Question #	Answer
1	The AI will not generalize well to real-life. Given the small size of the dataset (N=1464), I do not think that there are sufficient data points for the AI to generalize well to real-life. Furthermore, the dataset only accounts for 61 days, which is about 2 months, so it might not be able to capture all the seasons in the year, so can't really predict all weeks in the year.
2	There are 1272 samples.
3	There are two tensors that will be returned. The size of the input tensor is 168, which corresponds to total number of hours in a week (7 days) before index time, t. The size of the output tensor is 24, which corresponds to the rainfall data for 24 hours (a day) of rainfall data after index time, t.
4	Advantages of larger batch size: 1. Given an increase in batch size, our optimizer will use a larger batch size to perform stochastic gradient descent, which likely means better estimation of the MSE loss. 2. With more samples per batch, the computed gradient is a better approximation of the true gradient. It is likely there will be a smoother convergence of the training loss graph. It is likely there are more stable updates, reducing variance in optimization. Disadvantages of larger batch size: 1. A larger batch has a higher memory usage. 2. With a larger batch size there might be slower convergence. Small batches introduce noise in gradient updates, which can help escape local minima, whereas large batches can get such in sharp minima.
5	The value 10 corresponds to the number of batches in the Dataloader, i.e. the samples in the dataset are divided into 10 different batches, with each batch ideally the size of 128. Given that there are 1272 samples in the dataset and since we set each batch_size as 128, therefore, 1272/128 = 9.9375 ~= 10 (shown) Also, technically, there are 9 full batches of 128 samples, and 1 last batch with 120 samples.
6	A Seq2Seq model is a special type of many-to-many model. Its key difference relies in separating the analysis of the input sequence and the production of an output after seeing all inputs. This is often handled by two different part of a larger neural network, which is where it becomes related to the Encoder-Decoder models. In general, the first part of a Seq2Seq model will focus on analysing the input sequence, eventually producing a final memory vector output after having seen all inputs. This is the encoder part of the model. Furthermore, in general the second part of the model will focus on analysing the produced encoding vector and producing a sequence of some sort as output. This is the decoder part of the model. We want to implement the LSTM architecture drawn below. Its objective is to receives entire $ x(t), x(t+1), ..., x(t+167) $, 168 input points, and learn

	the dynamics of the data, in the hopes that we will later be able to use this information for future predictions.
7	Referencing the architecture above, the encoder model's intermediary outputs (output_0 to output_167) are not used because they each represent a memory vector that is only relevant for that specific point of time or intermediate state. However, this is not what we are interested in, instead we want a a final memory vector output after having seen and analysed all 168 inputs, so we can capture the inherent dependence on time steps for the whole week. The final memory vector contains a compressed representation of the 168 hours of temperature data (prior to time, t). The vector likely captures the temporal patterns and trends. We want our Encoder model to be represented by the EncoderRNN object, whose class prototype is shown below.

8	`#ANSWER FOR QUESTION 8`
	`class EncoderLSTM(nn.Module):`
	`  def __init__(self, input_size, hidden_size):`
	`  super(EncoderLSTM, self).__init__()`
	`  self.hidden_size = hidden_size`
	`  self.lstm = nn.LSTM(input_size, hidden_size, batch_first`
	
	`  def forward(self, inputs):`
	`  output, hidden = self.lstm(inputs)`
	
	`  # Return the final hidden state to be used by the decode`
	`  return hidden`
	✓ 0.0s

9	shape of vec1[0]: torch.Size([128, 64]) Information Contained: vec1[0] is the hidden vector which represents the final state of the LSTM for all 128 samples in the batch, capturing information from the last time step. shape of vec2[0]: torch.Size([128, 64]) Information Contained: vec2[0] is the cell vector which represents the final cell state of the LSTM for all 128 samples in the batch, capturing information from the last time step.
10	We should use the final hidden state and final state produced by the encoder, which it outputs after processing all the inputs $ x(t), x(t+1), ... x(t+167) $.
11	Referencing the architecture diagram above, the input into the LSTM of the auto-regressive decoder would be taken from the immediate output of the

	previous timestamp. To be more specific, at the start the input would be the the final vectors produced by the encoder, afterwhich at each step (from 1 to 24), the model produces an output, which is its prediction of mean temperature for that hour at that timestep.
12	Referencing the architecture diagram above, for k = 0, we should use the $ in_0 = x(t+167) $ produced by the encoder. For subsequent k, we should use $in_k = output_{k-1}$.
13	With reference to the task description in HW3-A, we are trying to predict the next 24 values (a day) , i.e. $ x(t+168), x(t+168+1), ..., x(t+168+23) $.
14	Given the task at hand, the LSTM hidden states $(y_1, y_2, .., y_24)$ are high-dimensional vectors, but the final prediction we want is a single scalar: the temperature for a specific hour. Thus, the linear layer allows for us to generate an output with the desired output space = 1. Referecing the code below, the for loop in the forward method is to generate the output sequence auto-regressively. For each time step, the decoder produces a single output of the sequence. Using a loop, that output is fed back as the input to be used by the decoder to generate the nest output of the sequence for the next time step.

15	`#ANSWER FOR QUESTION 15`
	`class DecoderLSTM(nn.Module):`
	`  def __init__(self, hidden_size, output_size):`
	`  super(DecoderLSTM, self).__init__()`
	`  self.output_size = output_size`
	`  self.lstm = nn.LSTM(output_size, hidden_size, batch_first`
	`  self.linear = nn.Linear(hidden_size, output_size)`
	
	`  def forward(self, x, hidden, output_length, target=None):`
	`  outputs = []`
	`  decoder_input = x`
	`  for i in range(output_length):`
	`  # Generate one time step prediction`
	`  # Feed the current prediction as the next input`
	`  decoder_input, hidden = self.lstm(decoder_input, hidde`
	`  decoder_input = self.linear(decoder_input)`
	`  outputs.append(decoder_input)`
	`  if target is not None:`
	`  decoder_input = target[i].unsqueeze(0)`
	`  return torch.cat(outputs, dim = 0)`

16	Final size of the **decoder_out** = torch.Size([24, 128, 1])
17	Our auto-regressive seq-2-seq model is better able to learn sequential uncertainty and time-step dependencies in outputs. Autoregressive models generate output one step at a time, which each new prediction influenced by the previous one. This is especially beneficial for learning temporal weather patterns as often there might be underlying interconnected weather features that influence the temperature between days, especially for changes between seasons. So, if there are any temporal drifts or

	uncertainty between the hours, the decoder can learn to model step by step. Whereas, in a vanilla LSTM, you have a single model trying to understand the context and generating a forecast based only on its internal state, where errors can accumulate quickly across steps, hurting the model's accuracy & ability to generalise well.

18	`#ANSWER FOR QUESTION 18`
	
	`class Seq2Seq(nn.Module):`
	`  def __init__(self, hidden_size, output_size):`
	`  super(Seq2Seq, self).__init__()`
	`  self.output_length = output_size`
	`  self.encoder = EncoderLSTM(1, hidden_size)`
	`  self.decoder = DecoderLSTM(hidden_size, 1)`
	
	`  def forward(self, inputs, outputs = None):`
	`  hidden = self.encoder(inputs)`
	`  decoder_input = inputs[-1:]`
	`  output = self.decoder(decoder_input, hidden, self.output_length, ta`
	`  return output`

19	`#ANSWER FOR QUESTION 19`
	
	`def train(dataloader, model, num_epochs, learning_rate):`
	`  # Set the model to training mode`
	`  model.train()`
	`  criterion = nn.MSELoss()`
	`  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)`
	
	`  for epoch in range(num_epochs):`
	`  total_loss = 0`
	`  for inputs, outputs in dataloader:`
	`  inputs = inputs.to(device)`
	`  outputs = outputs.to(device)`
	`  inputs = inputs.unsqueeze(-1).permute(1, 0, 2)`
	`  outputs = outputs.unsqueeze(-1).permute(1, 0, 2)`
	`  # Clear previous gradients`
	`  optimizer.zero_grad()`
	`  # Forward pass`
	`  pred = model(inputs, outputs)`
	`  # Calculate loss`
	`  loss = criterion(pred, outputs)`
	`  total_loss += loss.item()`
	`  # Backward pass and optimization`
	`  loss.backward()`
	`  optimizer.step()`
	
	`  # Print total loss every few epochs`
	`  if epoch % 25 == 0:`
	`  print(f'Epoch {epoch + 1}/{num_epochs}, Avg Loss: {total_loss / len(da`

20	Transfer Learning
21	I started with a MAE of 0.9264 using the default code provided. To improve performance of my final model, I tried three experiments, the best configuration was hidden size = 256 with MSE loss and the default learning

rate, achieving an MAE of 0.1673, representing a significant improvement over the baseline.

More on the Experiments:

Experiment #	Variables Changed	MAE	Avg Loss	Discussion
1	Increased hidden size to 256	0.1673	0.01160	A larger hidden size led to a substantial improvement in both loss and MAE, suggesting the model benefited from increased capacity.
2	Hidden size = 256, Loss fn = Huber Loss	0.2262	0.003943	Although the loss decreased, the MAE increased compared to Experiment 1, indicating the model was less accurate despite smoother optimization. Huber loss may be less suitable here.
3	Hidden size = 256, Learning rate = 1e-4	0.2647	0.1066	A smaller learning rate might have led to underfitting or slower convergence (where the training might have stopped before the model could reach the optimal minima), reducing performance.

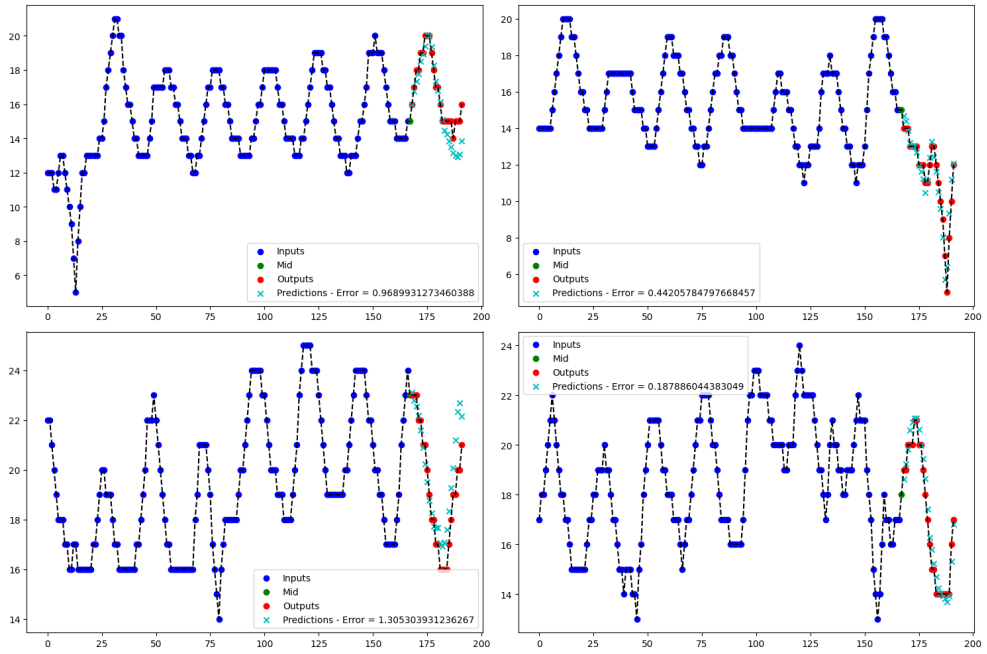
Refer to the screenshots below for the code results.

0. Default configuration provided in the homework results:

```
# Quick check on our Seq2Seq model
# (Seeding for reproducibility)
hidden_size = 64
seq2seq_model = Seq2Seq(hidden_size = hidden_size, \
                        output_size = n_outputs).to(device)
seq2seq_model.load_state_dict(torch.load('seq2seq_model_end.pth'))
seed_value = 187
test_model(seq2seq_model, pt_dataloader, seed_value)
```

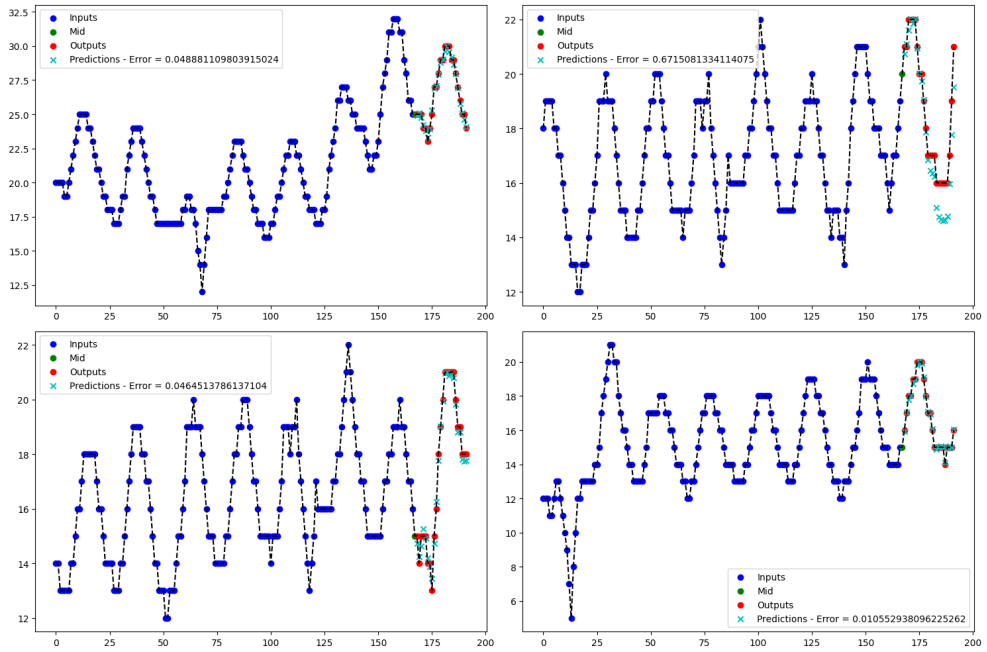
✓ 0.0s

```
Ground truth:  [[19. 18. 16. 16. 15. 15. 14. 14. 14. 14. 14. 14. 15. 15. 16. 17. 19. 21.
  22. 23. 25. 24. 23. 22.]]
Prediction:  [[19.120428  18.047054  16.726149  15.9983425 15.547313  15.193731
  14.681276  14.443553  14.415693  14.400859  14.44732   14.68005
  15.1154995 15.619518  16.211905  17.476038  19.255114  20.804506
  21.815401  22.879171  23.921862  23.35595   22.240602  21.478071 ]]
MAE:  0.9264912
```



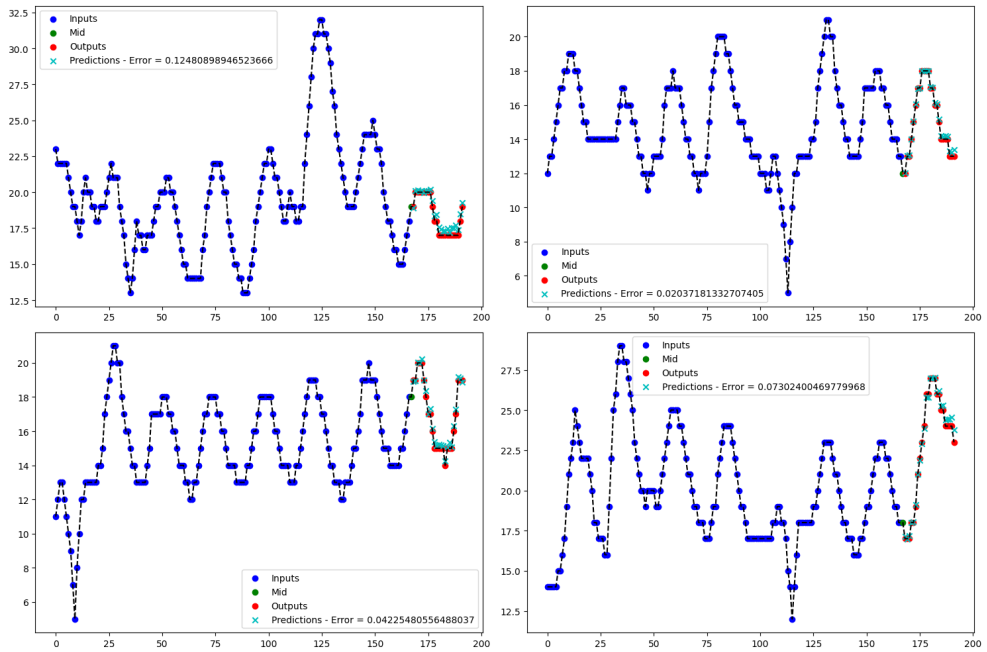
1. Experimentation by using larger hidden size of 256:

```
Epoch 1051/1051, Avg Loss: 0.011603372264653444
46min 26s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Ground truth:  [[19. 18. 18. 19. 20. 19. 19. 18. 18. 19. 19. 22. 24. 26. 28. 30. 31. 31.
  32. 32. 31. 31. 30. 29.]]
Prediction:  [[18.945644 18.042883 17.801981 18.900753 19.723366 18.98244  18.629864
  17.810043 17.773893 18.920929 18.953003 22.067928 23.873165 25.938566
  27.902607 30.15372  30.856146 30.897026 31.95671  31.707901 31.24931
  30.71125  29.947857 28.69932 ]]
MAE:  0.16728027
```



2. Experimentation by changing training loss function to Huber Loss Function and larger hidden size of 256:

```
Epoch 1051/1051, Avg Loss: 0.003943319036625326
45min 45s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Ground truth:  [[16. 16. 17. 17. 16. 16. 16. 16. 16. 16. 16. 17. 17. 18. 19. 20. 20. 19.
  19. 19. 18. 17. 17. 16.]]
Prediction:  [[15.981567  16.050251  16.912983  16.890858  15.732246  15.9609375
  15.770254  16.038769  15.9826    16.052261  16.158037  17.115349
  17.070961  18.18254   19.194172  20.068985  19.824501  18.910728
  19.167383  18.961823  18.080389  17.234608  17.415442  16.523579 ]]
MAE:  0.2261855
```



3. Experimentation by using the learning rate at 1e-4, and larger hidden size of 256:

```
Epoch 1051/1051, Avg Loss: 0.10661551505327224
42min 59s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Ground truth:  [[17. 17. 16. 15. 15. 15. 15. 15. 14. 15. 15. 15. 16. 17. 19. 19. 19. 18.
  19. 19. 20. 18. 17. 16.]]
Prediction:  [[17.526815  16.489962  15.63935   15.23418   15.108802  15.140106
  15.1092415 14.961542  14.855181  14.812046  14.891665  15.321793
  16.210655  17.37492   18.414268  18.900879  18.874632  18.843437
  19.055073  19.303892  19.20036   18.479982  17.352467  16.381094 ]]
MAE:  0.26470634
```