TST_model_testing

May 24, 2023

Testing a pre-trained TST Model Various TST models for crystallization (i.e., CrystalGPT) were pretrained and fine-tuned for certain tasks. This file imports the model of choice, and tests them for a new unseen crystal system to 1. Visualize attention scores 2. Compare time-series forecasting results

[ ]:
```
import numpy as np
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
from sklearn.metrics import r2_score
import pandas as pd
import math, random
import torch, os
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'Current Device: {device}')

from matplotlib import pyplot as plt
import seaborn as sns
```

```
Current Device: cpu
```

[ ]:
```
# Choice of Model
model_name_1 = 'crysGPT_double_12_fine_tuned'
```

Importing Test Dataset This dataset will be used to test and compare the different ML models

[ ]:
```
window_size, prediction_horizon = 12, 6
chunk_size = int(prediction_horizon/2)
input_features, output_features = 6, 4
batch_size = 16 # batch_size for fine-tuning much be less as it is ensures more␣
 ↪iterations during training
```

A new and unseen crystal system is imported that has data for ~7000 operating conditions. Each of the operating conditions are arbitary with varying temperature, concentration, seeding, and other characteristis, thereby encompasing a very large variable space.

[ ]:
```
size_single_dataset = 145 #Max lenght of an individual dataset | 7000 of such␣
 ↪are generated.
```

```
df_one = pd.read_csv('LSTM_controller_training_6000_dataset.csv')
df_two = pd.read_csv('ML_model_testing_7000_dataset.csv')
# Concatenate the two data frames vertically
df_combined = pd.concat([df_one, df_two], axis=0, ignore_index=True)
print(f'Shape of ORIGINAL dataset: {df_combined.shape}')

###remove error and setpoint columns
df_combined = df_combined.iloc[:,0:-2]
df_combined = df_combined[['jacket_temp',␣
↪'concentration','temperature','crystal_size','suspension_density','time']]
# Rename columns
df_combined = df_combined.rename(columns={'jacket_temp': 'T_jacket',␣
↪'concentration': 'conc.', 'temperature': 'temp.', 'crystal_size': 'SMD'})
# df_combined =df_combined.round(1)
df_combined.head()

# df_combined=df_combined.iloc[0:data_limiter, :]
print(f'Shape of CUT_DOWN dataset: {df_combined.shape}')
```

```
Shape of ORIGINAL dataset: (725000, 11)
Shape of CUT_DOWN dataset: (725000, 6)
```

[ ]:
```
# Load normalization parameters
norm_params = np.load('norm_params.npy', allow_pickle=True).item()

mean = norm_params['mean']
std = norm_params['std']

df_combined_norm = (df_combined - mean) / std

print(f'Mean values: {mean}')
print(f'Std. values: {std}')

#Print normalized df
df_combined_norm.head()
```

```
Mean values: T_jacket                  28.609945
conc.                       0.157804
temp.                      28.899302
SMD                       198.910277
suspension_density          0.669336
time                    43200.000000
dtype: float64
Std. values: T_jacket                   7.783112
conc.                       0.169761
temp.                       7.611891
SMD                        59.447470
```

```
suspension_density        0.190218
time                  25114.140310
dtype: float64
```

[ ]:

	T_jacket	conc.	temp.	SMD	suspension_density	time
0	0.680969	2.816002	0.658272	-1.019855	-2.861645	-1.720146
1	0.680969	2.789907	0.658272	-0.992893	-2.838355	-1.696256
2	0.680969	2.763459	0.658272	-0.966188	-2.814751	-1.672365
3	0.680969	2.736676	0.658272	-0.939748	-2.790848	-1.648474
4	0.680969	2.709574	0.658272	-0.913582	-2.766660	-1.624583

We can select a random operating condition from all the 7000 ones. Also, for time-series forecasting, a window size of W is required to provide as the input tensor, and get predictions over the next H time-steps.

[ ]:

```
data_set_selector = random.randrange(0, len(df_combined_norm),␣
 ↪size_single_dataset)
sample_df = df_combined_norm.iloc[data_set_selector:
 ↪data_set_selector+size_single_dataset,:]

# Generating input tensor of a certain window size (W)

def df_to_X_y(df_train, input_window_size, output_window_size):

    df_train_labels = df_train.iloc[:,1:]

    df_as_np_x = df_train.to_numpy()
    df_as_np_y = df_train_labels.to_numpy()

    X = []
    y = []
    flag = True
    for i in range(len(df_as_np_x)-2*output_window_size):

        if ((i+window_size+output_window_size-1)%size_single_dataset ==0):
            flag =False
        elif (i%size_single_dataset ==0):
            flag=True

        if (flag):
            row = [a for a in df_as_np_x[i:i+input_window_size]]
            X.append(row)
            label = [b for b in df_as_np_y[i+output_window_size:
 ↪int(i+output_window_size+output_window_size)]]
            y.append(label)

    y = np.array(y)
```

```
    y_outputs = y[:,:,0:-1]
    time_stamp  = y[:,:,-1]

    return np.array(X), np.array(y_outputs), np.array(time_stamp)


X_enc_combined,y_combined, time_array = df_to_X_y(sample_df,
 ↪window_size,prediction_horizon)
X_dec_combined = np.concatenate((X_enc_combined[:,-chunk_size:,1:-1],
 ↪y_combined[:,:chunk_size,:]),axis=1)
#Predicting Y = [Temp, Conc, Size] using X= [Temp, Conc, Size, Time]
print('X_enc Shape: ', X_enc_combined.shape)
print('X_dec Shape: ', X_dec_combined.shape)
print('y Shape: ', y_combined.shape)
print(f'time array Shape: {time_array.shape}')

# %%
##### Converting to TORCH TENSOR ##########
X_enc_combined = torch.tensor(X_enc_combined, dtype=torch.float32)
X_dec_combined= torch.tensor(X_dec_combined,dtype=torch.float32)
y_combined = torch.tensor(y_combined,dtype=torch.float32)
```

```
X_enc Shape:  (128, 12, 6)
X_dec Shape:  (128, 6, 4)
y Shape:  (128, 6, 4)
time array Shape: (128, 6)
```

Importing the Base TST Model Although we have chosen a certain TST model at the top, it is in form of a .pt file, which only has the learned parameters of the model. Thus, the structure of the model (in pytorch form) is required to be fed in the code.

Note: I do have a separate file (TST.py), which does this, and keeps it modular. However, since I am combining the code for Tesla, I have included it here for easy acccess.

[ ]:
```
######## THESE ARE DUMMY PARAMETERS (Just to Initialize the TST)
 ↪################
# The actual parameters of the trained model will be called from the checkpoint
 ↪(model) file
num_encoder_blocks = 1
num_decoder_blocks = 1
nhead = 16
dim_FFN = 16
d_model = 16
dropout = 0


class EmbeddingLayer(nn.Module):
```

```python
    def __init__(self, input_size, d_model):
        super(EmbeddingLayer, self).__init__()
        self.fc = nn.Linear(input_size, d_model)

    def forward(self, x):
        x = self.fc(x)
        return x


# %% [markdown]
# Positional Encoding (PE): Sinusoidal

# %%
class PositionalEncoding(nn.Module):  #@save
    """Positional encoding."""
    def __init__(self, d_model, dropout, max_len=1000):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        # Create a long enough P
        self.P = torch.zeros((1, max_len, d_model))
        position = torch.arange(max_len, dtype=torch.float32).reshape(-1, 1)
        div_term = torch.pow(10000, torch.arange(0, d_model, 2, dtype=torch.
 ↪float32) / d_model)
        X = position/div_term

        # X = torch.arange(max_len, dtype=torch.float32).reshape(
        #     -1, 1) / torch.pow(10000, torch.arange(
        #     0, d_model, 2, dtype=torch.float32) / d_model)

        self.P[:, :, 0::2] = torch.sin(X)
        self.P[:, :, 1::2] = torch.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].to(X.device)
        return self.dropout(X)

# %% [markdown]
# Single Encoder Block

# %%
class TransformerEncoderBlock(nn.Module):  #@save
    """The Transformer encoder block."""
    def __init__(self, d_model, nhead, dim_feedforward, dropout):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(d_model, nhead,␣
 ↪dropout=dropout,batch_first=True)
        self.feedforward = nn.Sequential(
```

```
            nn.Linear(d_model, dim_feedforward),
            nn.ReLU(),
            nn.Dropout(p=dropout),
            nn.Linear(dim_feedforward, d_model),
            nn.Dropout(p=dropout)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(p=dropout)
        self.dropout2 = nn.Dropout(p=dropout)


    def forward(self, x):
        residual = x
        x = self.norm1(x)
        x, self_attn_weights = self.self_attn(x, x, x)
        x = self.dropout1(x)
        x = residual + x

        residual = x
        x = self.norm2(x)
        x = self.feedforward(x)
        x = self.dropout2(x)
        x = residual + x
        return x, self_attn_weights


# %% [markdown]
# Multiple Encoder Block

# %%
class Multiple_Encoders(nn.Module):
    def __init__(self, input_size, output_size, d_model, nhead, dim_feedforward, num_encoder_blocks, w, h, dropout):
        super(Multiple_Encoders, self).__init__()

        self.w = w
        self.h = h

        self.encoder_blocks = nn.ModuleList([TransformerEncoderBlock(d_model, nhead, dim_feedforward,dropout=dropout) for _ in range(num_encoder_blocks)])

        self.encoder_linear = EmbeddingLayer(input_size, d_model)

        self.pos_encoder = PositionalEncoding(d_model,max_len = self.w,dropout=dropout)
```

```
    def forward(self, x):
        self_attn_weights_array = []  # List to store cross-attention weights
        x = self.encoder_linear(x)  # Pass through the encoder linear layer
        x = self.pos_encoder(x)  # Add positional encoding
        for block in self.encoder_blocks:
            x, self_attn_weights = block(x)  # Pass through the encoder␣
↪TSTBlock modules
            self_attn_weights_array.append(self_attn_weights)  # Save the␣
↪cross-attention weights
        return x, self_attn_weights_array


# %% [markdown]
# Single Decoder Block

# %%
class TransformerDecoderBlock(nn.Module):  #@save
    """The Transformer Decoder Block."""
    def __init__(self, d_model, nhead, dim_feedforward, dropout):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(d_model, nhead,␣
↪dropout=dropout,batch_first=True)
        self.cross_attn = nn.MultiheadAttention(d_model, nhead,␣
↪dropout=dropout,batch_first=True)
        self.feedforward = nn.Sequential(
            nn.Linear(d_model, dim_feedforward),
            nn.ReLU(),
            nn.Dropout(p=dropout),
            nn.Linear(dim_feedforward, d_model),
            nn.Dropout(p=dropout)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)

        self.dropout1 = nn.Dropout(p=dropout)
        self.dropout2 = nn.Dropout(p=dropout)
        self.dropout3 = nn.Dropout(p=dropout)


    def forward(self, x, memory):
        residual = x
        x = self.norm1(x)
        x, _ = self.self_attn(x, x, x)
        x = self.dropout1(x)
        x = residual + x
```

```python
        residual = x
        x = self.norm2(x)
        x, cross_attn_weights  = self.cross_attn(x, memory, memory)
        x = self.dropout2(x)
        x = residual + x

        residual = x
        x = self.norm3(x)
        x = self.feedforward(x)
        x = self.dropout3(x)
        x = residual + x
        return x, cross_attn_weights



# %% [markdown]
# Multiple Decoder Blocks

# %%
class Multiple_Decoders(nn.Module):
    def __init__(self, input_size, output_size, d_model, nhead,␣
 ↪dim_feedforward, num_decoder_blocks, w, h, chunk_size, dropout):
        super(Multiple_Decoders, self).__init__()

        self.w = w
        self.h = h

        self.decoder_blocks = nn.ModuleList([TransformerDecoderBlock(d_model,␣
 ↪nhead, dim_feedforward,dropout=dropout) for _ in range(num_decoder_blocks)])
        # self.cross_attn_weights = []  # List to store cross-attention weights

        self.decoder_linear = EmbeddingLayer(output_size, d_model)

        self.pos_decoder = PositionalEncoding(d_model,max_len =␣
 ↪2*chunk_size,dropout=dropout)

    def forward(self, x, memory):
        cross_attn_weights_array = []  # List to store cross-attention weights
        x = self.decoder_linear(x)  # Pass through the encoder linear layer
        x = self.pos_decoder(x)  # Add positional encoding
        for block in self.decoder_blocks:
            x, cross_attn_weights = block(x, memory)  # Pass through the␣
 ↪decoder TSTBlock modules
            cross_attn_weights_array.append(cross_attn_weights)  # Save the␣
 ↪cross-attention weights
        return x,cross_attn_weights_array
```

```python
# %% [markdown]
# Combining all Elements for TST Model

# %%
class TimeSeriesTransformer(nn.Module):
    def __init__(self, input_size, output_size, d_model=d_model, nhead=nhead,␣
↪dim_feedforward=dim_FFN,
                 num_encoder_blocks=num_encoder_blocks,␣
↪num_decoder_blocks=num_decoder_blocks,
                 dropout=dropout,
                 window_size=window_size,␣
↪prediction_horizon=prediction_horizon, chunk_size = chunk_size):

        super(TimeSeriesTransformer, self).__init__()

        # Set the model parameters
        self.d_model = d_model
        self.nhead = nhead
        self.num_encoder_blocks = num_encoder_blocks
        self.num_decoder_blocks = num_decoder_blocks
        self.window_size = window_size
        self.prediction_horizon = prediction_horizon

        # Encoder blocks
        self.multiple_encoder_blocks =␣
↪Multiple_Encoders(input_size=input_features,output_size=output_features,
                                                                     ␣
↪d_model=d_model,nhead=nhead,dim_feedforward=dim_FFN
                                  ,num_encoder_blocks=num_encoder_blocks,
                                  w=window_size,h=prediction_horizon,
                                  dropout=dropout)

        # Decoder blocks
        self.multiple_decoder_blocks =␣
↪Multiple_Decoders(input_size=input_features, output_size=output_features,
                                                                              ␣
↪d_model=d_model,nhead=nhead,dim_feedforward=dim_FFN,
                                                                              ␣
↪num_decoder_blocks=num_decoder_blocks,
                                           w=window_size,h=prediction_horizon,␣
↪chunk_size=chunk_size,
                                           dropout=dropout)

        # self.cross_attention_weights =
```

```
        # Output layer
        self.output_FFN = EmbeddingLayer(d_model,output_features)

    def forward(self, x_enc, x_dec):
        # Encode the input time series data
        encoder_output, self_attn_weights_array = self.
↪multiple_encoder_blocks(x_enc)

        # Decode the input time series data
        decoder_output, cross_attn_weights_array = self.
↪multiple_decoder_blocks(x_dec, encoder_output[:, -(2*chunk_size):, :])
        # cross_attention_weights = self.multiple_decoder_blocks.
↪cross_attn_weights

        # Pass the decoded output through the output layer
        final_output = self.output_FFN(decoder_output)

        # Return the predicted values
        return final_output, self_attn_weights_array, cross_attn_weights_array
```

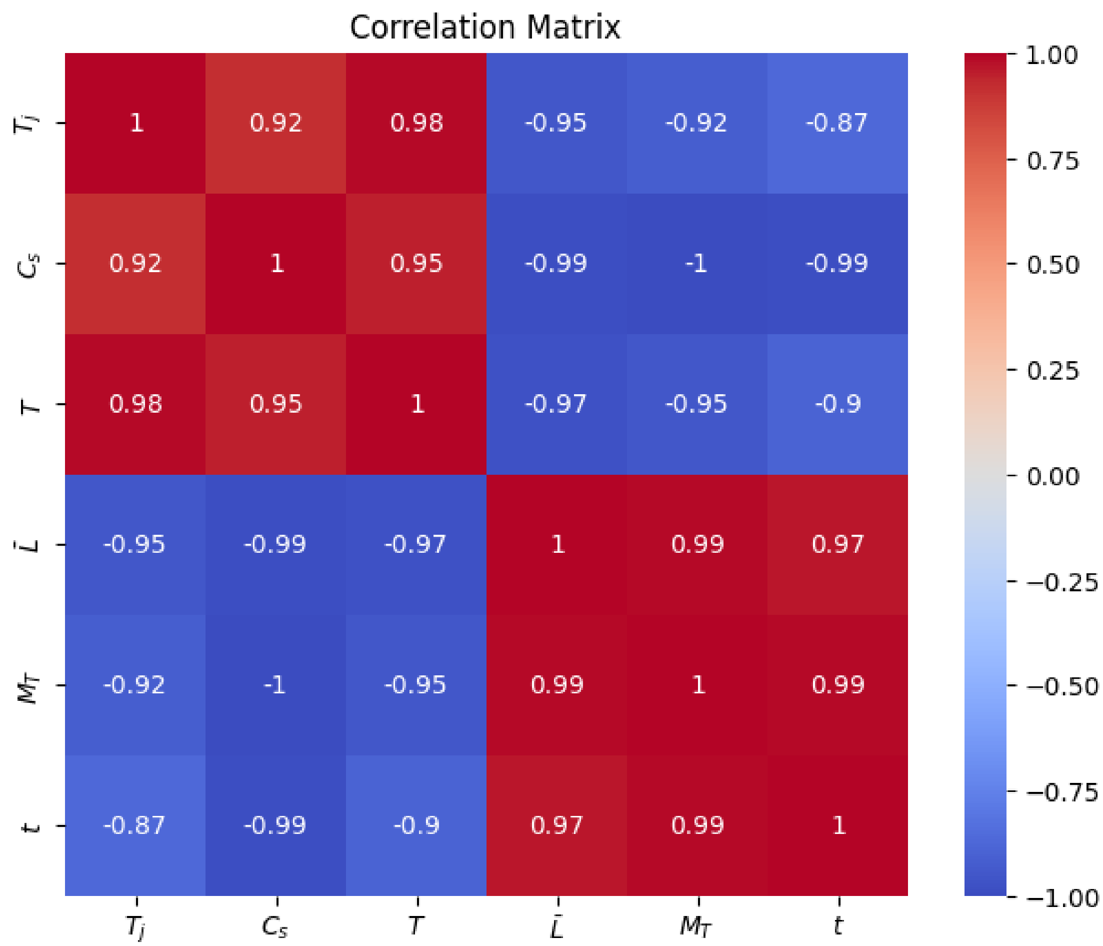
[ ]:
```
model_1 = torch.load(f'{model_name_1}.pt', map_location=device)
model_1.to(device)
print(f'Current Model: {model_name_1}.pt')


########## FIND NUMBER OF PARAMETERS ##############
def count_parameters(model_1):
    return sum(p.numel() for p in model_1.parameters() if p.requires_grad)

num_params = int(count_parameters(model_1)/1000)
print(f"Model 1: Number of parameters: {num_params}K")
```

```
Current Model: crysGPT_double_12_fine_tuned.pt
Model 1: Number of parameters: 10547K
```

PCA Scores Lets first simple plot heatmaps of different features to understand their relative importance

[ ]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Reshape the tensor to [batch * w, features]
X = X_enc_combined
X_reshaped = X.reshape(-1, X.shape[-1])
```

```
# Compute the correlation matrix
correlation_matrix = np.corrcoef(X_reshaped, rowvar=False)

# Plot the correlation matrix as a heatmap
plt.figure(figsize=(8, 6),dpi = 100)
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', square=True)
plt.title('Correlation Matrix')

# Set the x and y tick labels
# Set the x and y tick labels
feature_names = ['$T_j$', '$C_s$', '$T$', r'$\bar{L}$', '$M_T$', '$t$']
plt.xticks(np.arange(len(feature_names)) + 0.5, labels=feature_names,␣
 ↪ha='center')
plt.yticks(np.arange(len(feature_names)) + 0.5, labels=feature_names,␣
 ↪va='center')
plt.show()
```



Visualizing Attention Scores We can visualize self-attention scores (in encoders), and cross-attention scores (in decoders).

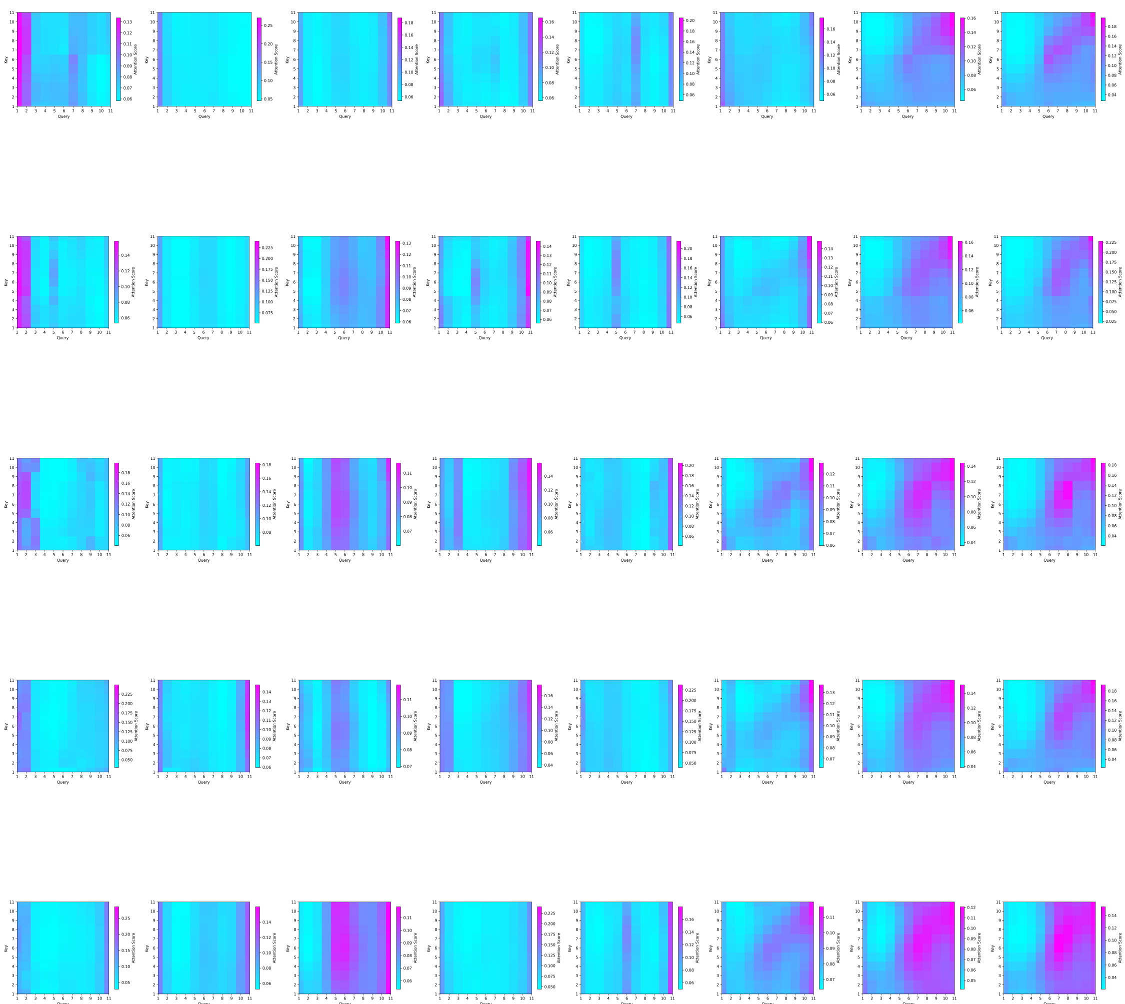
Encoder Self Attention

Randomly choosing an operating condition from the dextrose fine tuning dataset, and visualizing the attention scores for it.

```
[ ]: with torch.no_grad():
         y_preds, self_attn_weights_array, cross_attn_weights_array =␣
      ↪model_1(X_enc_combined,X_dec_combined)


     num_blocks = len(self_attn_weights_array)
     # Create a single figure with subplots
     # Generate an array of alpha values
     alpha_array = np.linspace(1, 1, num_blocks)
     batch_index = np.round(np.linspace(0, len(X_enc_combined) - 1, 5)).astype(int).
      ↪tolist()
     # batch_index = random.randint(0, len(X_enc_combined) - 1

     print(f"Time Index: {np.round(np.array(batch_index)*10/60)} h.")

     for batch in batch_index:
         # Iterate over the attention weights and plot them
         fig, axes = plt.subplots(1, num_blocks, figsize=(num_blocks * 5,10),dpi =␣
      ↪300)
         fig.set_label(f'Time Step: {batch}')
         for i, weights in enumerate(self_attn_weights_array):
             # Select the attention weights for the specified batch
             weights = weights[batch, :, :]

             # Select the corresponding subplot
             ax = axes[i]

             # Plot the attention weights with the specified alpha value
             img = ax.imshow(weights.squeeze(0).detach().numpy(), cmap='cool',␣
      ↪interpolation='nearest', alpha=alpha_array[i])
             # ax.set_title('Self-Attention Weights (Decoder Block {})'.format(i +␣
      ↪1))
             ax.set_xlabel('Query')
             ax.set_ylabel('Key')

             # Set xlim and ylim to match weights.shape[-1]
             xlim = weights.shape[-1]
             ylim = weights.shape[-1]
             ax.set_xlim(1, xlim-1)
             ax.set_ylim(1, ylim-1)
```

```
        # Set xticks and yticks to display exact integers
        ax.set_xticks(np.arange(1, xlim, 1))
        ax.set_yticks(np.arange(1, ylim, 1))

        # Add a color bar to the subplot
        cbar = fig.colorbar(img, ax=ax, shrink=0.3)
        cbar.ax.set_ylabel('Attention Score')

    plt.tight_layout()
    plt.show()
```

```
Time Index: [ 0.  5. 11. 16. 21.] h.
```



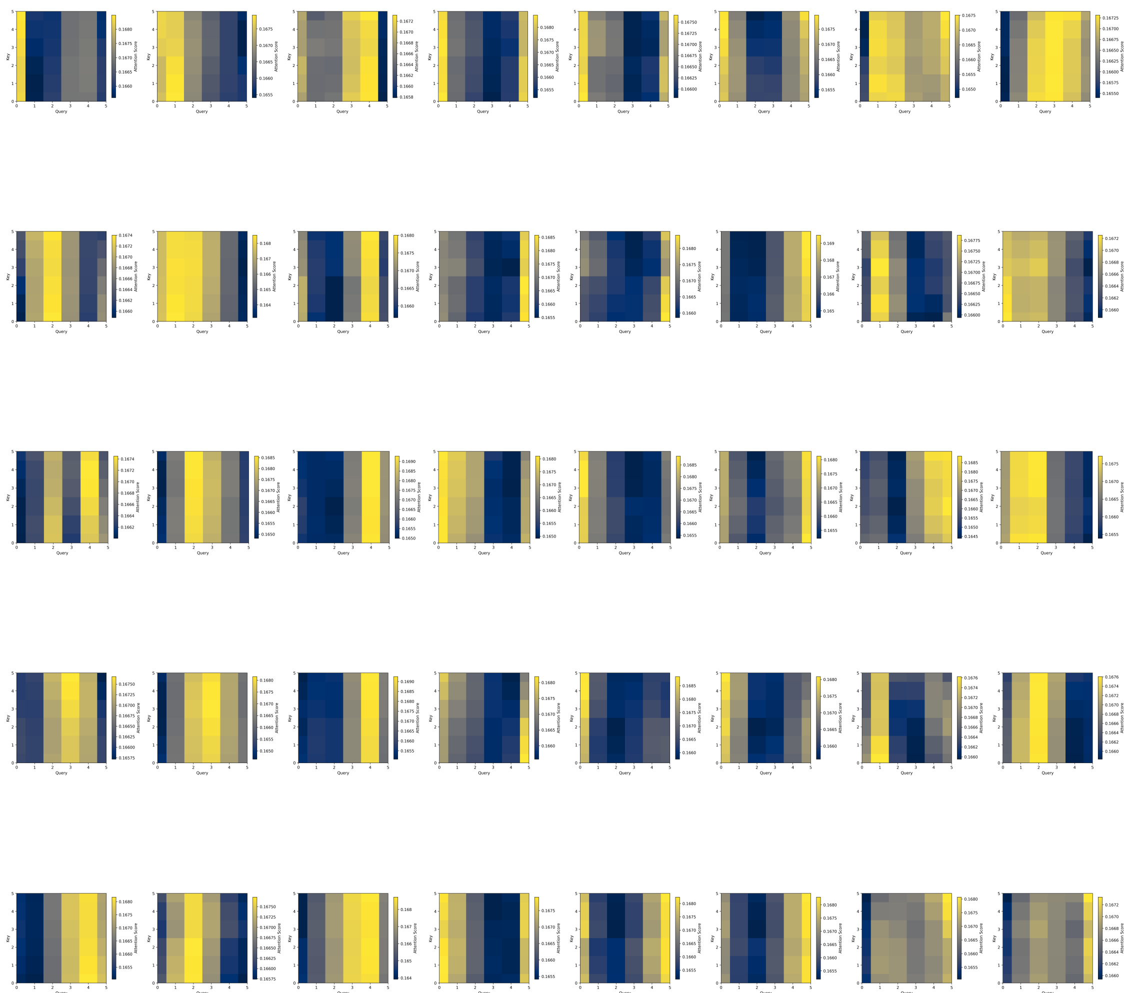
Decoder Cross Attention

Randomly choosing an operating condition from the dextrose fine tuning dataset, and visualizing the attention scores for it.

```
[ ]: with torch.no_grad():
         y_preds, self_attn_weights_array, cross_attn_weights_array = 
      ↪model_1(X_enc_combined,X_dec_combined)



     num_blocks = len(cross_attn_weights_array)
     # Create a single figure with subplots
     # Generate an array of alpha values
     alpha_array = np.linspace(1, 1, num_blocks)
     batch_index = np.round(np.linspace(0, len(X_enc_combined) - 1, 5)).astype(int).
      ↪tolist()
     # batch_index = random.randint(0, len(X_enc_combined) - 1

     print(f"Time Index: {np.round(np.array(batch_index)*10/60)} h.")

     for batch in batch_index:
         # Iterate over the attention weights and plot them
         fig, axes = plt.subplots(1, num_blocks, figsize=(num_blocks * 5,10),dpi =
      ↪300)
         fig.set_label(f'Time Step: {batch}')
         for i, weights in enumerate(cross_attn_weights_array):
             # Select the attention weights for the specified batch
             weights = weights[batch, :, :]

             # Select the corresponding subplot
             ax = axes[i]

             # Plot the attention weights with the specified alpha value
             img = ax.imshow(weights.squeeze(0).detach().numpy(), cmap='cividis',
      ↪interpolation='nearest', alpha=alpha_array[i])
             # ax.set_title('Cross-Attention Weights (Decoder Block {})'.format(i +
      ↪1))
             ax.set_xlabel('Query')
             ax.set_ylabel('Key')

             # Set xlim and ylim to match weights.shape[-1]
             xlim = weights.shape[-1]
             ylim = weights.shape[-1]
             ax.set_xlim(1, xlim-1)
             ax.set_ylim(1, ylim-1)

             # Set xticks and yticks to display exact integers
             ax.set_xticks(np.arange(0, xlim, 1))
             ax.set_yticks(np.arange(0, ylim, 1))
```

```
        # Add a color bar to the subplot
        cbar = fig.colorbar(img, ax=ax, shrink=0.3)
        cbar.ax.set_ylabel('Attention Score')

    plt.tight_layout()
    plt.show()
```

```
Time Index: [ 0.  5. 11. 16. 21.] h.
```



Plotting Model Predictions This section performs model validation (against experimental dataset) for key model predictions (i.e., temperature, concenttration, crystal size, and others)

[ ]:
```
# Plotting and error calculations
from sklearn.metrics import mean_squared_error as MSE
```

```python
def single_data_set_predictor(sample_model, norm_params, X_enc,X_dec,␣
 ↪y_actuals, time_array, start=0, end=100):

    mean = np.array(norm_params['mean'])
    std = np.array(norm_params['std'])

    # Convert input data to PyTorch tensors
    # X_enc_tensor = torch.from_numpy(X_enc[start:end,:,:]).float()
    # X_dec_tensor = torch.from_numpy(X_dec[start:end,:,:]).float()
    # y_tensor = torch.from_numpy(y_actuals[start:end,:,:]).float()
    time= time_array[start:end,-1]

    # Generate predictions for y using model
    with torch.no_grad():
        y_preds, _, _ = sample_model(X_enc,X_dec)
        y_preds = y_preds.detach().numpy()

    df_preds=pd.DataFrame()
    df_preds['pred_concentration'] = y_preds[:,-1,0]
    df_preds['pred_temperature'] = y_preds[:,-1,1]
    df_preds['pred_size'] = y_preds[:,-1,2]
    df_preds['pred_suspension_density'] = y_preds[:,-1,3]
    df_preds['time'] = time

    df_actuals = pd.DataFrame()
    df_actuals['concentration'] = y_actuals[:,-1,0]
    df_actuals['temperature'] = y_actuals[:,-1,1]
    df_actuals['size'] = y_actuals[:,-1,2]
    df_actuals['suspension_density'] = y_actuals[:,-1,3]
    df_actuals['time'] = time

    df_actuals = df_actuals*std[1:]+ mean[1:]
    df_preds = df_preds * std[1:] + mean[1:]

    return df_preds, df_actuals
```

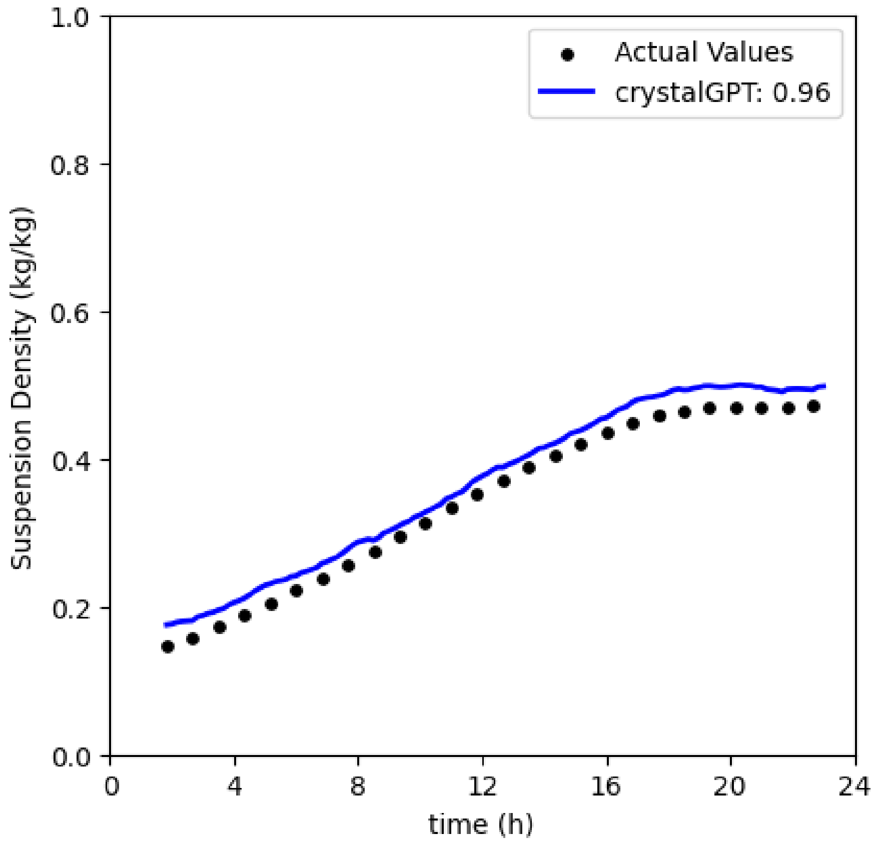
[ ]:
```python
model_choices = [model_1]  # List of model choices
model_names = ['crystalGPT']  # List of model names

colors = ['blue', 'magenta', 'blue', 'green']  # List of colors for plotting
linestyles = ['-', '--', '-.', ':']  # List of line styles for plotting
linewidths = [2, 3, 2, 1]  # List of line widths for plotting
dpi = 100  # Dots per inch for the figure

data_set = {}  # Dictionary to store the data sets
for name in np.arange(len(model_names)):
```

```
    # Predict using the specified model and obtain the data set
    data_set[str(model_names[name])], df_actuals = 
↪single_data_set_predictor(model_choices[name],
                                                                                     
↪norm_params,
                                                                                     
↪X_enc_combined,
                                                                                     
↪X_dec_combined,
                                                                                     
↪y_combined,
                                                                                     
↪time_array, 0, len(y_combined))

# Augment with actual data
data_set['Actual'] = df_actuals  # Add the actual data set to the dictionary
```

0.0.1 Suspension Density Plot

[ ]:
```
### Plotting on the same graph for comparison ######
from matplotlib.pyplot import figure

goodness_of_fit = {}
plt.figure(dpi=dpi, figsize=(5,5))
plt.scatter(df_actuals['time'][::5]/3600, df_actuals['suspension_density'][::
↪5],label='Actual Values'  ,color='black',alpha=1,marker = 'o',s=15)

for name in np.arange(len(model_names)):
    temp_df = data_set[model_names[name]]
    goodness_of_fit[model_names[name]] = 
↪r2_score(df_actuals['suspension_density'], 
↪temp_df['pred_suspension_density'])
    label_name = model_names[name] + ': ' + 
↪str(round(r2_score(df_actuals['suspension_density'], 
↪temp_df['pred_suspension_density']),3))
    plt.plot(temp_df['time']/3600, 
↪temp_df['pred_suspension_density'],label=label_name,
             color = colors[name],
             linewidth = linewidths[name],
             linestyle= linestyles[name])

plt.xlabel('time (h)')
plt.ylabel('Suspension Density (kg/kg)')
x_label = np.arange(0,25,step = 4)
plt.xlim([0,24])
plt.xticks(x_label)
```

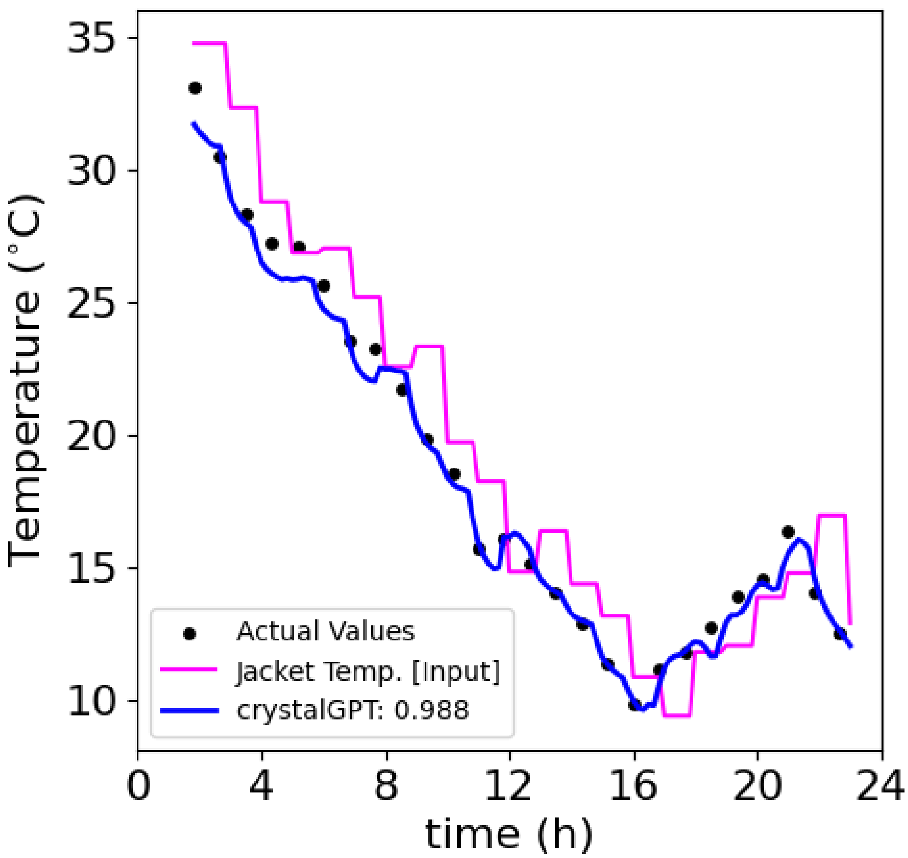
```
plt.ylim([0,1])
plt.legend(loc = 'upper right')
plt.show()
```



0.0.2 Temperature Curve

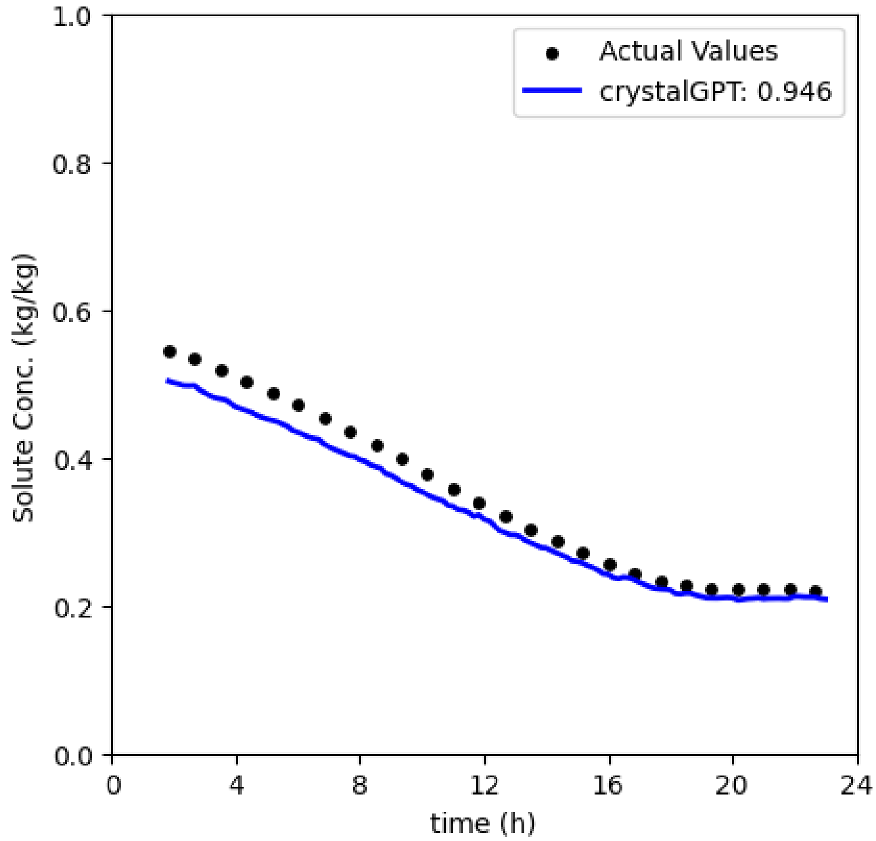
[ ]:
```
### Plotting on the same graph for comparison ######
from matplotlib.pyplot import figure
# figure(figsize=(4,4), dpi=300)


jacket_temp = X_enc_combined[:,0,0]*std[0] + mean[0]

plt.figure(dpi=dpi, figsize=(5,5))
plt.scatter(df_actuals['time'][::5]/3600, df_actuals['temperature'][::
 ↪5],label='Actual Values'  ,color='black',alpha=1,marker = 'o',s=15)
plt.plot(df_actuals['time']/3600, jacket_temp,label='Jacket Temp.␣
 ↪[Input]',color = 'magenta')
```

```
for name in np.arange(len(model_names)):
    temp_df = data_set[model_names[name]]
    goodness_of_fit[model_names[name]] = r2_score(df_actuals['temperature'],␣
 ↪temp_df['pred_temperature'])
    label_name = model_names[name] + ': ' +␣
 ↪str(round(r2_score(df_actuals['temperature'],␣
 ↪temp_df['pred_temperature']),3))
    plt.plot(temp_df['time']/3600,␣
 ↪temp_df['pred_temperature'],label=label_name,color = colors[name],
             linewidth = linewidths[name],
             linestyle= linestyles[name])

plt.xlabel('time (h)', fontsize=16)
plt.ylabel('Temperature ($^{\circ}$C)', fontsize=16)
x_label = np.arange(0, 25, step=4)
plt.xlim([0, 24])
# plt.ylim([25, 45])
plt.xticks(x_label, fontsize=16)
plt.yticks(fontsize=16)
plt.legend(loc='lower left')
plt.show()
```



0.0.3 Concentration Curve

[ ]:
```
### Plotting on the same graph for comparison ######
from matplotlib.pyplot import figure
# figure(figsize=(4,4), dpi=300)

goodness_of_fit = {}
plt.figure(dpi=dpi, figsize=(5,5))
plt.scatter(df_actuals['time'][::5]/3600, df_actuals['concentration'][::
 ↪5],label='Actual Values'  ,color='black',alpha=1,marker = 'o',s=15)

for name in np.arange(len(model_names)):
    temp_df = data_set[model_names[name]]
    goodness_of_fit[model_names[name]] = r2_score(df_actuals['concentration'],␣
 ↪temp_df['pred_concentration'])
    label_name = model_names[name] + ': ' +␣
 ↪str(round(r2_score(df_actuals['concentration'],␣
 ↪temp_df['pred_concentration']),3))
    plt.plot(temp_df['time']/3600,␣
 ↪temp_df['pred_concentration'],label=label_name,color = colors[name],
             linewidth = linewidths[name],
             linestyle= linestyles[name])

plt.xlabel('time (h)')
plt.ylabel('Solute Conc. (kg/kg)')
x_label = np.arange(0,25,step = 4)
plt.xlim([0,24])
plt.ylim([0,1])
plt.xticks(x_label)
plt.legend(loc = 'upper right')
plt.show()
```



0.0.4 Crystal Size Evolution

[ ]:
```
### Plotting on the same graph for comparison ######
from matplotlib.pyplot import figure
# figure(figsize=(4,4), dpi=300)

goodness_of_fit = {}
plt.figure(dpi=dpi, figsize=(5,5))
plt.scatter(df_actuals['time'][::5]/3600, df_actuals['size'][::5],label='Actual␣
↪Values'  ,color='black',alpha=1,marker = 'o',s=15)

for name in np.arange(len(model_names)):
    temp_df = data_set[model_names[name]]
    goodness_of_fit[model_names[name]] = r2_score(df_actuals['size'],␣
↪temp_df['pred_size'])
    label_name = model_names[name] + ': ' +␣
↪str(round(r2_score(df_actuals['size'], temp_df['pred_size']),3))
    plt.plot(temp_df['time']/3600, temp_df['pred_size'],label=label_name,color␣
↪= colors[name],
```

```
            linewidth = linewidths[name],
            linestyle= linestyles[name])

plt.xlabel('time (h)', fontsize=16)
plt.ylabel('Crystal Size ($\mu m$)', fontsize=16)
x_label = np.arange(0, 25, step=4)
plt.xlim([0, 24])
# plt.ylim([100, 180])
plt.xticks(x_label, fontsize=16)
plt.yticks(fontsize=16)
plt.legend(loc='lower right')
plt.show()
```