```python
import numpy as np
import torch
import torch.nn as nn

import transformers

from decision_transformer.models.model import TrajectoryModel
from decision_transformer.models.trajectory_gpt2 import GPT2Model


class DecisionTransformer(TrajectoryModel):

    """
    This model uses GPT to model (Return_1, state_1, action_1, Return_2, state_2, ...)
    """

    def __init__(
            self,
            state_dim,
            act_dim,
            hidden_size,
            max_length=None,
            max_ep_len=4096,
            action_tanh=True,
            **kwargs
    ):
        super().__init__(state_dim, act_dim, max_length=max_length)

        self.hidden_size = hidden_size
        config = transformers.GPT2Config(
            vocab_size=1,  # doesn't matter -- we don't use the vocab
            n_embd=hidden_size,
            **kwargs
        )

        # note: the only difference between this GPT2Model and the default Huggingface
version
        # is that the positional embeddings are removed (since we'll add those ourselves)
        self.transformer = GPT2Model(config)

        self.embed_timestep = nn.Embedding(max_ep_len, hidden_size)
        self.embed_return = torch.nn.Linear(1, hidden_size)
        self.embed_state = torch.nn.Linear(self.state_dim, hidden_size)
        self.embed_action = torch.nn.Linear(self.act_dim, hidden_size)

        self.embed_ln = nn.LayerNorm(hidden_size)

        # note: we don't predict states or returns for the paper
        self.predict_state = torch.nn.Linear(hidden_size, self.state_dim)
        self.predict_action = nn.Sequential(
            *([nn.Linear(hidden_size, self.act_dim)] + ([nn.Tanh()] if action_tanh else
[]))
        )
        self.predict_return = torch.nn.Linear(hidden_size, 1)

    def forward(self, states, actions, rewards, returns_to_go, timesteps,
attention_mask=None):

        batch_size, seq_length = states.shape[0], states.shape[1]

        if attention_mask is None:
            # attention mask for GPT: 1 if can be attended to, 0 if not
            attention_mask = torch.ones((batch_size, seq_length), dtype=torch.long)

        # embed each modality with a different head
        state_embeddings = self.embed_state(states)
```

map everything to vector of length hidden_size

normalise since s, a, r could have different magnitude representations

```
        action_embeddings = self.embed_action(actions)
        returns_embeddings = self.embed_return(returns_to_go)  # conditioned on requested returns
        time_embeddings = self.embed_timestep(timesteps)

        # time embeddings are treated similar to positional embeddings
        state_embeddings = state_embeddings + time_embeddings
        action_embeddings = action_embeddings + time_embeddings    # Each time step has 3 tokens
        returns_embeddings = returns_embeddings + time_embeddings

        # this makes the sequence look like (R_1, s_1, a_1, R_2, s_2, a_2, ...)
        # which works nice in an autoregressive sense since states predict actions
        stacked_inputs = torch.stack(
            (returns_embeddings, state_embeddings, action_embeddings), dim=1
        ).permute(0, 2, 1, 3).reshape(batch_size, 3*seq_length, self.hidden_size)
        stacked_inputs = self.embed_ln(stacked_inputs)

        # to make the attention mask fit the stacked inputs, have to stack it as well
        stacked_attention_mask = torch.stack(
            (attention_mask, attention_mask, attention_mask), dim=1
        ).permute(0, 2, 1).reshape(batch_size, 3*seq_length)

        # we feed in the input embeddings (not word indices as in NLP) to the model
        transformer_outputs = self.transformer(
            inputs_embeds=stacked_inputs,                          # main prediction call
            attention_mask=stacked_attention_mask,
        )
        x = transformer_outputs['last_hidden_state']

        # reshape x so that the second dimension corresponds to the original
        # returns (0), states (1), or actions (2); i.e. x[:,1,t] is the token for s_t
        x = x.reshape(batch_size, seq_length, 3, self.hidden_size).permute(0, 2, 1, 3)

        # get predictions
        return_preds = self.predict_return(x[:,2])  # predict next return given state and
action
        state_preds = self.predict_state(x[:,2])    # predict next state given state and
action
        action_preds = self.predict_action(x[:,1])  # predict next action given state

        return state_preds, action_preds, return_preds    # reverse-mapped to original s, a, r dimensions

    def get_action(self, states, actions, rewards, returns_to_go, timesteps, **kwargs):
        # we don't care about the past rewards in this model

        states = states.reshape(1, -1, self.state_dim)
        actions = actions.reshape(1, -1, self.act_dim)
        returns_to_go = returns_to_go.reshape(1, -1, 1)
        timesteps = timesteps.reshape(1, -1)

        if self.max_length is not None:
            states = states[:,-self.max_length:]            # max_length = K in the paper
            actions = actions[:,-self.max_length:]
            returns_to_go = returns_to_go[:,-self.max_length:]
            timesteps = timesteps[:,-self.max_length:]

            # pad all tokens to sequence length
            attention_mask = torch.cat([torch.zeros(self.max_length-states.shape[1]),
torch.ones(states.shape[1])])
            attention_mask = attention_mask.to(dtype=torch.long,
device=states.device).reshape(1, -1)
            states = torch.cat(
                [torch.zeros((states.shape[0], self.max_length-states.shape[1],
self.state_dim), device=states.device), states],
                dim=1).to(dtype=torch.float32)
            actions = torch.cat(
                [torch.zeros((actions.shape[0], self.max_length - actions.shape[1],
```

```
self.act_dim),
                                device=actions.device), actions],
                dim=1).to(dtype=torch.float32)
            returns_to_go = torch.cat(
                [torch.zeros((returns_to_go.shape[0], self.max_length-
returns_to_go.shape[1], 1), device=returns_to_go.device), returns_to_go],
                dim=1).to(dtype=torch.float32)
            timesteps = torch.cat(
                [torch.zeros((timesteps.shape[0], self.max_length-timesteps.shape[1]),
device=timesteps.device), timesteps],
                dim=1
            ).to(dtype=torch.long)
        else:
            attention_mask = None

        _, action_preds, return_preds = self.forward(
            states, actions, None, returns_to_go, timesteps,
attention_mask=attention_mask, **kwargs)

        return action_preds[0,-1]
```