PMSM Observer Design with Transformers

Desmond Nii Ashitey **Hammond**
Dieter H. **Hoogestraat**
OpenCampus SS2022: Transformers for NLP

Agenda

- Problem: Controlling PMSMs
- Idea: Estimation is all you need
- Data: MATLAB-Simulink
- Method: Transformers Tensorflow
- Method: ViT Model
- Results
- Conclusion

Controlling PMSMs

- An electric motor consists of at least two units: the immobile stator (“housing”) und the moving part, the rotor.

- Both units can be designed as coils, what is pretty easy to control but energetically wasteful, because both units need energy.

- The other way is, to design one unit as a permanent magnet (e.g. Neodyme) giving a Permanent Magnetic Synchronous Motor (PMSM).

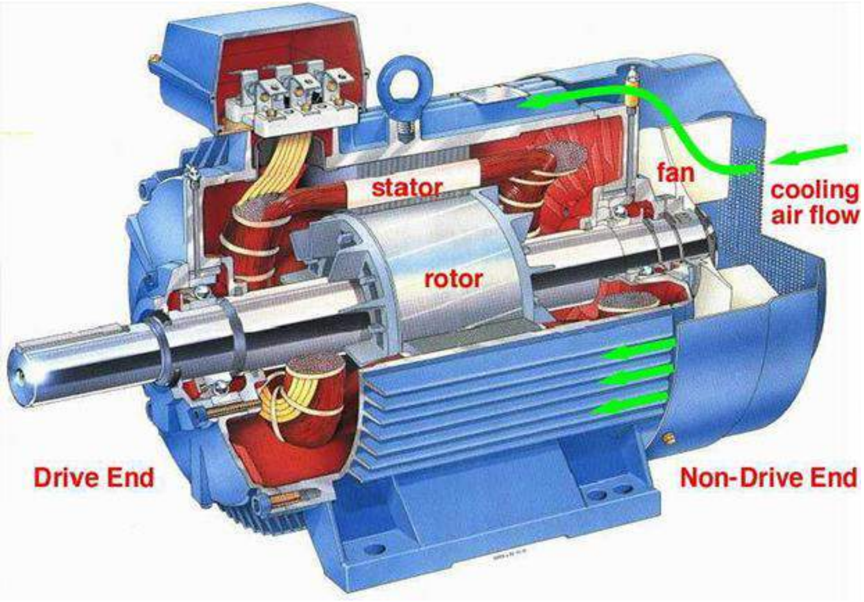


Fig. 1: A conventional electric motor (Source: D&F Liquidators)

Controlling PMSMs

This type of motor is used for a wide range of (increasingly sensitive) applications: e.g. electric cars, planes, surgical robots.

The problem:
Precise control of PMSMs requires information from costly sensors or :

Measurement	Price f. sensor
Current	$
Voltage	$
Rotation speed (rpm)	$$
Rotation angular position	$$
Torque	$$$$$

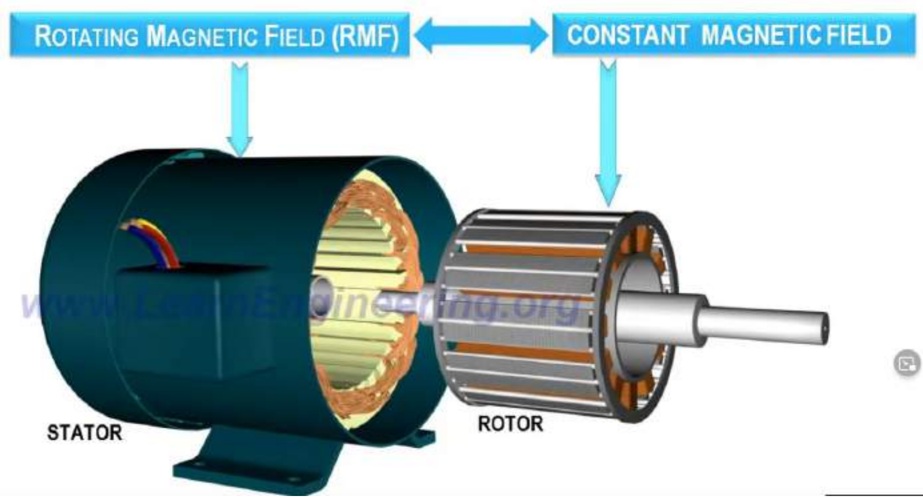


Fig. 2: A Permanent Magnet Synchronous Motor (PMSM) (Source: www.learnengineering.com)

Estimation is all you need

What usually happens (in a car):

pedal action ≙ asking for speed ≙ actually asking for torque => requires position ≙ expensive sensors

Ultimate Aim:

- Replace some sensors with a **Transformer-based Estimator(Observer)** in the control loop
- **Predict** the next value
- Use difference to control the **torque**

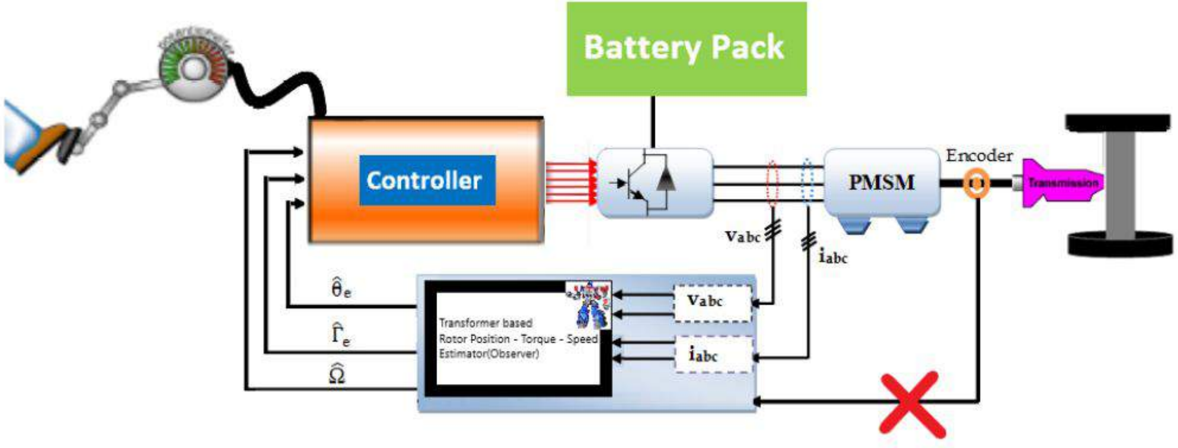


Fig. 3: PMSM control cycle

Dataset Generation – MATLAB/Simulink

- **Real data was not available. We used a Simulink-model developed**
- **PMSM was modeled and simulated using dynamic equations and parameters of Renault Zoe-motor.**
- **Each block contains a dynamic equation.**
- **Signals were sampled and exported as a multivariate timeseries CSV file.**
- **This delivers fairly noiseless data.**

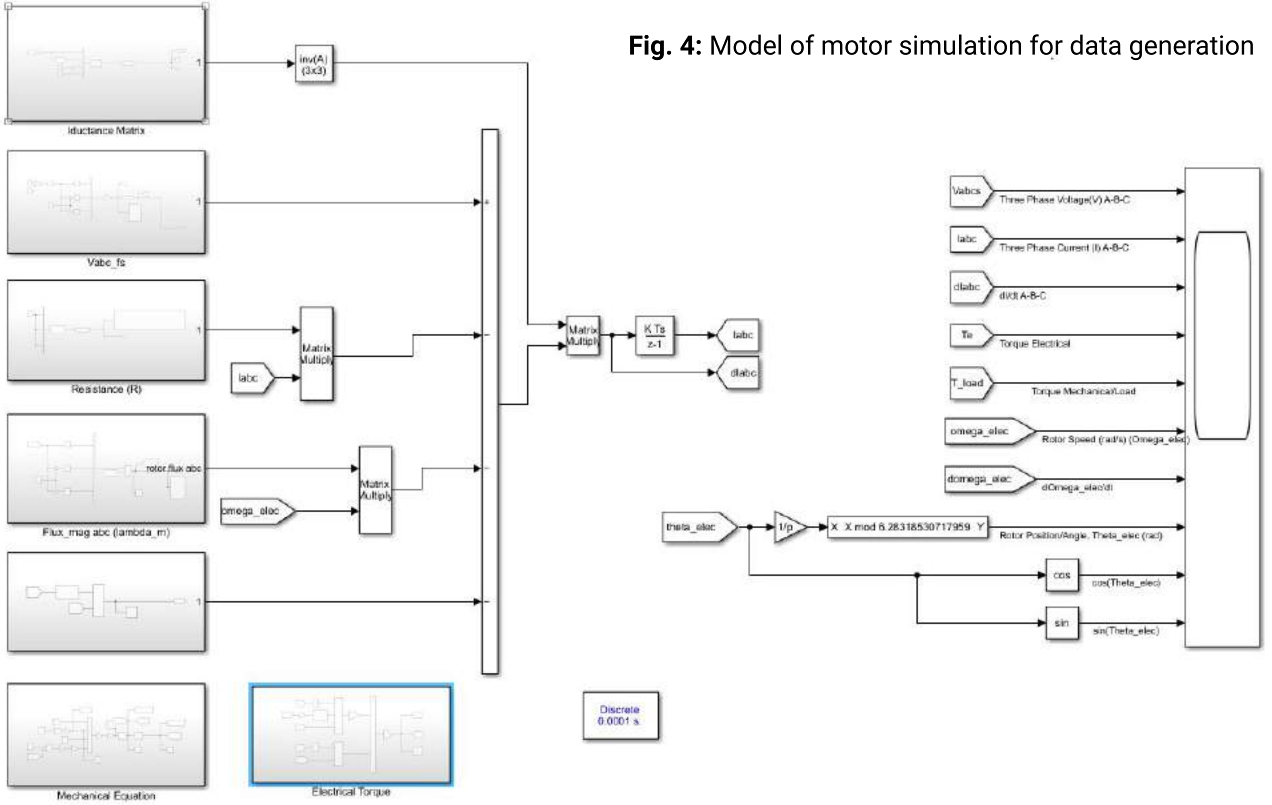
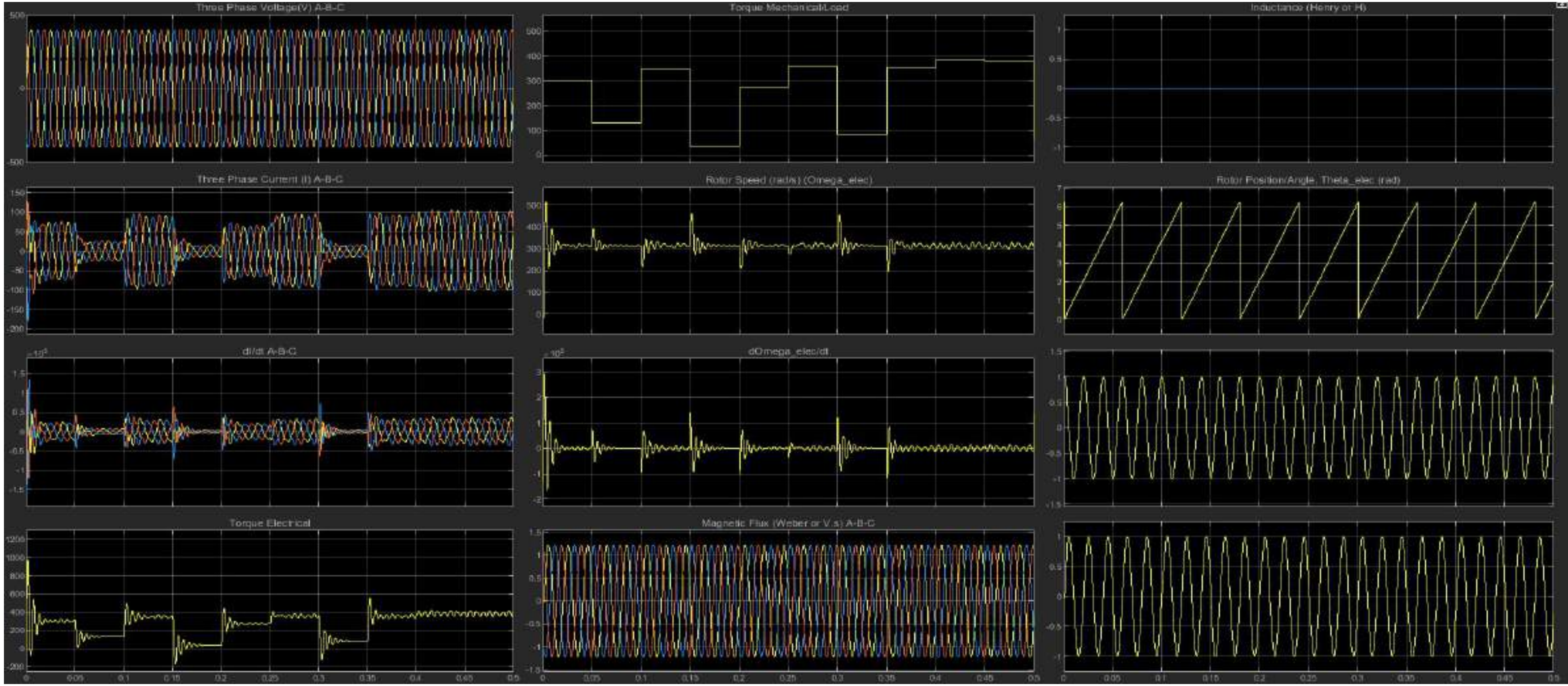


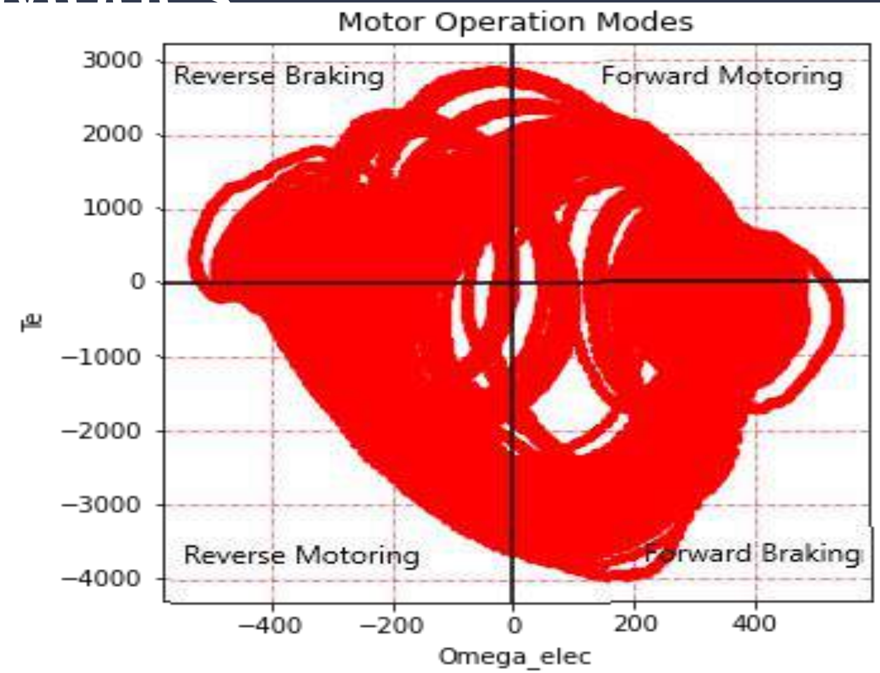
Fig. 4: Model of motor simulation for data generation

Characteristics of generated data

Fig. 5: Characteristics of 0.5 s of data generated by the Matlab/Simulink-Model described before



Dataset Characteristics – Motor Operation Modes



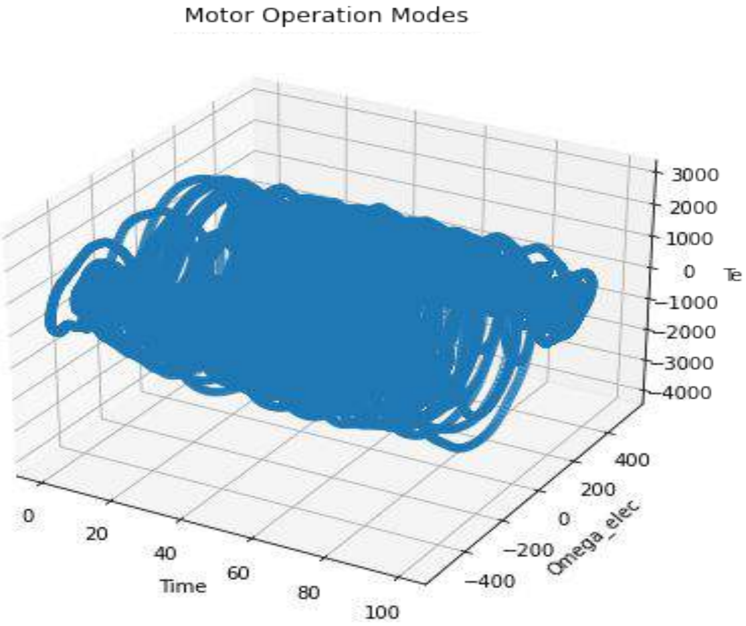
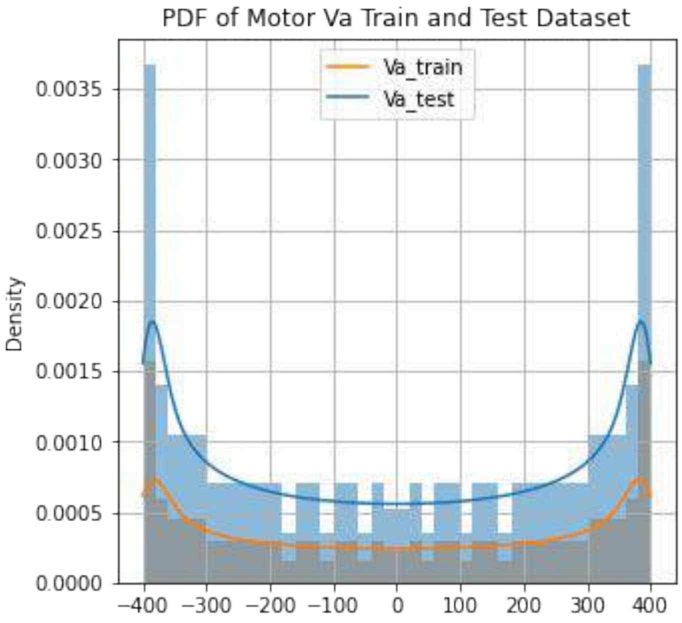


Fig. 6: Basic motor operation simulated: a loop from forward acceleration over forward breaking to backward acceleration and finally backward breaking befor acceleration forward igen (left) - 80 s of the looping signal (right)

Dataset Characteristics – Train/Test Split



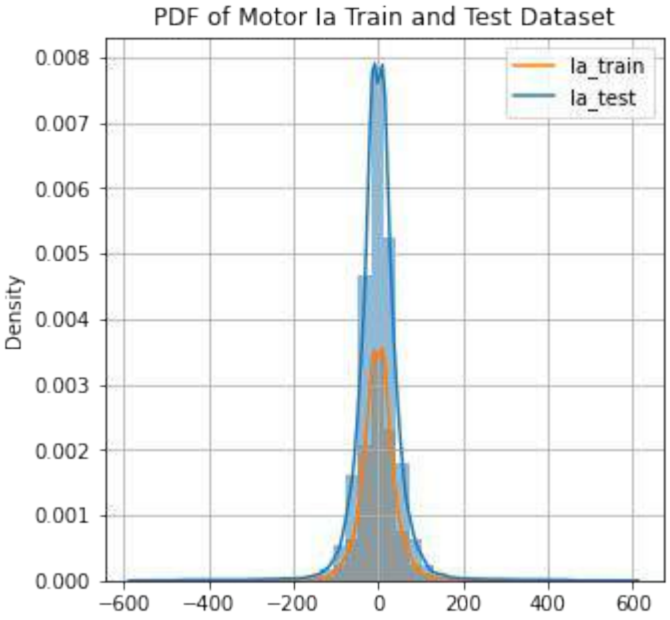


Fig. 7: Probability density Function of the Voltage A (Va) values for train and test data (left) and the Current A (Ia) (right)

Dataset Characteristics – Train/Test Split

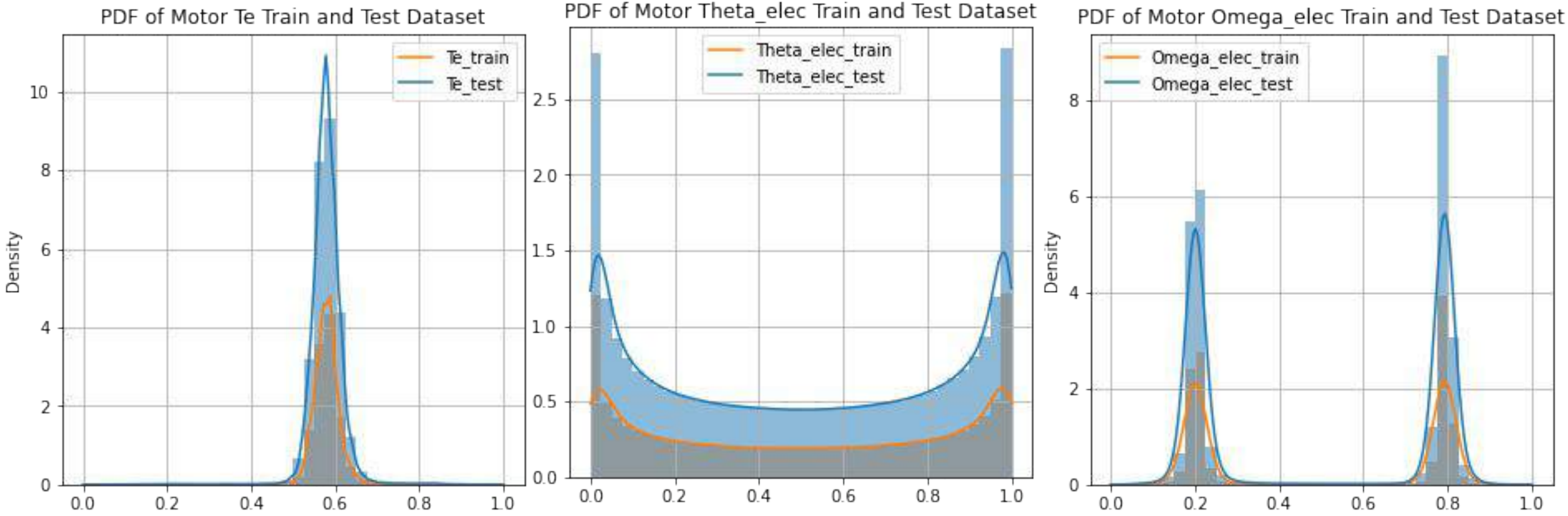


Fig. 8: Min-Max-Scaled Torque, Angle, and Speed data: Torque is slightly biased towards positive operation conditions

Dataset Characteristics – Possible Inconsistencies

- **Sampling Rate - Input sequence length probably too long or too short**
- **Sensor Noise Characteristics**
- **Inadequate Motor Operating conditions**
- **Data split for testing and validation should have equal operating conditions**
- **Inadequate Analytical Model**
- **Bias to Motor ratings - Fine tuning heads or Train with per unit values (MinMaxScale)**

Preprocessing Data – Fixed Sliding Window

- **We use a sliding window as a basis for predicting the next value.**

- **Sequence to vector.**

- **Of course, one of the challenges is the initial estimates of the observer, may be far away from the actual state of a running machine.**

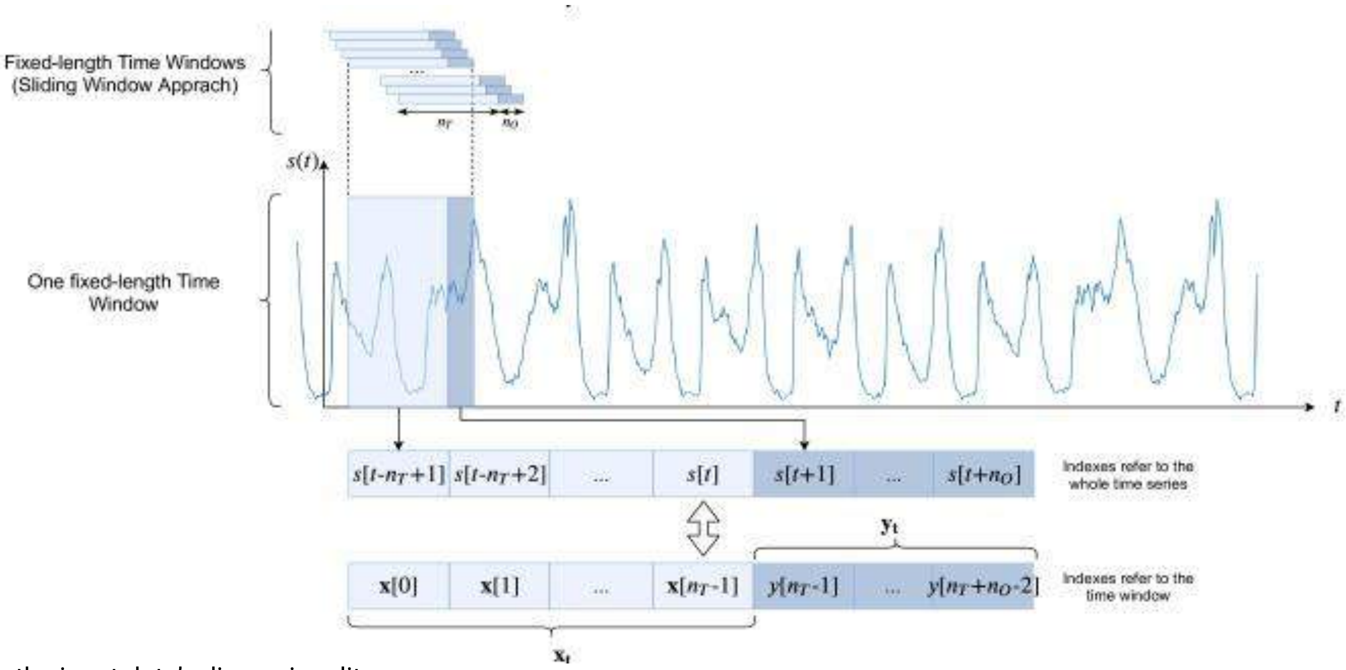


Fig. 9: A sliding window used to reduce the input data’s dimensionality

Preprocessing Data – Addition of Noise (SNRp dB)

We added some white noise (i. e. gaussian noise in the sense of python randn) to all input data. From this time series x(t) becomes the input data, while x(t+1) becomes the value to be predicted.

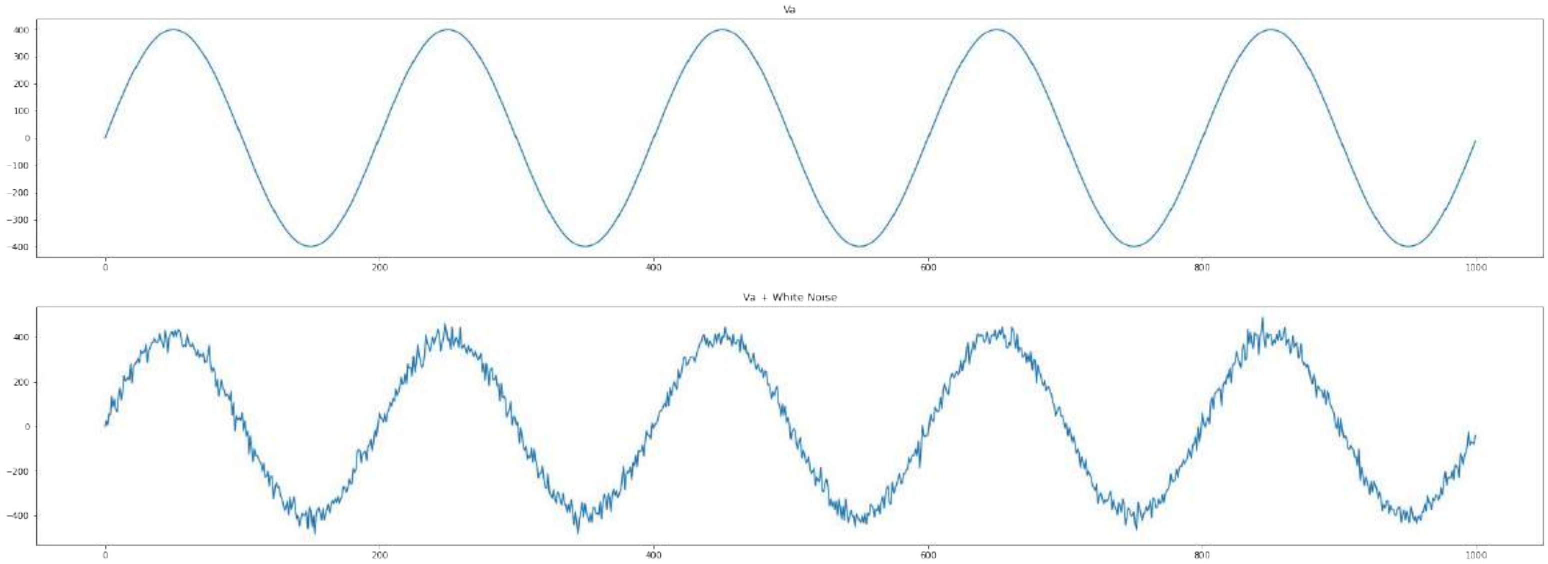


Fig. 10: “Clean” signal generated by simulation model (upper figure) and signal biased by gaussian noise.

A TensorFlow Transformers way

Building on the image classification example, the PMSM Obeserver Design with Transformers mainly consists of these stages:

Preprocessing

- read the simulation generated data
- add noise
- shift data
- scale data

Organize data stream to model

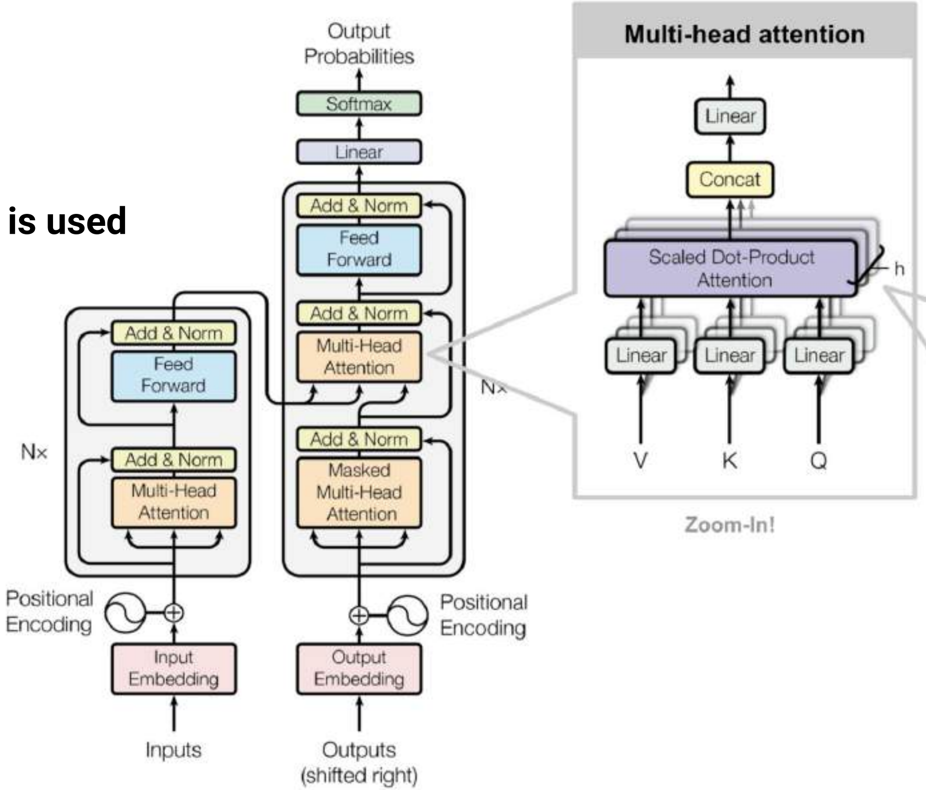
- Use augmentation
- patching

Transformers encoder blocks

Validation/Assessment

Transformer Architecture

Encoder is used



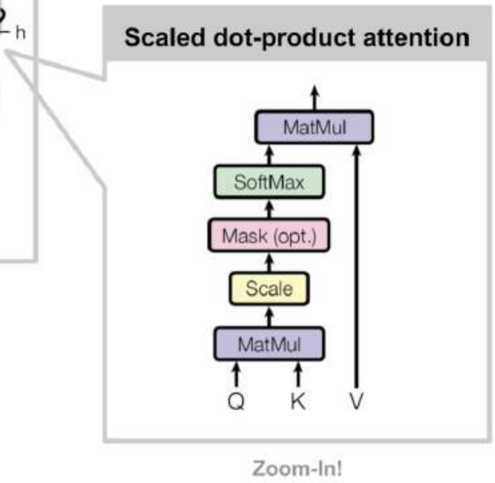
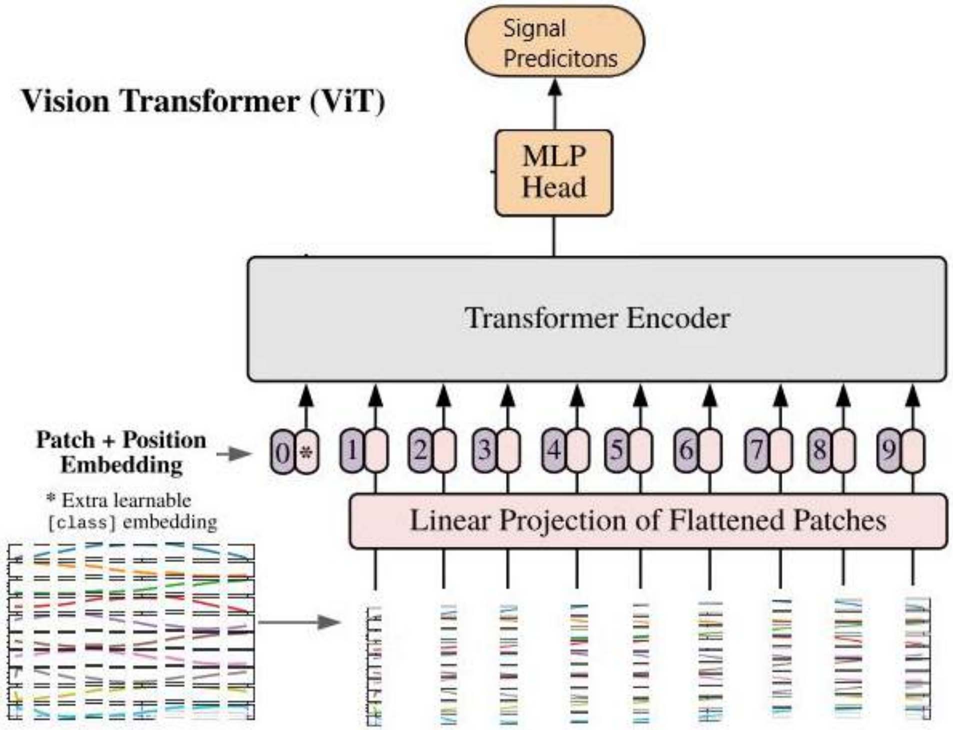


Fig. 11: The TensorFlow Transformers architecture from which we used the encoder part (Source: TensorFlow)

Transformer Architecture – ViT Inspired

Fig. 12: The TensorFlow VisionTransformer Encoder Input section (Source: Tensorflow)



A TensorFlow Transformers way

Organizing the data stream: augmentation

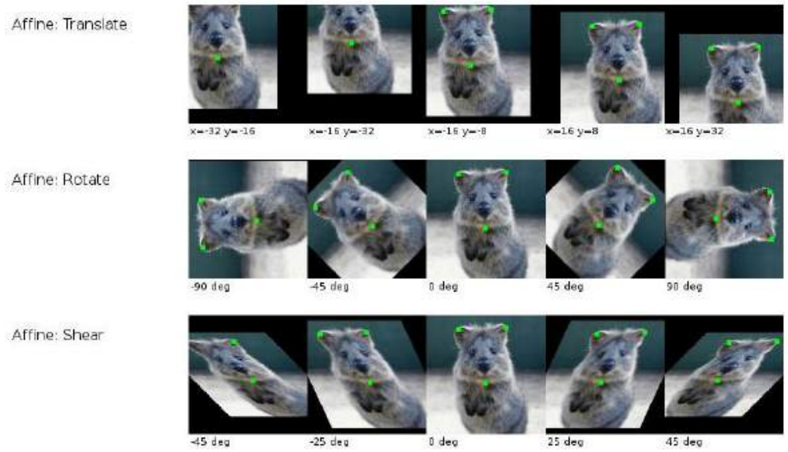


Fig. 13 : Data augmentation techniques used (Source: Tensorflow)

```
data_augmentation = keras.Sequential(
    [
        layers.Reshape(target_shape=(-1, num_x_signals, 1)),
        layers.Resizing(image_size[0], image_size[1]),
        layers.RandomFlip("vertical"),
        # layers.RandomRotation(factor=0.02),
        # layers.RandomZoom(height_factor=0.2, width_factor=0.2),
    ],
    name="Data_Augmentation_Layer",
)
```

A TensorFlow Transformers way

Organizing the data stream:

Patches - Each patch is vector of all input features in a single timestep (value = pixel)

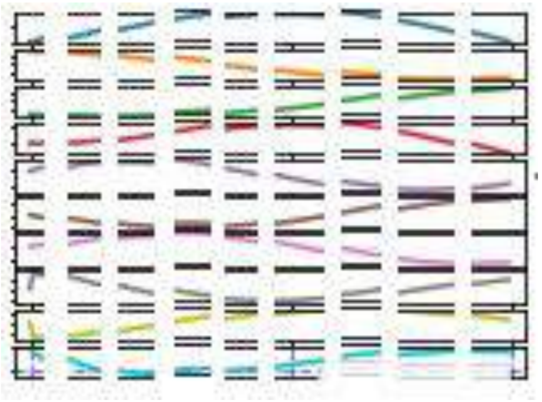
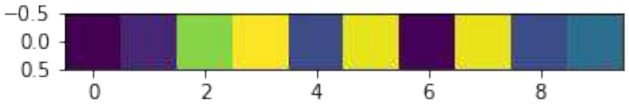


Fig. 13: Making patches from data blocks (Source: TensorFlow)

```
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super(Patches, self).__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches
```



The Model – Flow of Work

- **Noise Addition**
- **Random Windowing**
- **Sensor Faults**

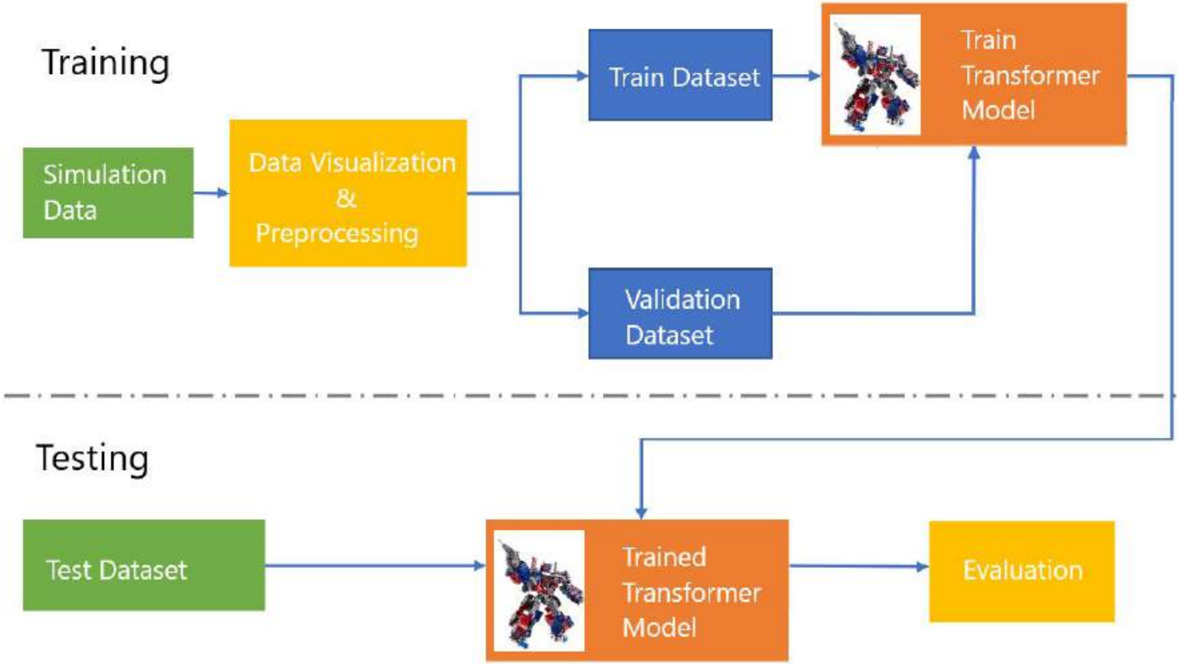


Fig. 14: Workflow of PMSM Ovserver Desing with Transformers

The Model – Summary

Tab. 1: Layerwise summary of the model used

Layer (type)	Output Shape	Param #	Connected to
Input_Layer (InputLayer)	[(None, None, 10)]	0	[]
Data_Augmentation_Layer (Sequential)	(None, 100, 10, 1)	0	['Input_Layer[0][0]']
Patch_Extraction (Patches)	(None, None, 10)	0	['Data_Augmentation_Layer[0][0]']
Patch_Encoding (PatchEncoder)	(None, 100, 10)	1110	['Patch_Extraction[0][0]']
layer_normalization (LayerNormalization)	(None, 100, 10)	20	['Patch_Encoding[0][0]']
- - - - - - -	- - -	- -	- - - -
flatten (Flatten)	(None, 1000)	0	['layer_normalization_8[0][0]']
dropout_8 (Dropout)	(None, 1000)	0	['flatten[0][0]']
dense_9 (Dense)	(None, 1000)	1001000	['dropout_8[0][0]']
dropout_9 (Dropout)	(None, 1000)	0	['dense_9[0][0]']
dense_10 (Dense)	(None, 500)	500500	['dropout_9[0][0]']
dropout_10 (Dropout)	(None, 500)	0	['dense_10[0][0]']
dense_11 (Dense)	(None, 7)	3507	['dropout_10[0][0]']
Output_Layer (Dense)	(None, 7)	56	['dense_11[0][0]']
Reshape_Output (Reshape)	(None, 1, 7)	0	['Output_Layer[0][0]']

Total params: 1,516,673
Trainable params: 1,516,673
Non-trainable params: 0

Baseline GRU Model

Code 1:
The baseline GRU model

```python
def create_GRU_model():
    inputs = layers.Input(name='Input_Layer', shape=(None, num_x_signals))
    # Scale data.
    scaled = CustomScalingLayer(name='MinMaxScaler', units=num_x_signals)(inputs)

    # Denoise data.
    filter = Bidirectional(GRU(name='DenoisingLayer', units=100, return_sequences=True, trainable = False), name='DenoisingLayer')(scaled)
    filter_out = GRU(name='DenoisingOutputLayer', units=num_x_signals, return_sequences=True, trainable = False)(filter)
    filter_out = Reshape(name='ShapedDenoisingOutputLayer', target_shape=(-1, num_x_signals))(filter_out)


    # Extract features
    features = GRU(name='ComputeLayer', units=100, input_shape=(None, num_x_signals), return_sequences=True)(filter_out)
    features_out = GRU(name='FeautureOutputLayer', units=num_y_signals)(features)

    Y_initializer = tf.keras.initializers.RandomNormal(mean=myData[target_names].describe()[1:3].values[0], stddev=myData[target_names].describe()[1:3].values[1].transpose())

    # Predict output from features
    rescaled_out = Dense(num_y_signals, name='RescaledOutputLayer', activation='linear', kernel_initializer=Y_initializer, use_bias=False)(features_out)

    # Reshape ouutputs
    outputs = Reshape(name='UnscaledOutputLayer', target_shape=(-1, num_y_signals))(rescaled_out)

    # Create the Keras model.
    model = keras.Model(inputs=inputs, outputs=outputs, name='GRU_Observer_Model')

    # Create Optimizer.
    optimizer = Adam(learning_rate=1e-2, amsgrad=True)

    model.compile(loss='mse', optimizer=optimizer, metrics=['acc', 'mae', 'mape'], run_eagerly=True)
    model.build((None, None, num_x_signals))

    return model

GRU_Observer_model = create_GRU_model()

GRU_Observer_model.summary()
```

✓ 11.7s

Baseline GRU Model

Tab. 2: Layerwise summary of the baseline GRU model

Layer (type)	Output Shape	Param #
Input_Layer (InputLayer)	[(None, None, 10)]	0
MinMaxScaler (CustomScaling Layer)	(None, None, 10)	0
DenoisingLayer (Bidirection al)	(None, None, 200)	67200
DenoisingOutputLayer (GRU)	(None, None, 10)	6360
ShapedDenoisingOutputLayer (Reshape)	(None, None, 10)	0
ComputeLayer (GRU)	(None, None, 100)	33600
FeautureOutputLayer (GRU)	(None, 7)	2289
RescaledOutputLayer (Dense)	(None, 7)	49
UnscaledOutputLayer (Reshap e)	(None, 1, 7)	0

Baseline GRU Model

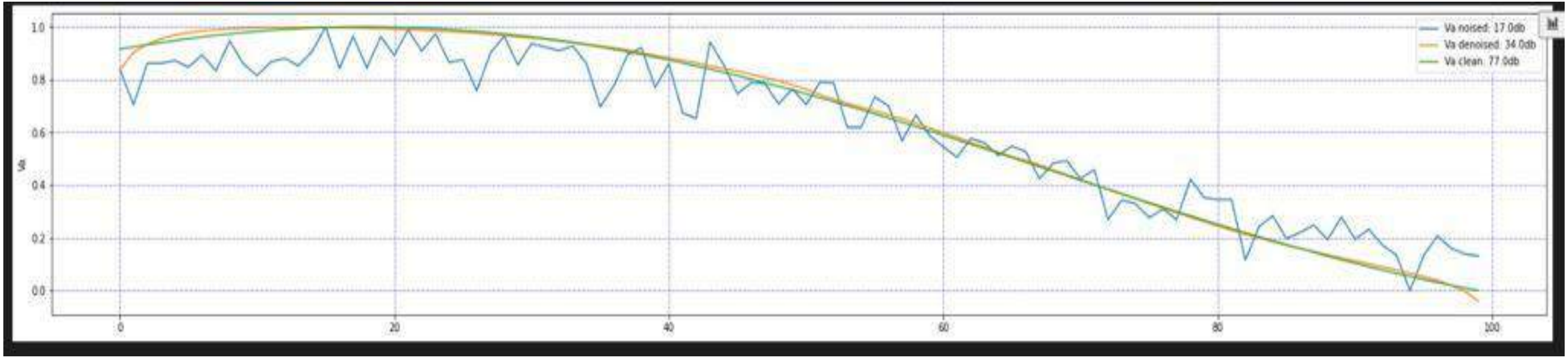
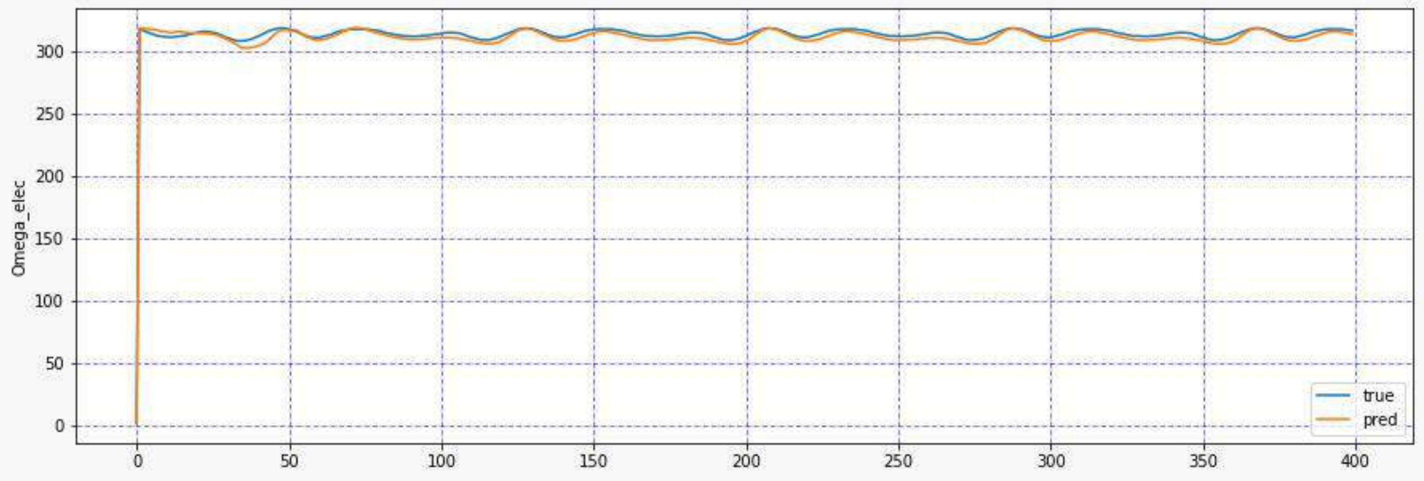
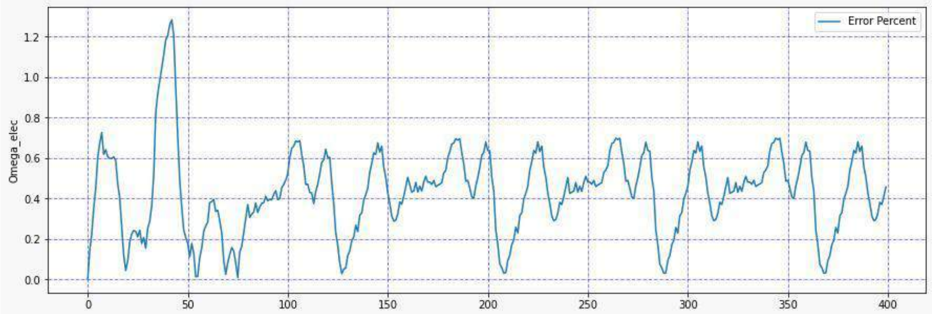


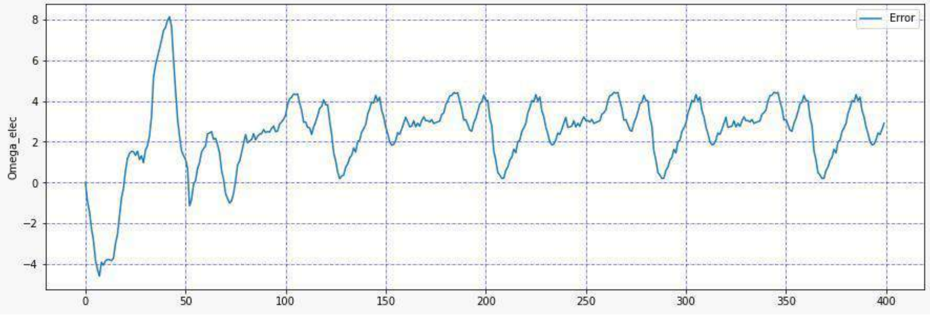
Fig. 15: Voltage A (Va) als model output (“clean”, green), with noise added (blue) and denoised (orange)

Baseline GRU Model – Results Motor Speed

Fig. 16: True and predicted data for the motor speed (right) as well as relative (down left) and absolute error (down right)

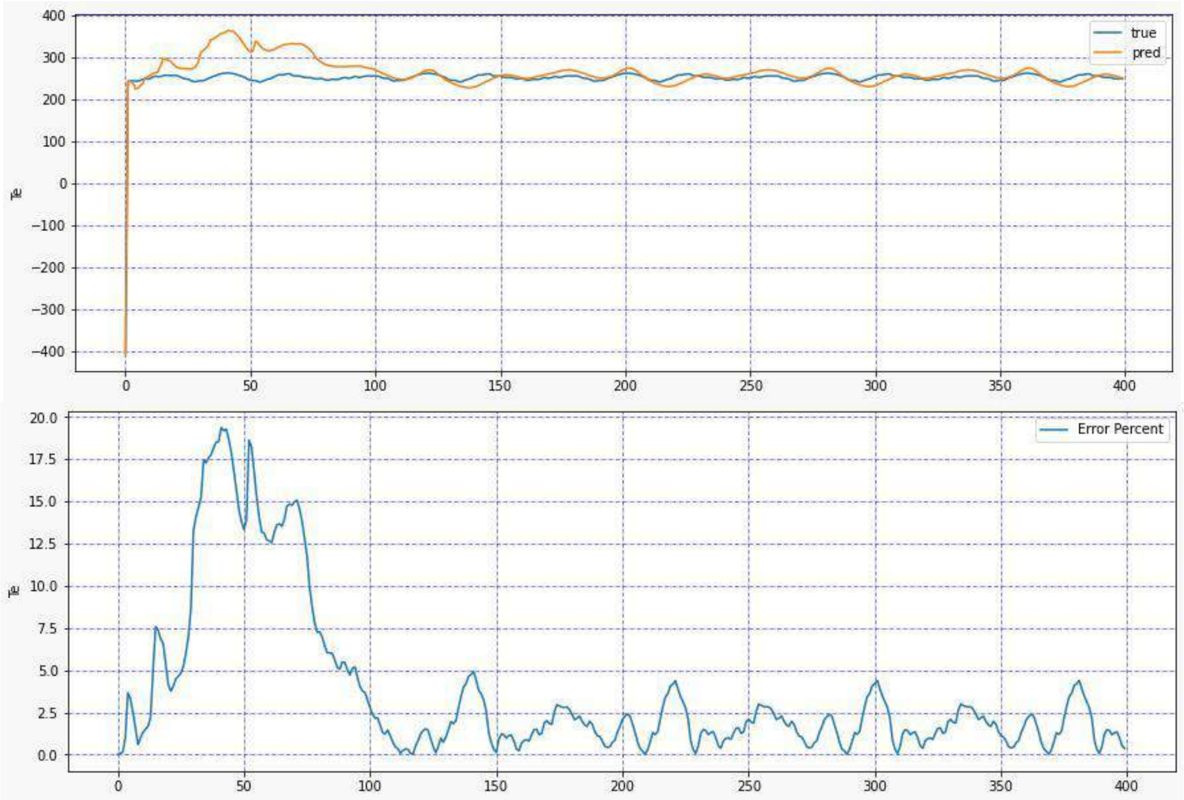






Baseline GRU Model – Motor Torque

Fig. 17: True and predicted values for motor torque (upper right) and absolute error.



Baseline GRU Model – Rotor Angular Position

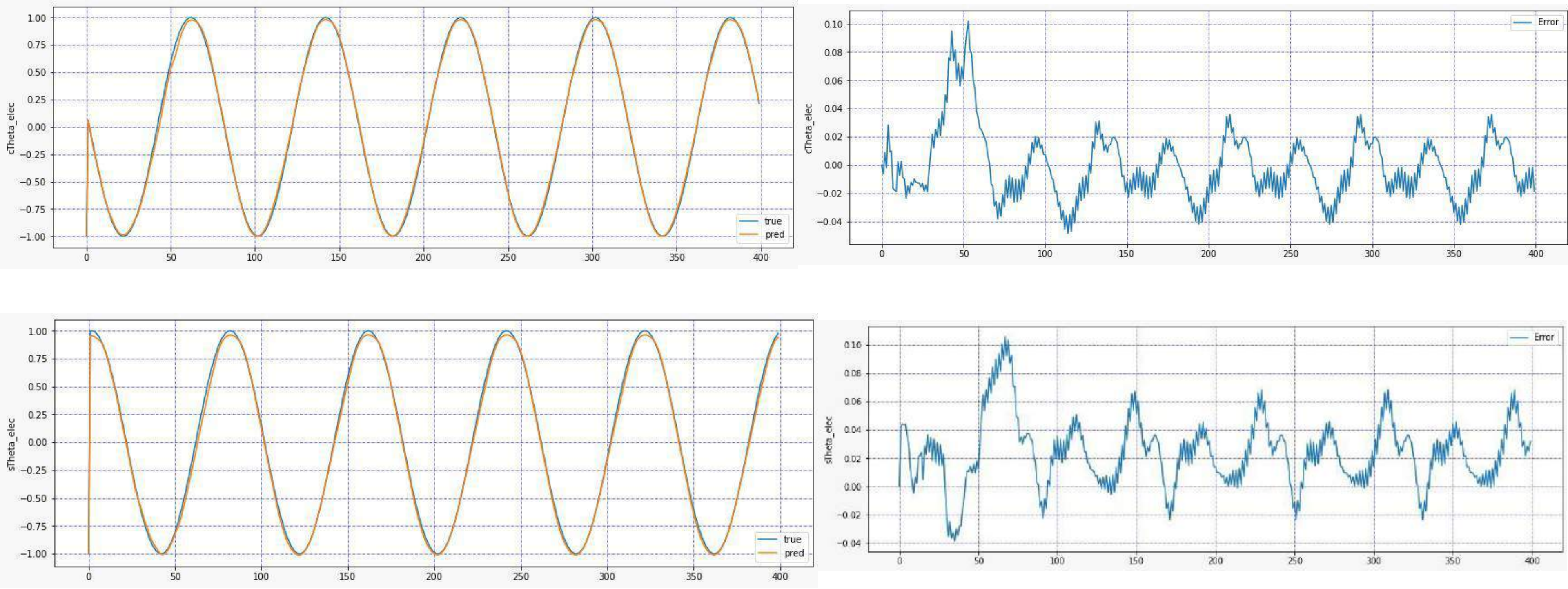
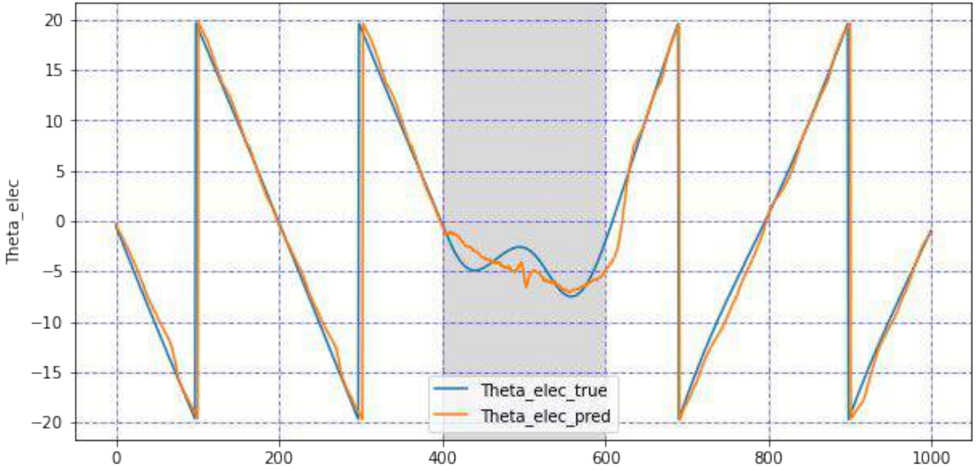
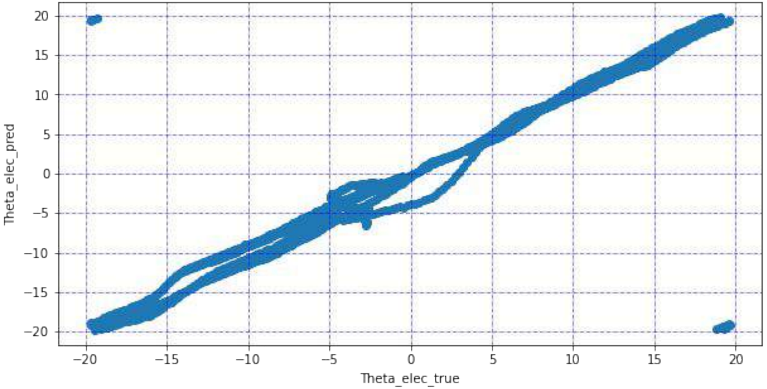


Fig. 18: Given and predicted data for angular rotor position

Model Predictions – Rotor Angle(Position)





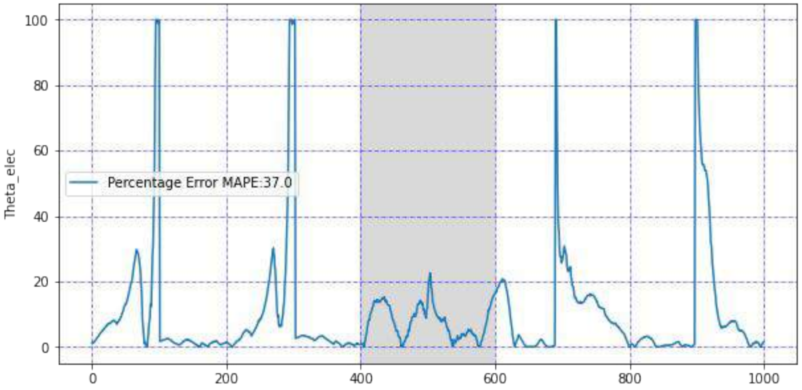
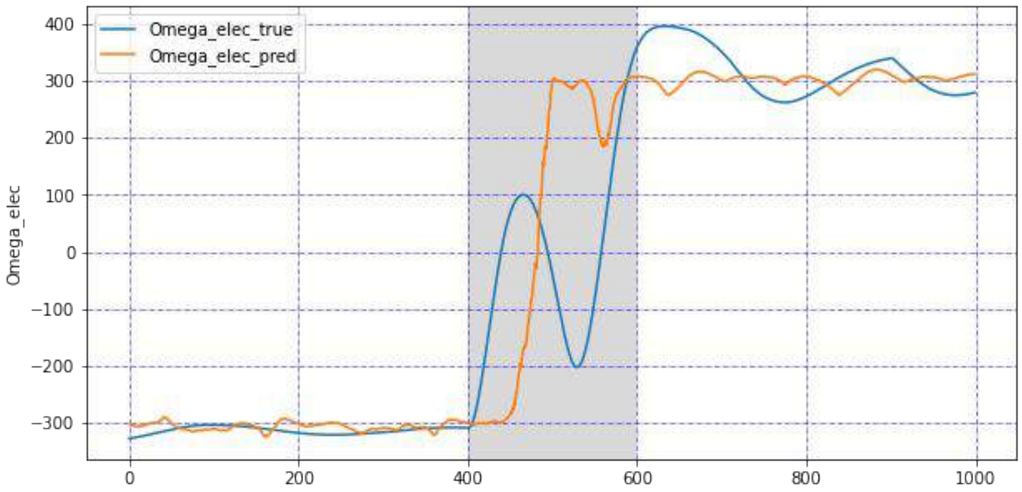
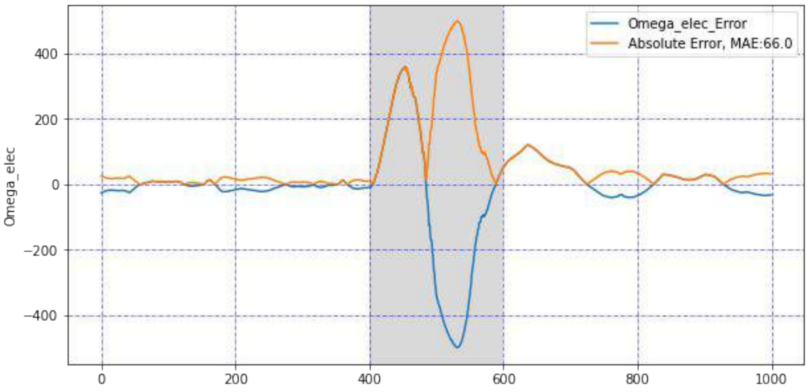


Fig. 19: Rotor angle position: modelled signal (blue) versus prediction (orange)

Model Predictions – Motor Speed (radians)





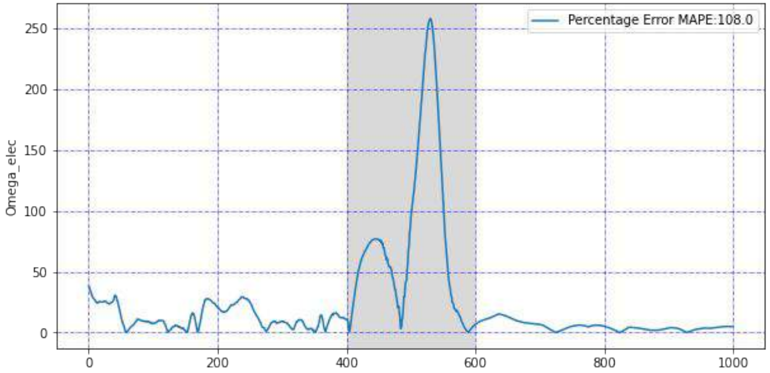
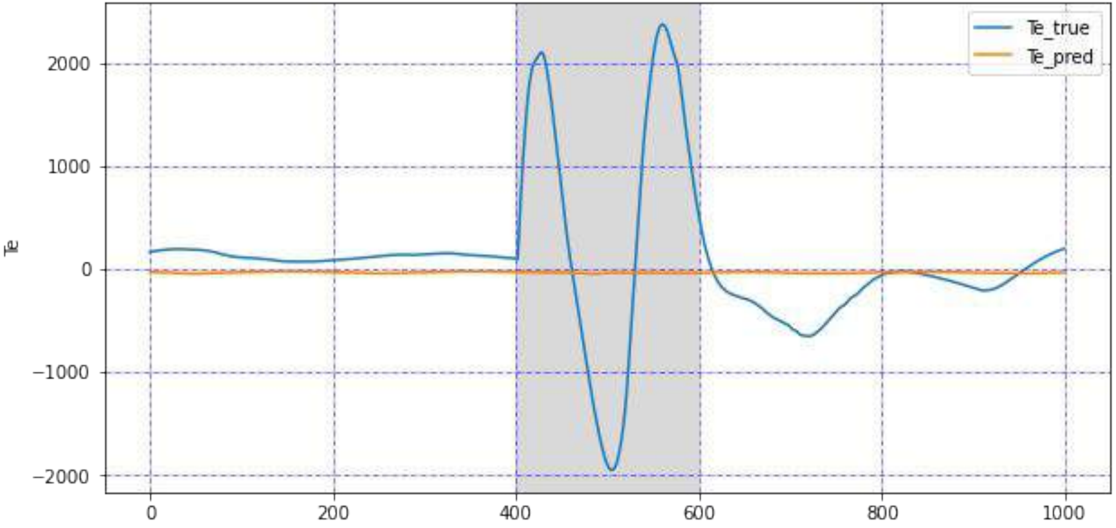


Fig. 20: Motor speed: modelled signal (blue) versus prediction (orange)

Model Predictions - Motor Speed (radians)



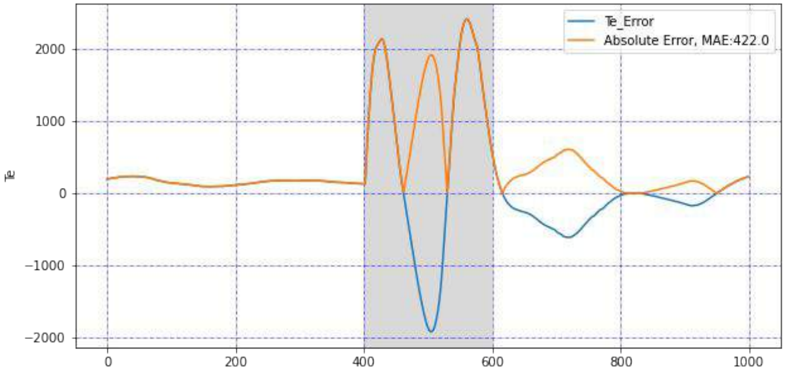


Fig. 21: Motor speed in radias: modelled signal (blue) versus prediction (orange)

Conclusion

Experiences

- model does not pick up unexpected time line events
- model does not pick up transient events adequately
- torque (crucial!) prediction is horrible, while rotor position and motor speed shows good results so far
- patching as a tokenization mechanism is questionable - using time step values as words could be useful, too
- simplification is needed: the model needs to run on a small processing unit

Current Challenges

- Quitting/crashing colab
- Crashing vs Code
- Hyperparameter selection and tuning

Literature Review

D. Janiszewski, "Unscented Kalman Filter for sensorless PMSM drive with output filter fed by PWM converter," IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society, 2012, pp. 4660-4665, doi: 10.1109/IECON.2012.6389495.

The Pulse Width Modulation (PWM) leads to high rate of voltage rise du/dt with long cable runs causes harmonic frequencies, the ripples, tension in motor winding, high frequency leakage currents, bearing damage, insulation failure, power losses, high acoustic noise levels, parasitic earth currents. There’s the risk of signal attenuation and phase shifting using conventional filters.

- Linear model after undergoing linear transformation by (KF and EKF) should maintain gaussian property (otherwise no convergence).
- The Unscented Kalman Filter (UKF) produces several sampling points (Sigma points) around the current state estimate based on its covariance.
- Then, propagating these points through the nonlinear map to get more accurate estimation of the mean and covariance of the mapping results.
- Avoids the need to calculate the Jacobian, hence computational load as the Extended Kalman Filter.
- Noise distribution parameters, and covariance influence performance.

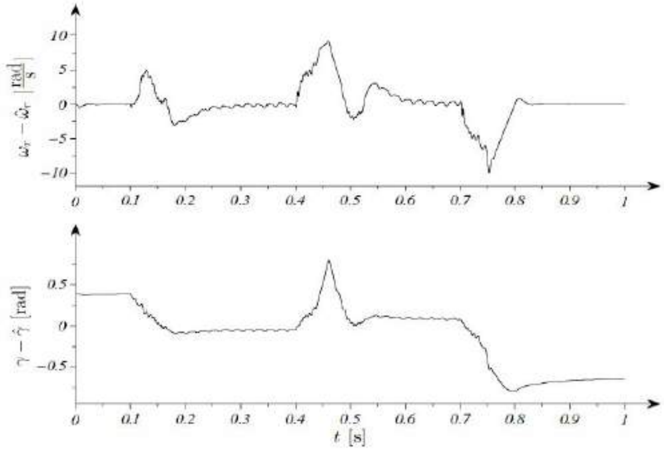
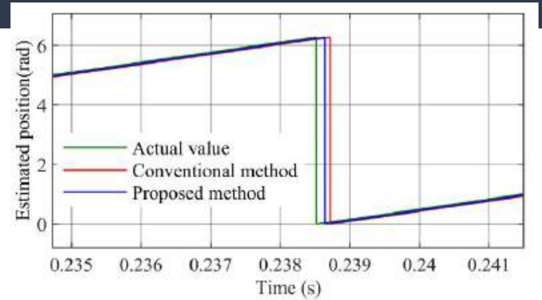
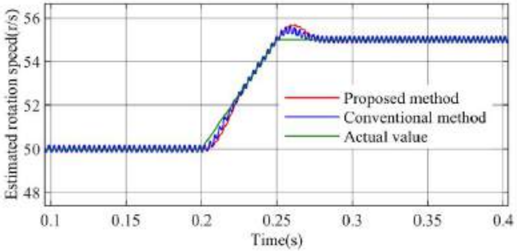


Fig. 5: Estimation errors during speed response

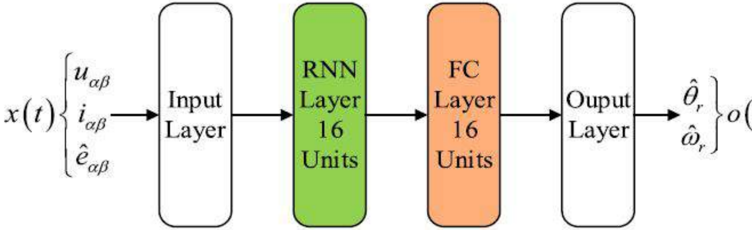
Literature Review

Qingzhong Gao, Changsheng Zong, Rotor position estimation method of PMSM based on recurrent neural network, Energy Reports, Volume 8, Supplement 1, 2022, Pages 883-889, ISSN 2352-4847, https://doi.org/10.1016/j.egyr.2021.11.091. (https://www.sciencedirect.com/science/article/pii/S2352484721012361)





- Existing methods cannot provide a satisfactory tracking performance under harmonic polluted condition in wide operating range, which degrade the position detecting accuracy.
- To tackle the nonlinear problem in wide operating range of PMSM, RNN is used as a nonlinear dynamic system to establish the mapping relationship of the voltages of stator and rotor position.
- The proposed method can also provide frequency-adaptive filtering capability to remove the affect of harmonic.
- To validate the method, a comparison simulations with conventional SRF-PLL method are used under speed varying and harmonic disturbances conditions.
- Rotor position estimation average error 4deg.
- The oscillation error of conventional method in rotor speed is almost
- 0.3 r/s, whereas the proposed method only 0.1 r/s.



Literature Review – 3 useful videos

An illustrated guide to Transformers

https://youtu.be/4Bdc55j80l8

An image is worth 16 by 16 word:

https://youtu.be/TrdevFK_am4

Vision Transformers: Keras code example

https://youtu.be/i2_zJ0ANrw0