

Project 2A

Team 4

曹心瞳, 黃旭鵬, 林詩柔, 朴俊鎬, 邱紋宸

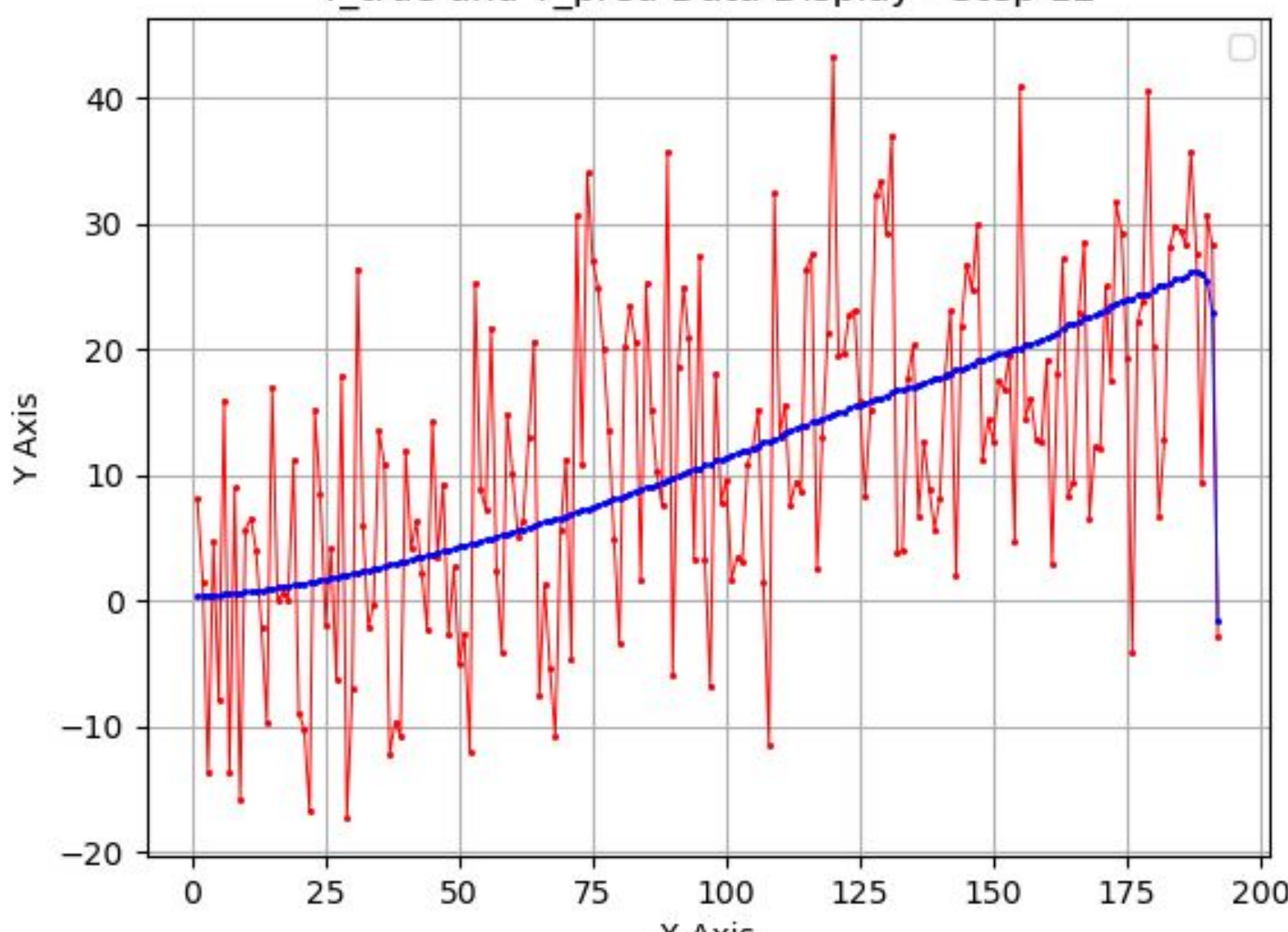
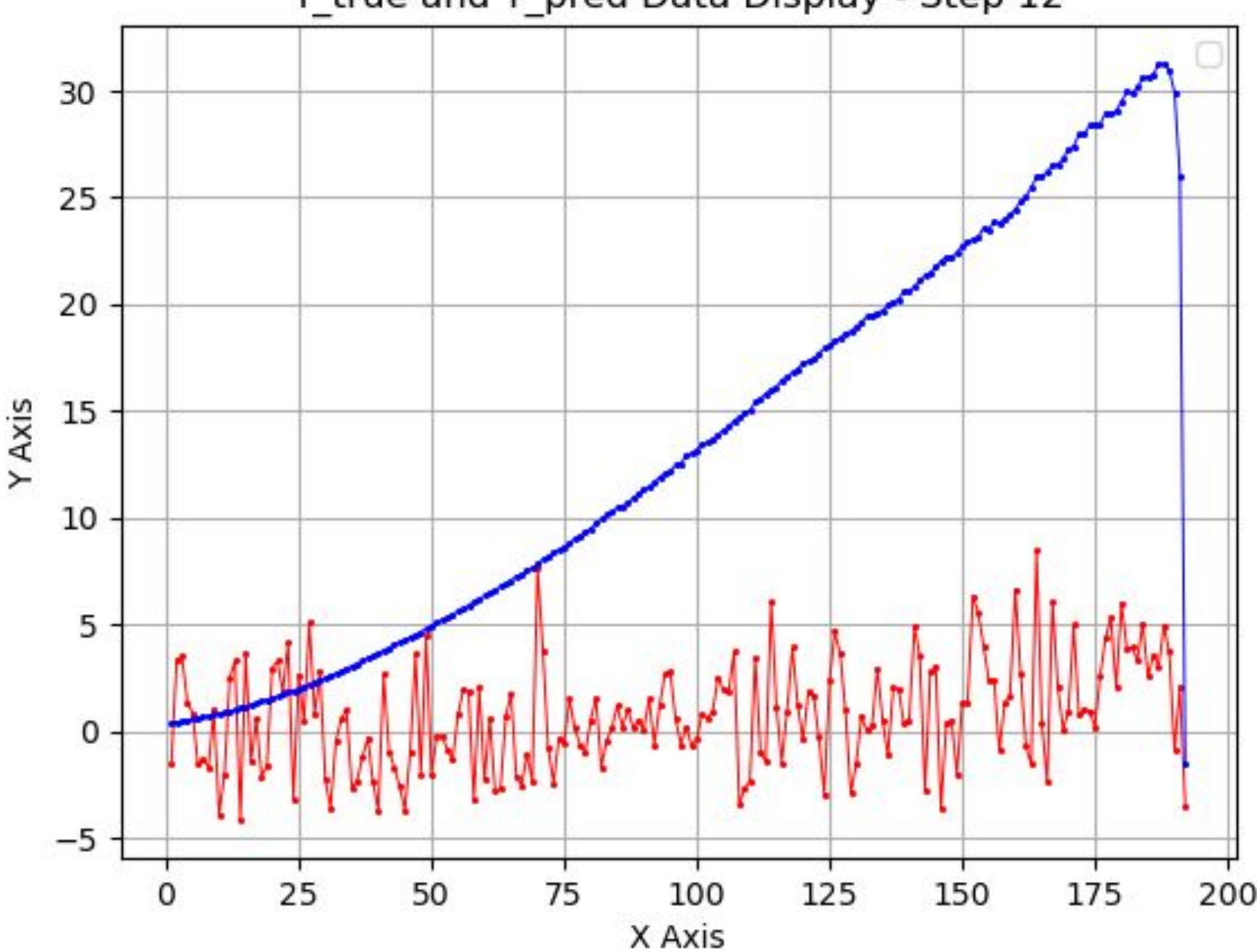
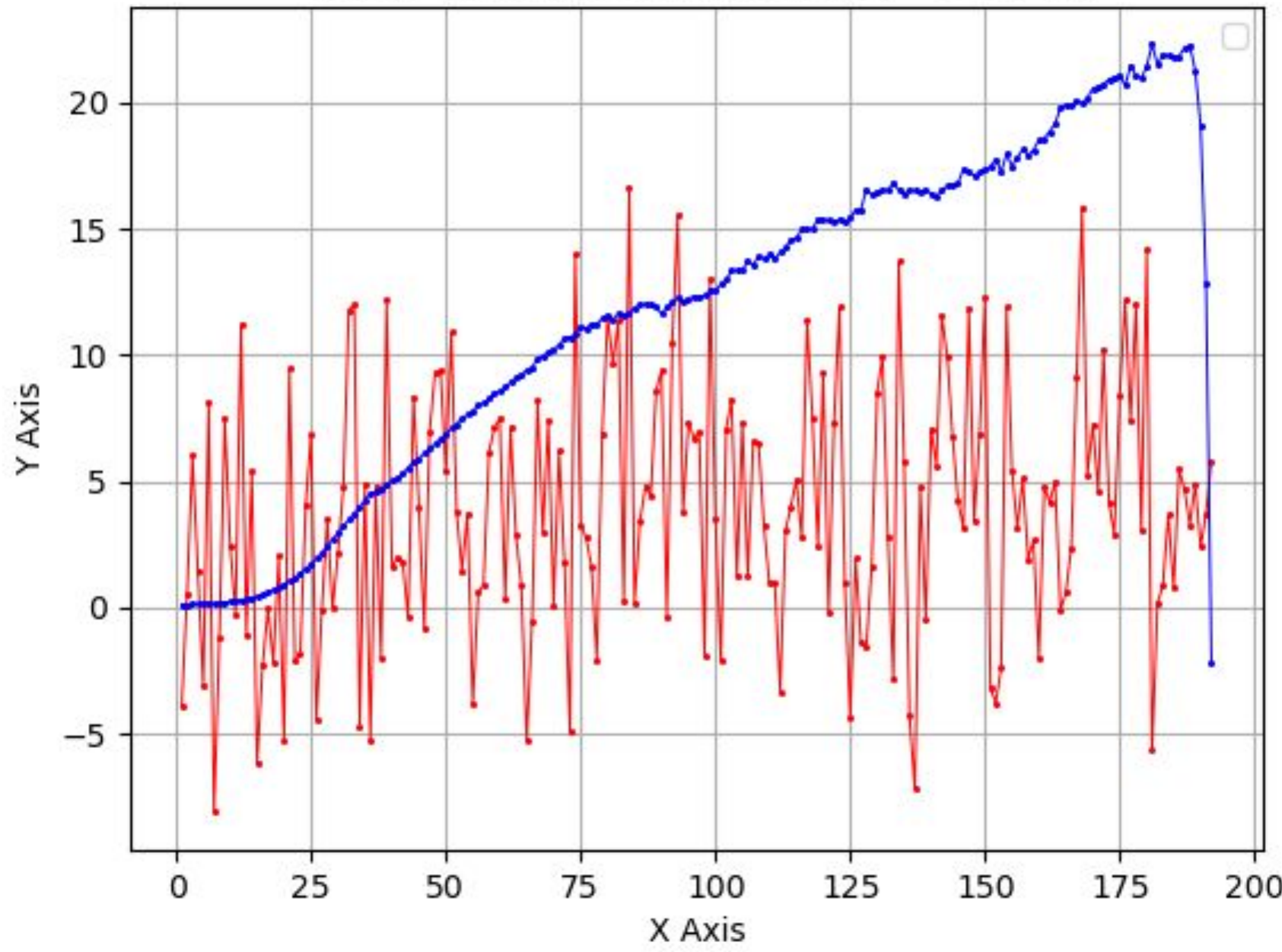
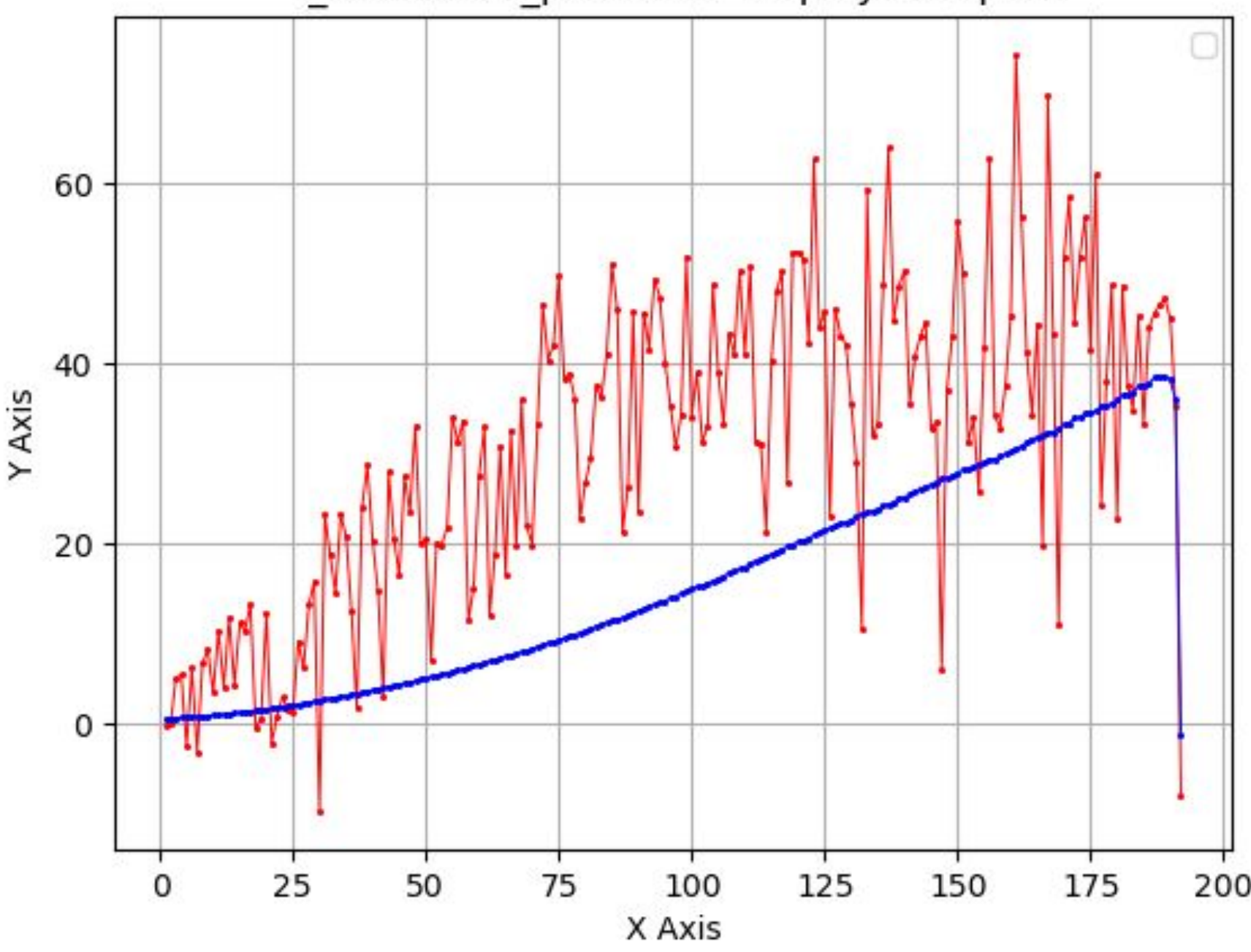
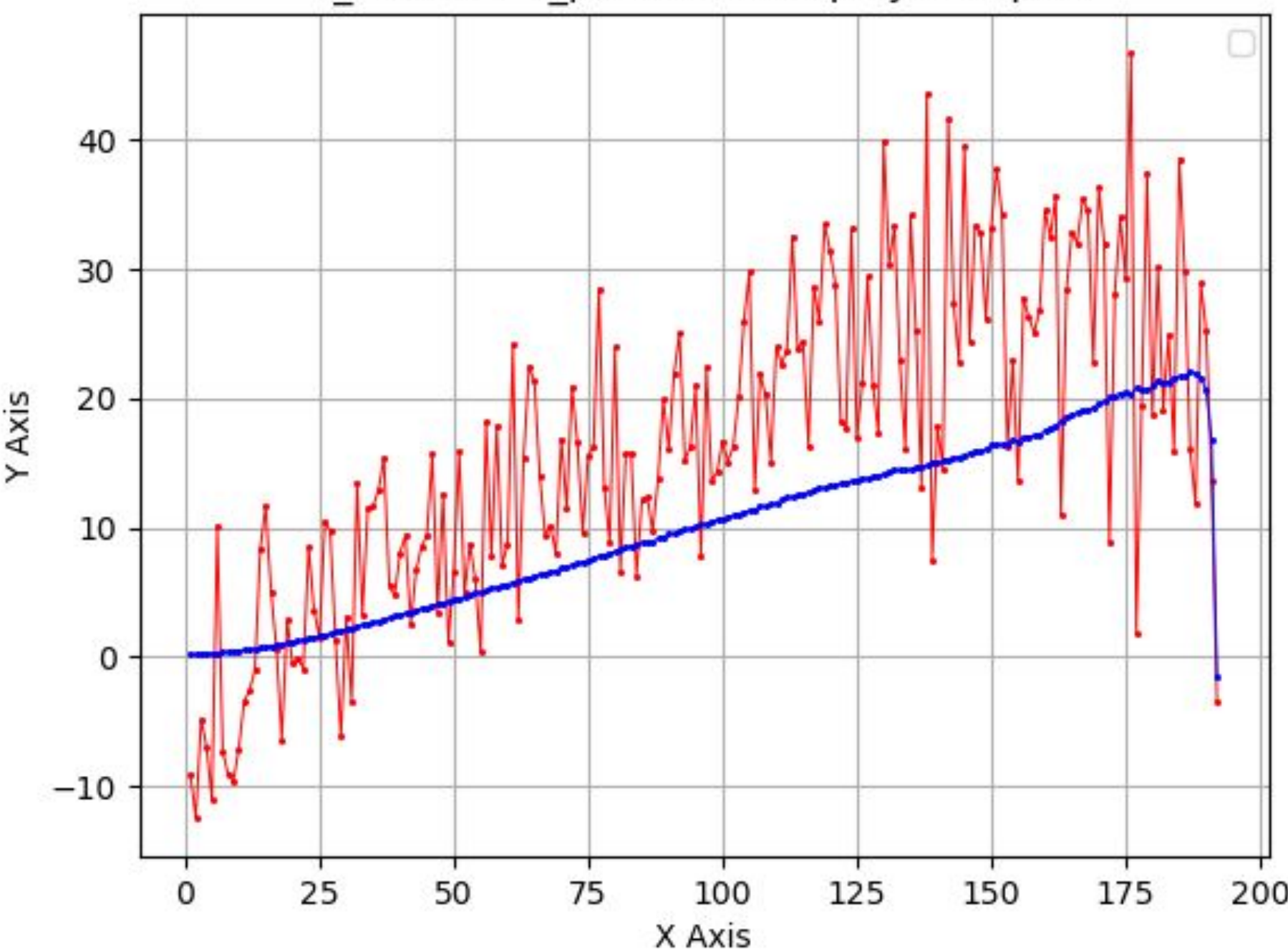
Abstract

We have examined the content of project 2A, focusing on optimizing the custom_loss function. We experimented with various approaches, including MSE, quantile loss, Huber loss, and smooth L1 loss. We trained different models with different custom_loss, then compared their performances. Upon observation, the combination of MSE loss and Huber loss with 50% weights for each yields the best results.

Loss Function We Used

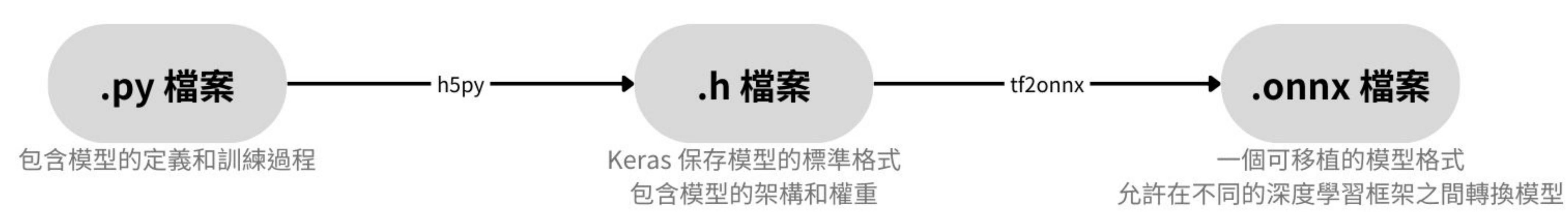
Loss function	Equation	Feature/Advantage
Mean Squared Error	$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$	Handle outliers efficiently.
Quantile Loss	$Quantile\ Loss = \sum_{i=y_i < \hat{y}_i} (\gamma - 1) y_i - \hat{y}_i + \sum_{i=y_i \geq \hat{y}_i} (\gamma) y_i - \hat{y}_i $	Used to forecast quantiles, which is the value that indicates how many values in group fall below or above the threshold.
Huber Loss	$L_{\delta} = \begin{cases} \frac{1}{2}(\gamma - \hat{\gamma})^2, & \text{if } (\gamma - \hat{\gamma}) < \delta \\ \delta \left((\gamma - \hat{\gamma}) - \frac{1}{2} \delta \right), & \text{otherwise} \end{cases}$	Combination of the quadratic and linear scoring algorithms, which has MSE and MAE’s advantages.
Smooth L1 Loss	$SmoothL1(\gamma, beta) = \begin{cases} \frac{1}{2}(\gamma - \hat{\gamma})^2 / beta, & \text{if } \gamma - \hat{\gamma} < beta \\ \gamma - \hat{\gamma} - \frac{1}{2} * beta, & \text{otherwise} \end{cases}$	<p>Closely related to Huber loss, being equivalent to huber(y)/beta. when beta->0, Smooth L1 loss converges to L1 loss.L2 Loss is used near zero, and L1 Loss is used elsewhere. This loss function can reduce the impact of outliers while maintaining smoothness to the predicted values.</p> <p>Advantage:</p> <ul style="list-style-type: none">Compared with the L1 loss function(MAE), it can converge faster.Compared with the L2 loss function(MSE), it is not sensitive to outliers and outliers, the gradient changes are relatively smaller, and it is not easy to run away during training.
log cosh loss	$Logcosh(t) = \sum_{i=1}^N \log(cosh(\hat{y}_i - y_i))$	Very close to the huber loss in behavior, but don’t have to decide the threshold as in Huber loss

individual tests of four different loss functions

loss function	def custom_loss()	Ypred (red) vs Ytrue (blue) at step 12	conclusion
Mean Squared Error	<div><div>test1</div><div>def custom_loss(y_true, y_pred): loss = tf.keras.losses.mse(y_true, y_pred) # for whole return loss</div><div>test2</div><div>def custom_loss(y_true, y_pred): true_valid_radar=y_true[:,384] pred_valid_radar=y_pred[:,384] true_valid_l=y_true[:,1] pred_valid_l=y_pred[:,1] mse_valid_radar=tf.keras.losses.mse(true_valid_radar, pred_valid_radar) mse_valid_l=tf.keras.losses.mse(true_valid_l,pred_vali d_l) loss = tf.keras.losses.mse(y_true, y_pred) # for whole 2383 vector loss=loss+0.2*mse_valid_radar+0.8*mse_valid_l #loss=0.8*loss+0.2*mse_valid_radar+0.2*mse_valid_l return loss</div></div>	<div>test1 plot</div>  <div>test2 plot</div> 	<div>test1</div> <div>The predicted value appears to be more preferable, but there is significant fluctuation both above and below the expected range.</div> <div>test2</div> <div>when we calculate mse loss of y_true[:,384] and y_pred[:,384] and add it to original loss value proportionally.The predicted result seems to be worse!</div>
Quantile Loss	<div>def custom_loss(y_true, y_pred): losses = [] # Define the quantiles we'd like to estimate quantiles = [.1, .5, .9] for i, quantile in enumerate(quantiles): error=y_true-y_pred loss_tp = tf.reduce_mean(tf.maximum(quantile*error, (quantile-1)*error), axis=-1) losses.append(loss_tp) combined_loss = tf.reduce_mean(tf.add_n(losses)) return combined_loss</div>		By using quartile loss,we can't get nice predicted value when x is bigger than 100
Huber Loss	<div>def custom_loss(y_true, y_pred): delta = 1 huber_loss = tf.keras.losses.huber(y_true, y_pred, delta) # for whole 2383 vector return huber_loss</div>		By using huber loss, The predicted trend does not align with actual values, and there is a significant deviation of 60 when x is between 150 and 175.
Smooth L1 Loss	<div>def custom_loss(y_true, y_pred): diff = tf.abs(y_true - y_pred) smooth_l1_loss = tf.where(diff < 1, 0.5 * diff ** 2, diff - 0.5) return tf.reduce_mean(smooth_l1_loss, axis=-1)</div>		By using smooth L1 Loss,the predicted value appears to be more preferable.The predicted trend closely follows actual values, with minimal fluctuations both above and below.

combination tests of two different loss functions

loss function	def custom_loss()	Ypred (red) vs Ytrue (blue) at step 12	conclusion
Smooth L1 Loss+quantile	<pre>def custom_loss(y_true, y_pred): losses = [] # Define the quantiles we'd like to estimate quantiles = [.1, .5, .9] for i, quantile in enumerate(quantiles): error=y_true-y_pred loss_tp = tf.reduce_mean(tf.maximum(quantile*error, (quantile-1)*error), axis=-1) losses.append(loss_tp) quantile_loss = tf.reduce_mean(tf.add_n(losses)) delta=1 diff = tf.abs(y_true - y_pred) smooth_l1_loss = tf.where(diff < delta, 0.5 * diff ** 2, delta * (diff - 0.5 * delta)) smooth_l1_loss = tf.reduce_mean(smooth_l1_loss, axis=-1) combined_loss = 0.5 * smooth_l1_loss + 0.5 * quantile_loss #combined_loss = 0.8* smooth_l1_loss + 0.2 * quantile_loss #combined_loss = 0.9 * smooth_l1_loss + 0.1 * quantile_loss return combined_loss</pre>		By using smooth L1 Loss and quantile loss ,the predicted result seems to be worse!
Smooth L1 Loss+huber	<pre>def custom_loss(y_true, y_pred): delta = 1 #delta=2 huber_loss = tf.keras.losses.huber(y_true, y_pred, delta) # for whole 2383 vector diff = tf.abs(y_true - y_pred) smooth_l1_loss = tf.where(diff < delta, 0.5 * diff ** 2, delta * (diff - 0.5 * delta)) smooth_l1_loss = tf.reduce_mean(smooth_l1_loss, axis=-1) combined_loss = 0.9 * smooth_l1_loss + 0.1 * huber_loss #combined_loss = 0.6 * smooth_l1_loss + 0.4 * huber_loss #combined_loss = 0.95 * smooth_l1_loss + 0.05 * huber_loss return combined_loss</pre>	<div>Step12</div> <div>Step22</div>	By using smooth L1 Loss and huber loss with 90% smooth loss and 10% huber loss we got out second best predicted result.
Smooth L1 Loss+MSE	<pre>def custom_loss(y_true, y_pred): mse_loss = tf.keras.losses.mse(y_true, y_pred) # for whole delta=1 diff = tf.abs(y_true - y_pred) smooth_l1_loss = tf.where(diff < delta, 0.5 * diff ** 2, delta * (diff - 0.5 * delta)) smooth_l1_loss = tf.reduce_mean(smooth_l1_loss, axis=-1) combined_loss = 0.5 * smooth_l1_loss + 0.5 * mse_loss #combined_loss = 0.6 * smooth_l1_loss + 0.4 * mse_loss #combined_loss = 0.7 * smooth_l1_loss + 0.3 * mse_loss #combined_loss = 0.8 * smooth_l1_loss + 0.2 * mse_loss #combined_loss = 0.2 * smooth_l1_loss + 0.8 * mse_loss return combined_loss</pre>		By using smooth L1 Loss and MSE loss ,the predicted result seems to be worse!
MSE+huberloss	<pre>def custom_loss(y_true, y_pred): delta = 1 huber_loss = tf.keras.losses.huber(y_true, y_pred, delta) # for whole 2383 vector mse_loss = tf.keras.losses.mse(y_true, y_pred) # for whole combined_loss = 0.4 * mse_loss + 0.6 * huber_loss #combined_loss = 0.5 * mse_loss + 0.5 * huber_loss #combined_loss = 0.1 * mse_loss + 0.9 * huber_loss #combined_loss = 0.9 * mse_loss + 0.1 * huber_loss #combined_loss = 0.3 * mse_loss + 0.7 * huber_loss #combined_loss = 0.7 * mse_loss + 0.3 * huber_loss return combined_loss</pre>	<div>40%mse+60%huber Step22</div> <div>50%mse+50%huber Step22</div>	By using MSE loss and huber loss,with 50% MSEloss and 50% huber loss we got our best predicted result.



Data Preparation and Custom Loss Function:
Defines functions to fetch and process data and a custom loss function using mean squared error (MSE).

- Main Training Process:**
- Parses command-line arguments.
 - Builds a neural network model using get_model from modelB6.
 - Sets up callbacks, optimizers, and compiles the model.
 - Fits the model using training data and validates using validation data.
 - Saves the trained model(.h5) and plots training/validation loss, RMSE, and MAE.

Training Time Calculation:
Calculates and prints the training time.

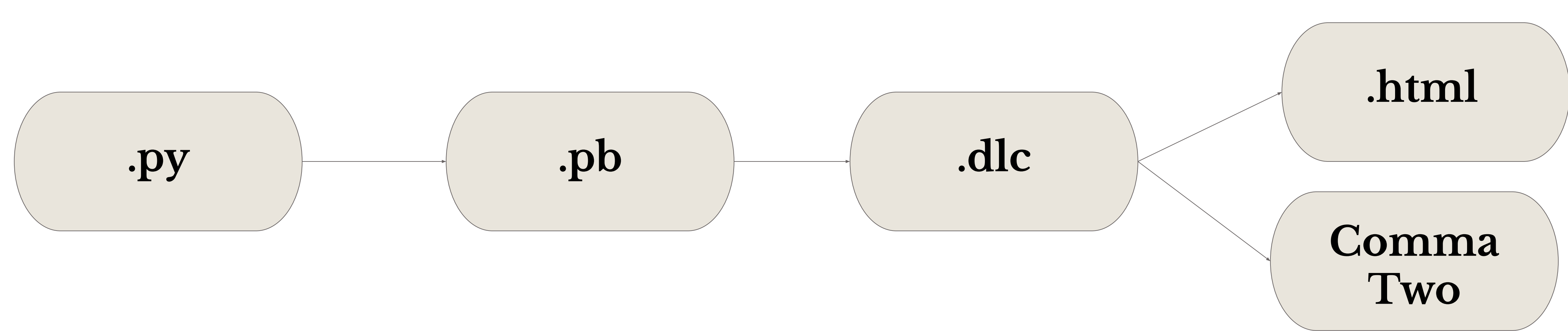
Plotting and Visualization:
Plots training/validation loss, RMSE, and MAE using Matplotlib.

Custom Functions:
Defines custom loss and RMSE functions to be used during the model conversion.

Loading Keras Model:
Loads a Keras model (modelB6.h5) that utilizes the defined custom loss and RMSE functions.

Conversion to ONNX and saving model :
Converts the loaded Keras model to ONNX format using tf2onnx's from_keras function. The conversion includes specifying an ONNX opset version and potential custom ops for handling TensorFlow operations that might not directly convert to ONNX. Then saves the converted ONNX model to a specified path.

Exception Handling:
Catches any exceptions that might occur during the conversion process and prints an error message if conversion fails.



What is dlc file:
dlc file stands for “Downloadable Content”, which can be deployed on to the target hardware. In this case, the hardware will be Comma Two

Conversion to dlc file:
Convert the pb file to dlc file using Snapdragon Neural Processing Engine (snpe) by Qualcomm developer network.

Conversion of dlc file to html:
Convert dlc file to html using snpe-dlc-viewer command

Conclusion

We experimented with four distinct loss functions: Mean Squared Error (MSE), Quantile Loss, Huber Loss, and Smooth L1 Loss. We conducted individual tests as well as combinations of two different loss functions to identify the optimal ypred value, ensuring that the predicted trend closely aligns with actual values while minimizing fluctuations above and below. After thorough evaluation, we determined that a combination of MSE loss and Huber loss, with a 50% weighting for each, yielded favorable results. Additionally, employing Smooth L1 Loss and Huber Loss with a 90% weighting for Smooth L1 Loss and 10% for Huber Loss proved to be the preferred loss function.

To make the model accessible and usable across different platforms, we navigated the process of transferring it from a Python script (.py) to a header file (.h) and then to the Open Neural Network Exchange format (.onnx). Subsequently, we further encapsulated Protocol Buffers (.pb) into a Deep Learning Container (.dlc) for deployment on specific platforms. Additionally, an interactive HTML file was generated using TensorFlow.js, allowing users to interact with and experience the model's predictions directly in a web browser. This multi-step process ensures the versatility and accessibility of our optimized machine learning model.