```
import numpy as np
import math
import pdb

import torch
import torch.nn as nn
from torch.nn import functional as F

from .ein import EinLinear

class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads
        self.key = nn.Linear(config.n_embd, config.n_embd)
        self.query = nn.Linear(config.n_embd, config.n_embd)
        self.value = nn.Linear(config.n_embd, config.n_embd)
        # regularization
        self.attn_drop = nn.Dropout(config.attn_pdrop)
        self.resid_drop = nn.Dropout(config.resid_pdrop)
        # output projection
        self.proj = nn.Linear(config.n_embd, config.n_embd)
        # causal mask to ensure that attention is only applied to the left in the input
sequence
        self.register_buffer("mask", torch.tril(torch.ones(config.block_size,
config.block_size))
                                     .view(1, 1, config.block_size, config.block_size))
        ## mask previous value estimates
        joined_dim = config.observation_dim + config.action_dim + 2
        self.mask.squeeze()[:,joined_dim-1::joined_dim] = 0
        ##
        self.n_head = config.n_head

    def forward(self, x, layer_past=None):
        B, T, C = x.size()

        # calculate query, key, values for all heads in batch and move head forward to be
the batch dim
        ## [ B x n_heads x T x head_dim ]
        k = self.key(x).view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B,
nh, T, hs)
        q = self.query(x).view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B,
nh, T, hs)
        v = self.value(x).view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B,
nh, T, hs)

        # causal self-attention; Self-attend: (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh,
T, T)
        ## [ B x n_heads x T x T ]
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(self.mask[:,:,:T,:T] == 0, float('-inf'))
        att = F.softmax(att, dim=-1)
        self._attn_map = att.clone()
        att = self.attn_drop(att)
        ## [ B x n_heads x T x head_size ]
        y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        ## [ B x T x embedding_dim ]
        y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs
side by side

        # output projection
        y = self.resid_drop(self.proj(y))
        return y
```

Each discretised dimension is mapped to n_embd sized vector

lower triangular form

```
class Block(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.ln1 = nn.LayerNorm(config.n_embd)
        self.ln2 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.mlp = nn.Sequential(
            nn.Linear(config.n_embd, 4 * config.n_embd),
            nn.GELU(),
            nn.Linear(4 * config.n_embd, config.n_embd),
            nn.Dropout(config.resid_pdrop),
        )

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x

class GPT(nn.Module):
    """  the full GPT language model, with a context size of block_size """

    def __init__(self, config):
        super().__init__()

        # input embedding stem (+1 for stop token)
        self.tok_emb = nn.Embedding(config.vocab_size * config.transition_dim + 1,
config.n_embd)

        self.pos_emb = nn.Parameter(torch.zeros(1, config.block_size, config.n_embd))
        self.drop = nn.Dropout(config.embd_pdrop)
        # transformer
        self.blocks = nn.Sequential(*[Block(config) for _ in range(config.n_layer)])
        # decoder head
        self.ln_f = nn.LayerNorm(config.n_embd)
        # self.head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
        self.head = EinLinear(config.transition_dim, config.n_embd, config.vocab_size +
1, bias=False)

        self.vocab_size = config.vocab_size
        self.stop_token = config.vocab_size * config.transition_dim
        self.block_size = config.block_size
        self.observation_dim = config.observation_dim

        self.action_dim = config.action_dim
        self.transition_dim = config.transition_dim
        self.action_weight = config.action_weight
        self.reward_weight = config.reward_weight
        self.value_weight = config.value_weight

        self.embedding_dim = config.n_embd
        self.apply(self._init_weights)

    def get_block_size(self):
        return self.block_size

    def _init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding)):
            module.weight.data.normal_(mean=0.0, std=0.02)
            if isinstance(module, nn.Linear) and module.bias is not None:
                module.bias.data.zero_()
        elif isinstance(module, nn.LayerNorm):
            module.bias.data.zero_()
            module.weight.data.fill_(1.0)

    def configure_optimizers(self, train_config):
```

```
        """
        This long function is unfortunately doing something very simple and is being very
defensive:
        We are separating out all parameters of the model into two buckets: those that
will experience
        weight decay for regularization and those that won't (biases, and layernorm/-
embedding weights).
        We are then returning the PyTorch optimizer object.
        """

        # separate out all parameters to those that will and won't experience
regularizing weight decay
        decay = set()
        no_decay = set()
        whitelist_weight_modules = (torch.nn.Linear, EinLinear)
        blacklist_weight_modules = (torch.nn.LayerNorm, torch.nn.Embedding)
        for mn, m in self.named_modules():
            for pn, p in m.named_parameters():
                fpn = '%s.%s' % (mn, pn) if mn else pn # full param name

                if pn.endswith('bias'):
                    # all biases will not be decayed
                    no_decay.add(fpn)
                elif pn.endswith('weight') and isinstance(m, whitelist_weight_modules):
                    # weights of whitelist modules will be weight decayed
                    decay.add(fpn)
                elif pn.endswith('weight') and isinstance(m, blacklist_weight_modules):
                    # weights of blacklist modules will NOT be weight decayed
                    no_decay.add(fpn)

        # special case the position embedding parameter in the root GPT module as not
decayed
        no_decay.add('pos_emb')

        # validate that we considered every parameter
        param_dict = {pn: p for pn, p in self.named_parameters()}
        inter_params = decay & no_decay
        union_params = decay | no_decay
        assert len(inter_params) == 0, "parameters %s made it into both decay/no_decay
sets!" % (str(inter_params), )
        assert len(param_dict.keys() - union_params) == 0, "parameters %s were not
separated into either decay/no_decay set!" \
                                                    % (str(param_dict.keys() -
union_params), )

        # create the pytorch optimizer object
        optim_groups = [
            {"params": [param_dict[pn] for pn in sorted(list(decay))], "weight_decay":
train_config.weight_decay},
            {"params": [param_dict[pn] for pn in sorted(list(no_decay))], "weight_decay":
0.0},
        ]
        optimizer = torch.optim.AdamW(optim_groups, lr=train_config.learning_rate,
betas=train_config.betas)
        return optimizer

    def offset_tokens(self, idx):
        _, t = idx.shape
        n_states = int(np.ceil(t / self.transition_dim))
        offsets = torch.arange(self.transition_dim) * self.vocab_size
        offsets = offsets.repeat(n_states).to(idx.device)
        offset_idx = idx + offsets[:t]
        offset_idx[idx == self.vocab_size] = self.stop_token
        return offset_idx

    def pad_to_full_observation(self, x, verify=False):
```

```
        b, t, _ = x.shape
        n_pad = (self.transition_dim - t % self.transition_dim) % self.transition_dim
        padding = torch.zeros(b, n_pad, self.embedding_dim, device=x.device)
        ## [ B x T' x embedding_dim ]
        x_pad = torch.cat([x, padding], dim=1)
        ## [ (B * T' / transition_dim) x transition_dim x embedding_dim ]
        x_pad = x_pad.view(-1, self.transition_dim, self.embedding_dim)
        if verify:
            self.verify(x, x_pad)
        return x_pad, n_pad

    def verify(self, x, x_pad):
        b, t, embedding_dim = x.shape
        n_states = int(np.ceil(t / self.transition_dim))
        inds = torch.arange(0, self.transition_dim).repeat(n_states)[:t]
        for i in range(self.transition_dim):
            x_ = x[:,inds == i]
            t_ = x_.shape[1]
            x_pad_ = x_pad[:,i].view(b, n_states, embedding_dim)[:,:t_]
            print(i, x_.shape, x_pad_.shape)
            try:
                assert (x_ == x_pad_).all()
            except:
                pdb.set_trace()

    def forward(self, idx, targets=None, mask=None):
        """
            idx : [ B x T ]
            values : [ B x 1 x 1 ]
        """
        b, t = idx.size()
        assert t <= self.block_size, "Cannot forward, model block size is exhausted."

        offset_idx = self.offset_tokens(idx)
        ## [ B x T x embedding_dim ]
        # forward the GPT model
        token_embeddings = self.tok_emb(offset_idx) # each index maps to a (learnable)
vector
        ## [ 1 x T x embedding_dim ]
        position_embeddings = self.pos_emb[:, :t, :] # each position maps to a
(learnable) vector
        ## [ B x T x embedding_dim ]
        x = self.drop(token_embeddings + position_embeddings)
        x = self.blocks(x)
        ## [ B x T x embedding_dim ]
        x = self.ln_f(x)

        ## [ (B * T' / transition_dim) x transition_dim x embedding_dim ]
        x_pad, n_pad = self.pad_to_full_observation(x)
        ## [ (B * T' / transition_dim) x transition_dim x (vocab_size + 1) ]
        logits = self.head(x_pad)
        ## [ B x T' x (vocab_size + 1) ]
        logits = logits.reshape(b, t + n_pad, self.vocab_size + 1)
        ## [ B x T x (vocab_size + 1) ]
        logits = logits[:,:t]

        # if we are given some desired targets also calculate the loss
        if targets is not None:
            loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), targets.view(-1),
reduction='none')
            if self.action_weight != 1 or self.reward_weight != 1 or self.value_weight !=
1:
                #### make weights
                n_states = int(np.ceil(t / self.transition_dim))
                weights = torch.cat([
                    torch.ones(self.observation_dim, device=idx.device),
```

```
                    torch.ones(self.action_dim, device=idx.device) * self.action_weight,
                    torch.ones(1, device=idx.device) * self.reward_weight,
                    torch.ones(1, device=idx.device) * self.value_weight,
                ])
                ## [ t + 1]
                weights = weights.repeat(n_states)
                ## [ b x t ]
                weights = weights[1:].repeat(b, 1)
                ####
                loss = loss * weights.view(-1)
            loss = (loss * mask.view(-1)).mean()
        else:
            loss = None

        return logits, loss

class ConditionalGPT(GPT):

    def __init__(self, config):
        ## increase block size by `observation_dim` because we are prepending a goal
observation
        ## to the sequence
        config.block_size += config.observation_dim
        super().__init__(config)
        self.goal_emb = nn.Embedding(config.vocab_size * config.observation_dim,
config.n_embd)

    def get_block_size(self):
        return self.block_size - self.observation_dim

    def forward(self, idx, goal, targets=None, mask=None):
        b, t = idx.size()
        assert t <= self.block_size, "Cannot forward, model block size is exhausted."

        #### goal
        offset_goal = self.offset_tokens(goal)
        goal_embeddings = self.goal_emb(offset_goal)
        #### /goal

        offset_idx = self.offset_tokens(idx)
        ## [ B x T x embedding_dim ]
        # forward the GPT model
        token_embeddings = self.tok_emb(offset_idx) # each index maps to a (learnable)
vector
        ## [ 1 x T x embedding_dim ]
        position_embeddings = self.pos_emb[:, :t, :] # each position maps to a
(learnable) vector
        ## [ B x T x embedding_dim ]
        x = self.drop(token_embeddings + position_embeddings)

        #### goal
        ## [ B + (obs_dim + T) x embedding_dim ]
        gx = torch.cat([goal_embeddings, x], dim=1)
        gx = self.blocks(gx)
        x = gx[:, self.observation_dim:]
        #### /goal

        ## [ B x T x embedding_dim ]
        x = self.ln_f(x)

        ## [ (B * T' / transition_dim) x transition_dim x embedding_dim ]
        x_pad, n_pad = self.pad_to_full_observation(x)
        ## [ (B * T' / transition_dim) x transition_dim x (vocab_size + 1) ]
        logits = self.head(x_pad)
        ## [ B x T' x (vocab_size + 1) ]
        logits = logits.reshape(b, t + n_pad, self.vocab_size + 1)
```

```
        ## [ B x T x (vocab_size + 1) ]
        logits = logits[:,:t]

        # if we are given some desired targets also calculate the loss
        if targets is not None:
            loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), targets.view(-1),
reduction='none')
            if self.action_weight != 1 or self.reward_weight != 1 or self.value_weight !=
1:
                #### make weights
                n_states = int(np.ceil(t / self.transition_dim))
                weights = torch.cat([
                    torch.ones(self.observation_dim, device=idx.device),
                    torch.ones(self.action_dim, device=idx.device) * self.action_weight,
                    torch.ones(1, device=idx.device) * self.reward_weight,
                    torch.ones(1, device=idx.device) * self.value_weight,
                ])
                ## [ t + 1]
                weights = weights.repeat(n_states)
                ## [ b x t ]
                weights = weights[1:].repeat(b, 1)
                ####
                loss = loss * weights.view(-1)
            loss = (loss * mask.view(-1)).mean()
        else:
            loss = None

        return logits, loss
```