed_dim)
        self.attn = MultiHeadAttention(n_head, embed_dim, head_size,␣
 ↪block_size, dropout)
        self.ln2 = nn.LayerNorm(embed_dim)
        self.mlp = FeedForward(embed_dim, expansion=mlp_expansion,␣
 ↪dropout=dropout)

    def forward(self, x):
        # TODO
        x = x + self.attn(self.ln1(x)) # skip connection
        x = x + self.mlp(self.ln2(x)) # skip connection

        return x
```

0.0.3 Motor GPT

[8]:
```
class MotorGPT(nn.Module):
    def __init__(self, action_size=6, embed_dim=192, traj_size=128, n_layer=4,␣
 ↪n_head=4, dropout=0.0):
        super().__init__()
        self.action_size = action_size
        self.traj_size = traj_size

        # "token embedding" for continuous actions
        self.token_emb = nn.Linear(action_size, embed_dim, bias=False)
        self.pos_emb   = nn.Embedding(traj_size, embed_dim)

        self.blocks = nn.ModuleList([
```

```
            Block(embed_dim, n_head, traj_size, dropout=dropout)
            for _ in range(n_layer)
        ])
        self.ln_f = nn.LayerNorm(embed_dim)

        # regression head back to action space
        self.head = nn.Linear(embed_dim, action_size, bias=True)

        self.apply(self._init_weights)

    def _init_weights(self, m):
        # NanoGPT init is fine conceptually, but add LayerNorm explicitly
        if isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, mean=0.0, std=0.02)
            if m.bias is not None:
                nn.init.zeros_(m.bias)
        elif isinstance(m, nn.Embedding):
            nn.init.normal_(m.weight, mean=0.0, std=0.02)
        elif isinstance(m, nn.LayerNorm):
            nn.init.ones_(m.weight)
            nn.init.zeros_(m.bias)

    def forward(self, x, targets=None):
        """
        x:       (B, T, A)  normalized actions
        targets: (B, T, A)  next-step normalized actions
        """
        B, T, A = x.shape
        assert T <= self.traj_size, "Sequence length exceeds traj_size"


        #token and position embedding
        tok = self.token_emb(x)  # (B, T, E)
        pos = self.pos_emb(torch.arange(T, device=x.device)).unsqueeze(0)  #␣
↪(1, T, E)
        h = tok + pos

        #pass through Transformer blocks
        for block in self.blocks:
            h = block(h)

        # regression head
        h = self.ln_f(h)
        pred = self.head(h)      # (B, T, A)

        loss = None
        if targets is not None:
```

```
            loss = F.mse_loss(pred, targets)  # regression v1

        return pred, loss

    @torch.no_grad()
    def generate(self, seed_actions, max_new_steps=100, noise_std=0.0):
        """
        seed_actions: (B, T0, A) normalized actions
        returns:      (B, T0+max_new_steps, A) normalized actions
        """
        self.eval()
        out = seed_actions
        for _ in range(max_new_steps):
            cond = out[:, -self.traj_size:, :]      # crop context, so only␣
↪last traj_size steps
            pred, _ = self(cond)                    # (B, Tcond, A), this␣
↪includes the whole context
            next_a = pred[:, -1, :]                 # (B, A), take only the␣
↪last time step

            # optional stochasticity (since MSE tends to be "average")
            if noise_std > 0:
                next_a = next_a + noise_std * torch.randn_like(next_a)

            out = torch.cat([out, next_a.unsqueeze(1)], dim=1)
        return out
```

[ ]:
```
def estimate_loss(model, train_eps, val_eps, train_wlens, val_wlens,
                  trajectory_len, batch_size, eval_iters, device):
    # samples random batches from train and val sets, computes mean loss
    model.eval()
    out = {}
    with torch.no_grad():
        for name, eps, wl in [("train", train_eps, train_wlens), ("val",␣
↪val_eps, val_wlens)]:
            losses = []
            for _ in range(eval_iters):
                xb, yb = get_batch_train(eps, wl, trajectory_len, batch_size,␣
↪device)
                _, loss = model(xb, yb)
                losses.append(loss.item())
            out[name] = float(np.mean(losses))
    model.train()
    return out # returns {"train": mean_loss, "val": mean_loss}


def save_checkpoint(path, model, optimizer, step, best_val=None, metadata=None):
```

```python
    # builds a directory and saves model and optimizer state dicts
    state = {
        "model": model.state_dict(),
        "optimizer": optimizer.state_dict(),
        "step": step,
        "best_val": best_val,
        "metadata": metadata or {},
    }
    torch.save(state, path)


def train_model(model, train_eps, val_eps, train_wlens, val_wlens,
                trajectory_len=128, batch_size=64, max_iters=2000,
                eval_interval=100, lr=3e-4, weight_decay=0.1,
                grad_clip=1.0, device="cuda", checkpoint_dir=None,
                save_interval=None, save_best=True, metadata=None):

    model.to(device)
    opt = torch.optim.AdamW(model.parameters(), lr=lr,␣
 ↪weight_decay=weight_decay, betas=(0.9, 0.95))


    if checkpoint_dir is not None:
        os.makedirs(checkpoint_dir, exist_ok=True)
        if save_interval is None:
            save_interval = eval_interval

    best_val = float("inf")
    training_losses, validation_losses, eval_steps = [], [], []

    for it in range(max_iters):
        # sample batch, compute loss
        xb, yb = get_batch_train(train_eps, train_wlens, trajectory_len,␣
 ↪batch_size, device)
        _, loss = model(xb, yb)

        # backprop and update
        opt.zero_grad(set_to_none=True)
        loss.backward()
        if grad_clip is not None:
            torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)
        opt.step()

        #periodic evaluation
        if it % eval_interval == 0 or it == max_iters - 1:
            est = estimate_loss(model, train_eps, val_eps, train_wlens,␣
 ↪val_wlens,
```

```
                                trajectory_len, batch_size, eval_iters=25,␣
↪device=device)
            print(f"iter {it:5d} | train {est['train']:.6f} | val {est['val']:.
↪6f}")
            training_losses.append(est["train"])
            validation_losses.append(est["val"])
            eval_steps.append(it)

            if checkpoint_dir is not None:
                if save_best and est["val"] < best_val:
                    best_val = est["val"]
                    best_path = os.path.join(checkpoint_dir, "best.pt")
                    save_checkpoint(best_path, model, opt, it,␣
↪best_val=best_val, metadata=metadata)

                if it % save_interval == 0 or it == max_iters - 1:
                    ckpt_path = os.path.join(checkpoint_dir, f"ckpt_{it:06d}.
↪pt")
                    save_checkpoint(ckpt_path, model, opt, it,␣
↪best_val=best_val, metadata=metadata)

    plt.figure()
    plt.plot(eval_steps, training_losses, label="train")
    plt.plot(eval_steps, validation_losses, label="val")
    plt.xlabel("Iteration")
    plt.ylabel("MSE loss")
    plt.title("MotorGPT Train/Val Loss")
    plt.legend()
    plt.grid(True)
    plt.show()
```