Final Answer to Your Question

Yes, the problem statement exactly matches your data and objective.
Using `Δ Time = (target_time - time)` is the correct and best approach for training.
You should improve normalization, loss weighting, optimizer, and architecture tuning to reduce RMSE significantly.

If you want, I can:

- Rewrite the model architecture with transformer encoder + LayerNorm
- Rewrite training loop with AdamW + scheduler + gradient clipping
- Give a custom WeightedLoss formula

Do you want me to **rewrite the model code in optimized form**?

explain the proposed solution clearly and also help me to visualize what type of data it is

Below is a **clear, structured explanation** of the proposed solution and a **conceptual visualization of what type of data you are working with**, without coding.

---

1. Understanding Your Data Type

Your dataset is **time-series trajectory data from a radar tracking system**.
Each row represents a **snapshot in time** of the target's state:

Input / Features

Feature	Meaning
TimeSec	Current timestamp of the radar sample
Pos_X, Pos_Y, Pos_Z	3D position coordinates of the target at that instant
Vel_X, Vel_Y, Vel_Z	Velocity components in 3D space
Range	Distance from radar to target

These values evolve over time because the target is **moving**, so the dataset is **sequential time-series**.

Outputs (Targets to Predict)

Target	Meaning
PIP_X1, PIP_Y1, PIP_Z1	3D location of interception point 1
…	…
PIP_X4, PIP_Y4, PIP_Z4	3D location of interception point 4
ΔT1, ΔT2, ΔT3, ΔT4	Time-to-go until interception (target_time – current time)

So the model learns:

- **Given current motion, predict future predicted interception positions and how long it takes to reach each one.**

---

2. Visualizing the Data (Conceptually)

Time-series view

```cpp
t1: [pos, vel, range] → future PIP1, ΔT1
t2: [pos, vel, range] → future PIP2, ΔT2
t3: [pos, vel, range] → future PIP3, ΔT3
```

Sliding Window View (LSTM Input)

```css
[ t1, t2, t3, ... t10 ]  -> Predict [PIP_x1 ... PIP_z4, ΔT1...ΔT4]
```

This is exactly how the LSTM learns temporal patterns.

Trajectory Visualization

Think of the target moving like this:

```pgsql
(Z axis altitude)
     |
40k -|       * target future path
     |     *
     |   *
     | *
     |* current position
     ------------------------ X (horizontal lateral motion)
       \
        \
         Y (depth motion)
```

Your PIP outputs represent predicted future positions along that trajectory.

---

3. Interpretation of the Correlation Matrix

Key findings:

- **PIP_X outputs strongly correlate with Pos_X and Vel_X**
- **PIP_Y outputs strongly correlate with Pos_Y and Vel_Y**
- **PIP_Z outputs correlate with Pos_Z, Vel_Z, Range**
- **ΔT correlates positively with Range and negatively with speed**
  (Farther distance → more time to go, faster velocity → less time to go)

This means your data has **strong physical structure**, and the model should capture kinematics.

---

4. Explanation of Proposed Improvements

1) Normalizing output per-column (very important)

Right now, all 12 outputs are scaled together → high magnitude values (Z or ΔT) dominate the loss.

Instead scale **each output independently**, so the model learns equally well on all dimensions.

2) Use Smooth L1 loss instead of MSE

MSE is sensitive to spikes and amplifies noise.
Trajectory prediction is noisy naturally, so SmoothL1 stabilizes learning.

3) Use AdamW optimizer

Better generalization + built-in regularization → reduces overfitting and RMSE.

4) Apply Learning-rate Scheduling

Lets model converge smoother rather than overjump minima.

5) Improve feature representation

Include features like **acceleration** or **radial velocity**.

6) Add LayerNorm & Positional Encoding

This stabilizes attention layers.

---

5. Why ΔT = (target_time – time) is a good choice

Raw Time	Δ Time
Absolute time in seconds	Time relative to interception
Very large values	Small compact values
Harder to learn	Easier to learn
No correlation with motion	Strong relationship with speed + range

So your change significantly improves learnability.

---

6. Summary in One Paragraph

You are working with **multivariate multistep time-series trajectory data** used for **radar interception prediction**. Inputs contain state information of a moving target (position, velocity, range) over time, and the output predicts future interception coordinates and time-to-go. The solution proposal involves improving model training stability and accuracy by using per-output normalization, a better optimizer (AdamW), SmoothL1 loss, learning-rate scheduling, and model architectural enhancements (LayerNorm & positional encoding). These improvements address imbalance among target dimensions and stabilise gradients, reducing error and improving prediction accuracy.

Would you like next?

I can provide:

1. A clean visual conceptual diagram of your data and pipeline
2. A rewritten improved training architecture (Transformer + LayerNorm)
3. A script to visualize trajectories in 3D using matplotlib

Which one would you like next?

- A) Improved model code
- B) Visualization 3D plot example
- C) Full report-style explanation for thesis / research paper
- D) All of the above

Tell me A/B/C/D.

i dont want to use the the transformers be stick to lstm, i need to improve the code with LSTM mechism

Great, we will stay fully in the LSTM world and make the **best possible LSTM pipeline** from what you already have.

I will do three things:

1. Explain clearly what your data looks like and how the columns relate (using the correlation matrix idea).
2. Explain the **LSTM-only** improvements conceptually.
3. Give you **concrete code changes** you can paste into your script.

---

1. What type of data is this?

You are working with **multivariate time-series kinematic data** from a radar tracking scenario.

For each target run, you have a sequence of rows:

- **Inputs (current state at time t):**
  - `TimeSec` – current time step.
  - `Pos_X, Pos_Y, Pos_Z` – 3D position.
  - `Vel_X, Vel_Y, Vel_Z` – 3D velocity.

  - `Range` – distance from radar to the object.
- **Outputs (what you want to predict):**
  - `PIP_X1..PIP_X4` – future X positions at 4 predicted interception points.
  - `PIP_Y1..PIP_Y4` – future Y positions.
  - `PIP_Z1..PIP_Z4` – future Z positions.
  - `TGT_TIME1..TGT_TIME4` – interception times.
    You have wisely converted these to
    `ΔT_i = TGT_TIME_i - TimeSec` = **time-to-go**, which makes the learning easier.

How the columns relate (from the correlation matrix)

From your heatmap, even though we do not read exact numbers, the structure is clear:

- **PIP_X1–PIP_X4**
  - Strong positive correlation with `Pos_X`.
  - Strong positive correlation with `Vel_X`.
  - Intuition: if current X position or X velocity is larger, future interception X will also be larger.
- **PIP_Y1–PIP_Y4**
  - Strong correlation with `Pos_Y`, `Vel_Y`.
  - Intuition: lateral motion along Y drives where the interception points are in Y.
- **PIP_Z1–PIP_Z4**
  - Correlate with `Pos_Z`, `Vel_Z`, and somewhat with `Range`.
  - Intuition: altitude and radial distance dominate vertical interception positions.
- **ΔT (time-to-go)**
  - Positively correlated with `Range` (larger distance → more time needed).
  - Negatively correlated with overall speed (larger speed → less time to go).

So physically, your data is **very well-structured**: outputs are smooth functions of the current kinematic state.

---

2. LSTM-only improvement strategy

We keep the core idea: **sliding window → LSTM → predict 12 outputs (PIP & ΔT).**

The main problems in your current code:

1. **Y normalization is global**, not per output dimension. Big-magnitude outputs dominate the loss.
2. **Training loss per epoch is computed incorrectly** (loss_list never reset).
3. LSTM stack + attention is quite heavy and may overfit / be unstable.

4. No gradient clipping; optimizer is plain Adam without weight decay.
5. You computed `x_mean`, `x_std` but then switched to MinMax; this is OK but can be cleaner.

High-impact changes without transformers

1. **Normalize each output column separately** (very important).
2. **Fix the loss bookkeeping bug** so each epoch sees its own mean loss.
3. **Use AdamW + weight decay + LR scheduler**.
4. **Add gradient clipping** to stabilise training.
5. **Slightly simplify the LSTM stack + add LayerNorm + Dropout** but still LSTM+residual.
6. Keep your `ΔT = TGT_TIME - TimeSec` (this is good).

---

3. Concrete code changes

3.1. Better normalization of X and Y

Replace your current X/Y scaling block with this:

```python
# --- build X_data, Y_data as you already do above ---

X_data = np.concatenate(X_data, axis=0)  # (N, seq_len, input_size)
Y_data = np.concatenate(Y_data, axis=0)  # (N, num_outputs=12)

# ---------- INPUT NORMALIZATION (per feature) ----------
X_flat = X_data.reshape(-1, input_size)  # (N*seq_len, 7)
x_mean = X_flat.mean(axis=0)
x_std  = X_flat.std(axis=0) + 1e-6

X_data_norm = (X_data - x_mean) / x_std

# ---------- OUTPUT NORMALIZATION (per output dimension) ----------
y_mean = Y_data.mean(axis=0)        # shape (12,)
y_std  = Y_data.std(axis=0) + 1e-6  # shape (12,)

Y_data_norm = (Y_data - y_mean) / y_std

print("X_mean:", x_mean)
print("X_std :", x_std)
print("Y_mean:", y_mean)
print("Y_std :", y_std)

from sklearn.model_selection import train_test_split
X_train, X_eval, y_train, y_eval = train_test_split(
    X_data_norm, Y_data_norm, test_size=0.2, random_state=42, shuffle=True
)

print(X_train.shape, X_eval.shape, y_train.shape, y_eval.shape)

# reshape already in (N, seq_len, input_size), so just convert to tensors
featuresTrain = torch.from_numpy(X_train).float()
featuresEval  = torch.from_numpy(X_eval).float()
targetsTrain  = torch.from_numpy(y_train).float()
targetsEval   = torch.from_numpy(y_eval).float()
```

And when you save the checkpoint, save `y_mean` and `y_std` as vectors, not scalars.

3.2. Cleaner LSTM model with residual + attention

Here is a refined version of your LSTM (still LSTM, no transformers):

python

```python
class LSTMResidualAttn(nn.Module):
    def __init__(self, num_classes, input_size, hidden_size, num_layers,
seq_length):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # Single multi-layer LSTM instead of hand-stacking
        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            bidirectional=True,
            dropout=0.2 if num_layers > 1 else 0.0,
        )

        self.attn = nn.MultiheadAttention(embed_dim=2*hidden_size,
num_heads=2,
                                          batch_first=True)
        # Project original input for residual connection
        self.input_proj = nn.Linear(input_size, 2*hidden_size)

        self.norm1 = nn.LayerNorm(2*hidden_size)
        self.dropout = nn.Dropout(0.2)

        self.mlp = nn.Sequential(
            nn.Linear(2*hidden_size, 64),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, num_classes),
        )

    def forward(self, x):
        # x: (B, T, input_size)
        lstm_out, _ = self.lstm(x)             # (B, T, 2*hidden)
        attn_out, _ = self.attn(lstm_out, lstm_out, lstm_out)  # (B, T, 2*hidden)

        # Residual + LayerNorm
        res = self.input_proj(x)               # (B, T, 2*hidden)
        out = self.norm1(attn_out + res)
        out = self.dropout(out)

        # Use last time step
        last = out[:, -1, :]                   # (B, 2*hidden)
        logits = self.mlp(last)                # (B, num_classes)
        return logits
```

Use it like:

```python
input_size = 7
hidden_size = 64
num_layers = 2
num_classes = 12

lstm = LSTMResidualAttn(num_classes, input_size, hidden_size, num_layers, seq_length=seq_length)
```

This is still pure LSTM + attention, but:

- Simpler, fewer places to explode.
- LayerNorm stabilizes residual + attention.
- Dropout reduces overfitting.

3.3. Improved optimizer, loss, gradient clipping, and training loop

Loss and optimizer

```python
device = "cuda" if torch.cuda.is_available() else "cpu"
lstm = lstm.to(device)

criterion = nn.SmoothL1Loss()  # better for noisy regression than plain MSE

optimizer = torch.optim.AdamW(lstm.parameters(), lr=1e-3, weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', patience=8, factor=0.5, verbose=True
)
```

DataLoaders (same as you but keep)

```python
batch_size = 64
train_dataset = torch.utils.data.TensorDataset(featuresTrain, targetsTrain)
eval_dataset  = torch.utils.data.TensorDataset(featuresEval, targetsEval)

train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)
val_loader    = torch.utils.data.DataLoader(eval_dataset,
batch_size=batch_size, shuffle=False)
```

Fixed training loop (note where I reset `loss_list`)

```python
num_epochs = 150
best_val_loss = float('inf')
best_epoch = 0

for epoch in range(num_epochs):
    # ----- TRAIN -----
    lstm.train()
    train_losses = []   # reset every epoch

    for features, labels in tqdm(train_loader, leave=False):
        features = features.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        outputs = lstm(features)

        loss = criterion(outputs, labels)
        loss.backward()

        # gradient clipping
        torch.nn.utils.clip_grad_norm_(lstm.parameters(), max_norm=1.0)

        optimizer.step()
        train_losses.append(loss.item())

    train_loss = float(np.mean(train_losses))

    # ----- VALIDATION -----
    lstm.eval()
    val_losses = []
    with torch.no_grad():
        for features, labels in val_loader:
            features = features.to(device)
            labels = labels.to(device)

            outputs = lstm(features)
            val_loss = criterion(outputs, labels)
            val_losses.append(val_loss.item())

    val_loss = float(np.mean(val_losses))
    scheduler.step(val_loss)

    print(f"Epoch {epoch+1}/{num_epochs} - train_loss: {train_loss:.6f} -
val_loss: {val_loss:.6f}")

    # ----- SAVE BEST MODEL -----
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_epoch = epoch
        torch.save(
            {
                'state_dict': lstm.state_dict(),
                'model': lstm,
                'train_loss': train_loss,
                'val_loss': val_loss,
                'epoch': epoch + 1,
                'input_means': x_mean,
                'input_stds': x_std,
                'output_means': y_mean,
                'output_stds': y_std,
            },
            f"exp_nov/run{file_index}/best_lstm.pt"
        )
        print(f"  -> New best model saved at epoch {epoch+1} with val_loss
{val_loss:.6f}")
```

Key fixes vs your original:

- `train_losses` reset each epoch (no accumulation across epochs).
- AdamW + weight decay instead of plain Adam.
- SmoothL1 loss instead of scaled MSE.
- Gradient clipping to prevent spikes.
- Scheduler on validation loss.

---

4. How this should help your RMSE

These changes address the main pain points you see in your plots:

- **Per-output scaling** makes PIP_Z and ΔT learnable instead of being dominated by one axis.
- **Stabilised LSTM stack** + clipping reduces the oscillations you saw in Z1, Z2 predictions.
- **Better optimizer and scheduler** should give a lower, smoother validation curve.

You should see:

- Validation loss decreasing more steadily.
- Much closer tracking in most PIP plots.
- Lower per-output RMSE, especially on the noisy axes.

If you want next, I can:

- Show you exactly how to **denormalize predictions** (using `y_mean`, `y_std` as vectors) and compute RMSE per each of the 12 outputs.
- Or help design **diagnostic plots** to see which dimension is still bad and tune its weight in a custom loss.

earlier you told me that that my z-axis contains noisy tell me how are you fixing it and what about the positional encoding did you apply it?

Good questions. Let me connect the dots clearly.

---

1. Why I said your Z-axis is "noisy"

From your plots:

- PIP_Z predictions wiggle a lot and often diverge from ground truth.
- RMSE for Z components is much larger than for X/Y.
- Z is also strongly coupled with **Range**, so small errors in range/altitude blow up the Z coordinate.

This comes from a combination of:

1. **Scale imbalance**
   Z and Range are ~40k, while some other outputs are much smaller. When you used a **single global** `y_mean` **/** `y_std`, the loss was dominated by the large-magnitude dimensions (especially Z).
2. **Dynamic behaviour**
   The altitude profile is steeper / more curved than X,Y. So the model needs more stable gradients and better regularisation to fit it.
3. **Training instability**
   Multiple LSTM layers + attention + residuals without norm/clipping can easily produce spikes or oscillations, which appear as "noisy" predictions on the hardest dimensions (like Z).

---

2. How the proposed changes specifically help the Z-axis

2.1 Per-output normalization of Y

Previously:

```python
y_mean = np.mean(Y_data)
y_std  = np.std(Y_data)
Y_data = (Y_data - y_mean) / y_std
```

Now (what I recommended):

```python
y_mean = Y_data.mean(axis=0)        # shape (12,)
y_std  = Y_data.std(axis=0) + 1e-6  # shape (12,)
Y_data = (Y_data - y_mean) / y_std
```

Effect on Z:

- Z outputs (big values) get **their own** mean/std.
- After scaling, all 12 outputs live roughly in the same range (≈ −1 to 1).
- The loss sees Z, X, Y, ΔT as **equally important**, instead of Z dominating or being under-fitted.

This alone usually cuts a big chunk off Z-RMSE.

2.2 SmoothL1 loss instead of plain MSE

```python
criterion = nn.SmoothL1Loss()
```

- MSE heavily punishes occasional big errors; with a noisy Z signal that leads to unstable gradients.
- SmoothL1 behaves like L1 for large errors → more **robust to outliers**.
- For Z, where a few bad points can be very far off, this stabilises learning and avoids the wild swings you saw.

2.3 Gradient clipping

```python
loss.backward()
torch.nn.utils.clip_grad_norm_(lstm.parameters(), max_norm=1.0)
```

```
optimizer.step()
```

- If Z is hard to fit, gradients from its error can explode.
- Clipping keeps all parameter updates bounded → prevents the network from making huge jumps that create oscillating or jagged Z predictions.

2.4 LayerNorm + simpler residual/attn structure

In the revised LSTM model I suggested:

```python
 self.norm1 = nn.LayerNorm(2*hidden_size)
...
out = self.norm1(attn_out + res)
```

- LayerNorm keeps the **scale of hidden activations stable** across time + across features.
- Because Z is more sensitive to small changes, stabilizing the hidden state helps produce smoother, more consistent Z outputs.
- Simplifying the stack (single multi-layer LSTM instead of repeated manual LSTMs) also reduces overfitting to noise.

---

3. What about positional encoding?

Short answer:

No, I did not add explicit transformer-style positional encoding.
And that is deliberate, because:

1. **LSTMs already encode order**
   In an LSTM, each step's hidden state depends on all previous steps. Temporal order is built into the recurrence, unlike transformers which need extra positional encodings.
2. **You already have "time" as a feature**
   Your input includes `TimeSec` (or an equivalent time index in each file). That acts as a **continuous positional signal**: the model knows if it is early or late in a trajectory.

So the combination of:

- LSTM recurrence, and
- Explicit time feature

is usually enough to encode position in sequence for this kind of problem.

4. If you still want extra “positional” signal (still LSTM-only)

You can add a very simple positional feature without using transformers:

```python
# Before building X_data
seq_len = 10
pos_encoding = np.linspace(0, 1, seq_len)  # shape (10,)

# When you build each sliding window x (shape: seq_len x 7)
# you can concatenate position as an 8th feature:
x_window = sliding_windows(features.values, seq_length, stride)
# x_window: (num_windows, seq_len, 7)
pos_feat = pos_encoding[None, :, None]     # (1, seq_len, 1)
x_window = np.concatenate([x_window, np.repeat(pos_feat, x_window.shape[0], axis=0)], axis=2)
# -> input_size becomes 8
```

This is a simple “relative step index” ∈ [0,1] per time step.
It is not a transformer positional encoding, but it plays a similar role while keeping everything LSTM.

---

5. Summary

- Z-axis looked noisy mainly because of **scale imbalance, harder dynamics, and training instability**.
- We addressed that by:
  - Per-output normalization (`y_mean` , `y_std` per column),
  - Robust loss (SmoothL1),
  - Gradient clipping,
  - LayerNorm + a cleaner LSTM+attention architecture.
- I did **not** add transformer-style positional encoding; your LSTM + time feature already handle order.
  If you want an extra positional signal, you can add a simple normalized step index as an additional input feature.

If you like, next I can help you:

- design a **per-output RMSE report** (for X/Y/Z and each PIP) so you can see exactly how much Z improved after these changes.