Cartpole Decision Transformer code understanding

```python
import gym
import numpy as np
import pickle
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import time
```

Step 1 - Importing required libraries.
Dataset and dataset loader are important.
Pickle is format in which dataset is loaded

Step 2 - Defining transformer model characteristics

```python
class DecisionTransformer(nn.Module):
    def __init__(self, state_dim=4, act_dim=2, embed_dim=128, num_layers=3, num_heads=4):
        super().__init__()
        self.embed_dim = embed_dim
```

state dimensions = 4 because we have $(x, \dot{x}, \theta, \dot{\theta})$ as the states
action dimensions = 2 because left Right actions
embedding size = 128
number of transformer layers = 3
number of attention heads = 4

will be thoroughly discussed in "Understanding Transformers" section.

Step 3 - embedding layers
What they do? Why are they relevant will be discussed in next section.

```python
self.state_embed = nn.Linear(state_dim, embed_dim)
self.action_embed = nn.Embedding(act_dim, embed_dim)
self.return_embed = nn.Linear(1, embed_dim)
self.time_embed = nn.Embedding(1000, embed_dim)
```

The state vector, action vector, returns to go vector and time step are converted into embedding size of 128.

Based on notebook information - Q = Query = XW_Q, K = key = XW_K and V = Value = XW_V
W_Q, W_K and W_V are weight matrices which are learnt during training. X is the embedding vector. Formed by combining input information to embeddings and combining them.

Step 4 - Transformer encoder
Importance? Need? Relevance? will be discussed later

```python
self.transformer = nn.TransformerEncoder(
    nn.TransformerEncoderLayer(
        d_model=embed_dim,
        nhead=num_heads,
        dim_feedforward=4 * embed_dim,
        batch_first=True
    ),
    num_layers=num_layers
)
```

d_model = embedding size = 128
nhead = num_heads = number of attention heads = 4
dim_feedforward = 4 x 128
It is the feedforward network inside the transformer.
num_layers = number of transformer layers

We actually defined the parameters in step 1 so that they can be used here.

Step 5 - Prediction head and forward pass

```python
self.action_head = nn.Linear(embed_dim, act_dim)

def forward(self, states, actions, returns, timesteps):
    batch_size = states.shape[0]

    # Embeddings
    state_emb = self.state_embed(states)
    action_emb = self.action_embed(actions)
    return_emb = self.return_embed(returns.unsqueeze(-1))
    time_emb = self.time_embed(timesteps)

    # Combine embeddings
    x = state_emb + action_emb + return_emb + time_emb

    # Transformer
    x = self.transformer(x)

    # Predict next action
    action_logits = self.action_head(x)
    return action_logits
```

Predicts the next action based on transformer embeddings

Creates embedding for states, actions, returns and timestep through function defined in step 3.

Adds them together

Passes it through transformer encoder shown in step 4.

Predicts next action using action head.

Data handling Steps -

```python
class TrajectoryDataset(Dataset):
    def __init__(self, trajectories, seq_len=50):
        self.seq_len = seq_len
        self.data = []

        for traj in trajectories:
            states = traj['states']
            actions = traj['actions']
            returns = traj['returns_to_go']

            for i in range(len(states) - seq_len):
                self.data.append((
                    states[i:i + seq_len],
                    actions[i:i + seq_len],
                    returns[i:i + seq_len],
                    np.arange(i, i + seq_len)
                ))
```

Step I -

trajectories is list of saved episode data and stores state - action - return - timestep for seq length.

```python
    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        states, actions, returns, timesteps = self.data[idx]
        return (
            torch.FloatTensor(states),
            torch.LongTensor(actions),
            torch.FloatTensor(returns),
            torch.LongTensor(timesteps)
        )
```

Step II - Converting NumPy arrays into pytorch tensors

Model Training -

```python
def train():
    with open('cartpole_dt_dataset_iter2.pkl', 'rb') as f:
        trajectories = pickle.load(f)
```

The data is loaded from pkl file.

```python
dataset = TrajectoryDataset(trajectories)
dataloader = DataLoader(dataset, batch_size=256, shuffle=True, pin_memory=True)
```

The dataset runs through above trajectory function to format it as per required seq length

```python
model = DecisionTransformer().to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
criterion = nn.CrossEntropyLoss()
```

The model runs over gpu. The model is initialised through Adam optimiser. The loss is called as per cross entropy loss as actions are categorical.

```python
print(f"Training on {device}...")
for epoch in range(100):
    model.train()
    total_loss = 0

    for states, actions, returns, timesteps in dataloader:
        states, actions, returns, timesteps = states.to(device), actions.to(device), retu
```

Training for 100 epoch on cuda.

```python
preds = model(
    states[:, :-1],   # States up to t-1
    actions[:, :-1],  # Actions up to t-1
    returns[:, :-1],  # Returns up to t-1
    timesteps[:, :-1] # Timesteps up to t-1
)
```

prediction of next action based on previous data. Depends on seq length

```python
loss = criterion(
    preds.reshape(-1, 2),
    actions[:, 1:].reshape(-1)
)
```

entropy loss is calculated

```python
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
optimizer.step()
```

Backpropogation to update the model

Finally, model is saved.