Pat Nyem Ake' – PNA Documentation

Submitted on

September 21, 2024

for

ExxonMobil DataWorks Challenge 2024

Volintine Ander Jiloh Jr.	volintine_21001524@utp.edu.my
Mclaren Maja Franklin	mclaren_21001469@utp.edu.my
Eddry Haqimy Yusri	eddry_21001460@utp.edu.my
Zechariah Hwong	zechariah_21001596@utp.edu.my

Representing

Universiti Teknologi PETRONAS

Supervised by

AP Ir Dr. Idris B Ismail (idrisim@utp.edu.my)

Setup

The model was entirely developed in a Jupyter notebook (located in `../workspace/source_code/`), hosted in a Google Colab runtime. All necessary libraries are imported in the beginning code block. It is advised to run the notebook in a Google Colab instance.

Special Instructions

For ease of running the source code, upload the `dataset_zipped.zip` file (located in `../workspace/external_files/`) into the Colab runtime. This zip file contains the training, validation, and testing datasets provided by the DataWorks team. Please make sure to connect to a **T4 GPU runtime** instead of the default CPU runtime.

Development

Data Preprocessing

This step begins with importing all the training and validation dataset into a list. It was extracted using `pd.read_csv()` for all files in the folder and placed into a large pandas dataframe containing all 62 reservoir training data for state.csv and production.csv. The dataframes are then put into a list.

```
[ ] dflist_validation_inputs = [pd.read_csv(validation_path + path + '/state.csv') for path in pathlist_to_validation]
    dflist_validation_outputs = [pd.read_csv(validation_path + path + '/production.csv') for path in pathlist_to_validation]

[ ] df_inputs_list_validation = [reservoir for reservoir in dflist_validation_inputs] # Create list of dataframes for each reservoir's state.csv
    df_outputs_list_validation = [reservoir for reservoir in dflist_validation_outputs] # Create list of dataframes for each reservoir's production.csv
```

Next, the date column was converted into an index variable.

```
# Training production.csv dataset - index by date
for reservoir in range(len(df_outputs_list)):
  df_outputs_list[reservoir]['Date'] = pd.to_datetime(df_outputs_list[reservoir]['Date'])
  df_outputs_list[reservoir].set_index('Date', inplace=True)

# Validation production.csv dataset - index by date
for reservoir in range(len(df_outputs_list_validation)):
  df_outputs_list_validation[reservoir]['Date'] = pd.to_datetime(df_outputs_list_validation[reservoir]['Date'])
  df_outputs_list_validation[reservoir].set_index('Date', inplace=True)
```

The spatial (state) data is reshaped into a 3D array by vertically stacking the data.

```
df_spatial = (np.vstack(dflist_training_inputs)).reshape(62, 5183, 8) # vstack and reshape spatial training dataset
df_spatial_validation = (np.vstack(dflist_validation_inputs)).reshape(26, 5183, 8) # vstack and reshape spatial validation dataset
```

For the temporal (production) datasets of all reservoirs to be predicted by the model, the data dimensions must be homogeneous. The temporal dataset of each reservoir has uneven number of entries.

To maximize the number of reservoirs used in training, we first find the minimum number of rows any reservoir could have. We refer to the process of removing further rows beyond the minimum as *culling*.

```
[ ]  # This block finds the minimum number of production.csv rows. It should be 260.
     dummy = []
     for i in range(len(df_outputs_list)):
       dummy.append(df_outputs_list[i].shape[0])
     print(min(dummy))
```

```
260
```

```
[ ]  # Cull the training temporal dataset
     culled_outputs_list = []
     for i in range(len(df_outputs_list)):
       culled_outputs_list.append(df_outputs_list[i][:260])

     # Cull the validation temporal dataset
     culled_outputs_list_validation = []
     for i in range(len(df_outputs_list_validation)):
       culled_outputs_list_validation.append(df_outputs_list_validation[i][:260])
```

```
df_temporal_culled = np.vstack(culled_outputs_list).reshape(62, 260, 3)
df_temporal_culled_validation = np.vstack(culled_outputs_list_validation).reshape(26, 260, 3)
```

Prior to training, the data is first normalized using `MinMaxScaler()`. However, the scaler expects ndim=2, thus the dataset is reshaped into 2D [Reservoir*Rows, Features]. Once normalization is done, the data is reshaped back into its original shape.

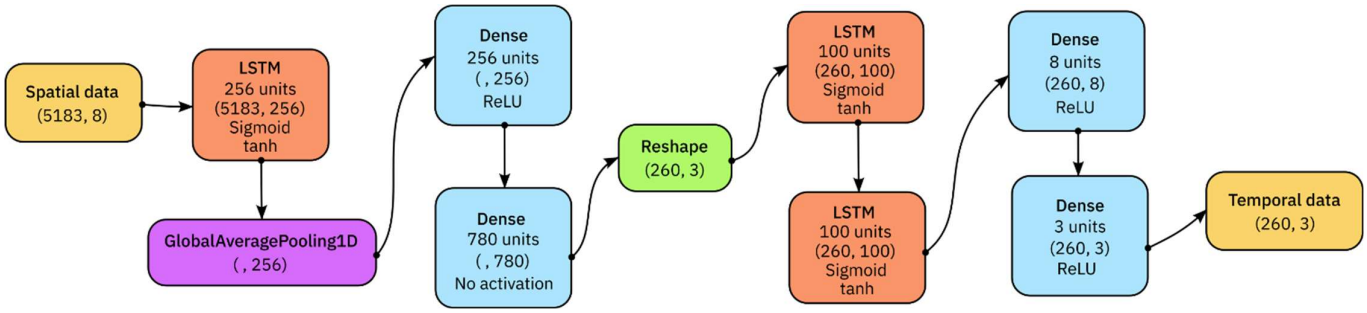
```
train_scaler_X = MinMaxScaler()
train_scaler_Y = MinMaxScaler()

normalized_Y = train_scaler_Y.fit_transform(df_temporal_culled.reshape(62*260, 3))
normalized_X = train_scaler_X.fit_transform(df_spatial.reshape(62*5183, 8))

normalized_Y = normalized_Y.copy().reshape(62, 260, 3)
normalized_X = normalized_X.copy().reshape(62, 5183, 8)
```

```
valid_scaler_X = MinMaxScaler()
valid_scaler_Y = MinMaxScaler()

normalized_Y_valid = valid_scaler_Y.fit_transform(df_temporal_culled_validation.reshape(26*260, 3))
normalized_X_valid = valid_scaler_X.fit_transform(df_spatial_validation.reshape(26*5183, 8))

normalized_Y_valid = normalized_Y_valid.copy().reshape(26, 260, 3)
normalized_X_valid = normalized_X_valid.copy().reshape(26, 5183, 8)
```

```
X_train = normalized_X
y_train = normalized_Y

X_valid = normalized_X_valid
Y_valid = normalized_Y_valid
```

Model architecture

The normalized dataset is then fed into an LSTM-based model.



1. **LSTM Layer 1**:
    - **Units**: 256
    - **Input Shape**: (5183, 8) — This layer expects sequences of 5183 rows of spatial data in 8 features.
    - **return_sequences=True**: This configuration ensures that the layer returns the full sequence of outputs, which will be fed into the next layer.
2. **Global Average Pooling 1D Layer**:
    - **Function**: This layer reduces the dimensionality by calculating the average of each feature map across all 5183 timesteps, significantly reducing the size of the data. Returns an array of length 256.
    - **Purpose**: This step simplifies the model, reducing computational complexity and helping prevent overfitting.
3. **Dense Layer 1**:
    - **Units**: 256
    - **Activation Function**: ReLU (Rectified Linear Unit) — Introduces non-linearity into the model, allowing it to learn complex patterns from the data.
4. **Dense Layer 2 (Output Layer)**:
    - **Units**: 260 * 3 = 780 (this is equivalent to generating 260 sequences, each containing 3 features)
    - No activation is applied, implying a linear transformation of the input data.
    - **Purpose**: This layer provides an output that can be reshaped into a target shape for further processing.

5. **Reshape Layer**:
    - **Reshaping**: The output is reshaped from a 1D tensor of size 780 into a 2D tensor of shape (260, 3). This means the model now outputs 260 timesteps with 3 features each.
6. **LSTM Layer 2**:
    - **Units**: 100
    - **return_sequences=True**: This layer continues to return a full sequence for further temporal processing.
7. **LSTM Layer 3**:
    - **Units**: 100
    - **return_sequences=True**: Similar to the previous layer, it returns a sequence output, allowing the model to capture temporal patterns in the data.
8. **Dense Layer 3**:
    - **Units**: 8
    - **Activation Function**: ReLU — This layer introduces further non-linearity to refine the model's ability to learn.
9. **Dense Layer 4 (Final Output Layer)**:
    - **Units**: 3
    - **Activation Function**: ReLU — Force the output to be strictly positive. This is because the output dataset contains no negative production values. This layer outputs 3 features for each timestep.

The architecture is implemented as below:

```
model = Sequential()

# Example LSTM layer
model.add(LSTM(256, input_shape=(5183, 8), return_sequences=True))

# Global Average Pooling to reduce dimensionality
model.add(GlobalAveragePooling1D())

# Dense layers to adjust output
model.add(Dense(256, activation='relu'))
model.add(Dense(260*3))  # Output dimension (260 sequences, 3 features each)

# Reshape output to desired shape (260, 3)
model.add(Reshape((260, 3)))
model.add(LSTM(100, return_sequences = True))
model.add(LSTM(100, return_sequences = True))
model.add(Dense(8, activation = 'relu'))
model.add(Dense(3, activation = 'relu', ))

# # Compile the model
# model.compile(optimizer='adam', loss='mean_squared_error')  # Use appropriate loss for your task

model.summary()
```

Custom Loss

Since the predicted output is cumulative, the predictions must be monotonically increasing with time. This constraint is added to the loss function by a penalty based on the relative magnitudes of the features (Oil/Water/Gas cumulative production) and whether the previous time step prediction is smaller than the current prediction.

The heuristically determined loss function used is shown below. These hyperparameters were chosen over trial and error. The 0.7 figure is obtained as a rule of thumb from experimentation. *a, b*, and *c* are feature-specific penalties.

$$\text{Modified loss} = \text{MSE}(Y_{\text{true},i}, Y_{\text{pred},i}) - \kappa(a\phi(\text{Gas}) + b\phi(\text{Oil}) + c\phi(\text{Water}))$$

$$\text{Penalty factor, } \kappa = 0.001$$

$$\phi(\text{feature}) = \begin{cases} Y^{\text{feature}}_{\text{pred},i} - Y^{\text{feature}}_{\text{pred},i+1}, & \text{if } Y^{\text{feature}}_{\text{pred},i} - Y^{\text{feature}}_{\text{pred},i+1} < 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\sqrt{a^2 + b^2 + c^2} \approx 0.7 \text{ chosen } \{a = 0.1, b = 0.25, c = 0.6 \mid a < b < c\}$$

The custom loss function is implemented as below:

```
penalty_factor = 0.001

def modified_mse(y_true, y_pred):
    """Custom loss function that adds a penalty if the previous prediction is greater than the current prediction.

    Args:
        y_true: The true labels.
        y_pred: The predicted labels.

    Returns:
        The loss value.
    """

    # Calculate the difference between current and previous predictions
    diff_1 = y_pred[:, 1:, 0:1] - y_pred[:, :-1, 0:1]
    diff_2 = y_pred[:, 1:, 1:2] - y_pred[:, :-1, 1:2]
    diff_3 = y_pred[:, 1:, 2:3] - y_pred[:, :-1, 2:3]
    # sign = tf.cast(tf.greater(diff, 0), tf.float32)
    # if sign:
    #   mask = 0
    # else:
      # mask = tf.cast(-1.0*(diff*diff), tf.float32)
    # mask = tf.where(diff > 0, 0.0, -1.0*diff)
    mask_1 = tf.where(diff_1 > 0, 0.0, diff_1)
    mask_2 = tf.where(diff_2 > 0, 0.0, diff_2)
    mask_3 = tf.where(diff_3 > 0, 0.0, diff_3)

    # Use tf.greater instead of > for comparison in graph mode
    # mask = tf.cast(tf.greater(diff, 0), tf.float32)

    # Calculate the penalty based on the mask and a penalty factor
    penalty_1 = -0.1 * penalty_factor * mask_1
    penalty_2 = -0.25 * penalty_factor * mask_2
    penalty_3 = -0.60 * penalty_factor * mask_3


    # Calculate the mean squared error (MSE) loss
    mse = tf.keras.losses.MeanSquaredError()(y_true, y_pred) # Use MSE as a function

    # Combine the MSE and penalty
    loss = mse + tf.reduce_sum(penalty_1) + tf.reduce_sum(penalty_2) + tf.reduce_sum(penalty_3) # Ensure penalty is a scalar

    return loss
```

Training

```
model.compile(optimizer='adam', loss=modified_mse)
history = model.fit(X_train, y_train, validation_data=(X_valid, Y_valid), epochs=75, verbose=1)
```

After defining the architecture of the model and modified loss function, the model is trained in 75 epochs. Training and validation losses are plotted to look for signs of over/underfit.

```
# Plot training & validation loss values
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.savefig('train_valid_loss.png', dpi=300)
plt.show()
```

The produced model is then saved and used to produce predictions using the validations dataset. However, the output predictions are normalized values since normalized datasets were used to train the model. Thus, before plotting the values the normalization must be inverted.

```
model.save('PNA_model_loss_trick.keras')
model = load_model('/content/PNA_model_loss_trick.keras', custom_objects={'modified_mse': modified_mse})
```

```
# Make predictions
train_predictions = model.predict(X_train)
validation_predictions = model.predict(X_valid)
```

As discussed earlier, the normalization functions cannot work with 3D arrays. Thus, the data is temporarily converted to 2D arrays before being reshaped back. The validation data can then be plotted to verify the performance of the model and then tested with normalized test reservoir state data.

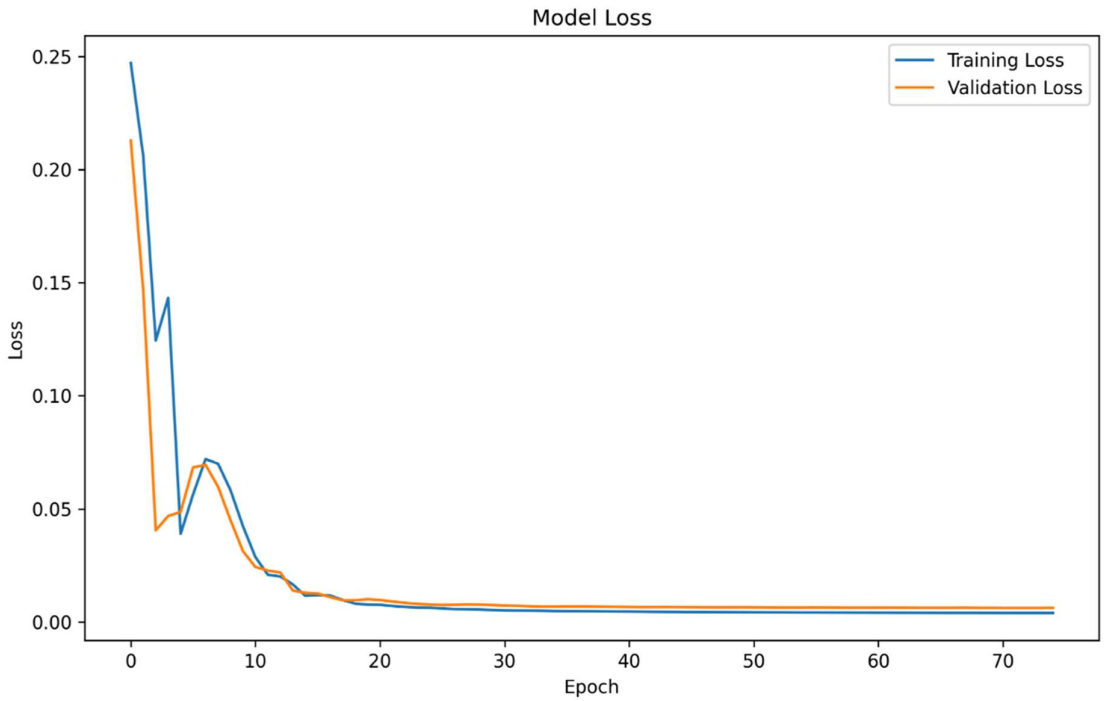
Denormalize predictions

```
[ ] # Inverse transform the predictions
    train_predictions_denorm = train_scaler_Y.inverse_transform(train_predictions.reshape(62*260, 3))
    train_predictions = train_predictions_denorm.copy().reshape(62, 260, 3) # Reshape back to original shape
```

```
[ ] validation_predictions_denorm = valid_scaler_Y.inverse_transform(validation_predictions.reshape(26*260, 3))
    validation_predictions = validation_predictions_denorm.copy().reshape(26, 260, 3) # Reshape back to original shape
```

Findings/Visualization

Model Performance



The loss curves show that the validation and training losses decrease at roughly the same rate which converges at near the end as the model is being trained. This indicates that the model is not showing signs of overfitting and underfitting and has reasonable generalization ability.

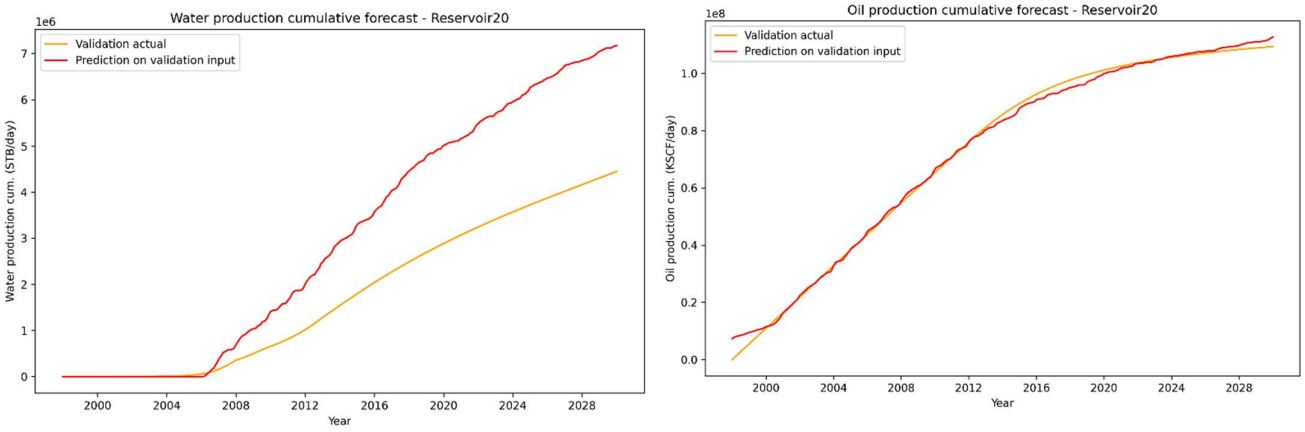
Model Output

The model validation prediction was compared to the production validation values. Since the dataset is in large order of magnitudes, the normalized dataset for the actual dataset and predicted is used. This can be done by storing `Y_valid` into an array of shape [26,260,3] while the normalized prediction can be obtained by running model inference without denormalization. The metrics of Mean Absolute Error (MAE), Mean Square Error (MSE) and Root-Mean Square Error (RMSE) was used and results in the following:

Validation Metrics

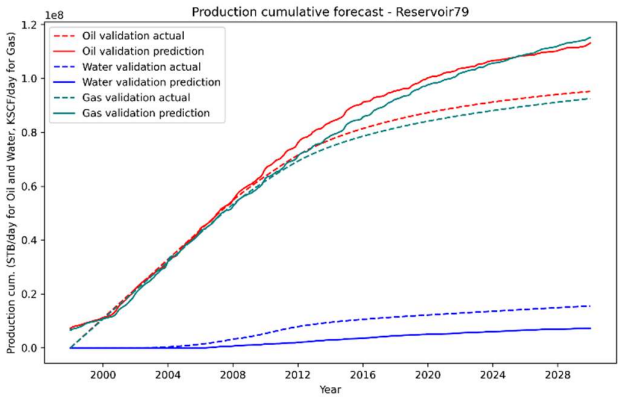
Production Type	Average Metrics Across All Reservoirs
Gas Production Cumulative	**MAE:** 0.0300 **MSE:** 0.0028 **RMSE:** 0.0404
Oil Production Cumulative	**MAE:** 0.0331 **MSE:** 0.0031 **RMSE:** 0.0432
Water Production Cumulative	**MAE:** 0.0673 **MSE:** 0.0126 **RMSE:** 0.0874

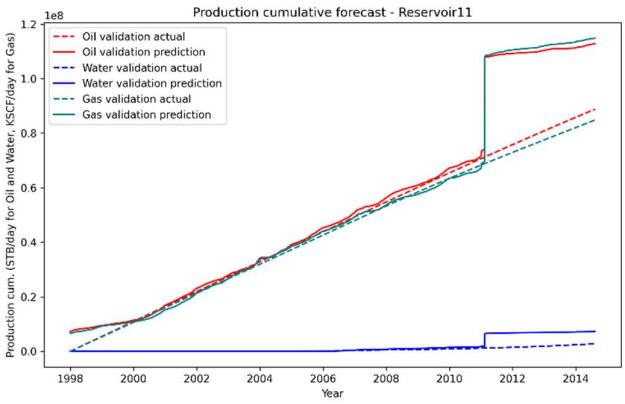
The model performs reasonably well in predicting gas and oil production, with relatively low average error metrics. Water production prediction exhibits higher error rates, suggests that the model may need refinement for more accurate water production forecasts. This could be due to increased variability or more complex factors affecting water production, which the model struggles to capture accurately. However, this error in the feature was expected as it consistently showed in varying degrees when tested with several iterations of the model. We attribute this to a lack of diversity in the training data.



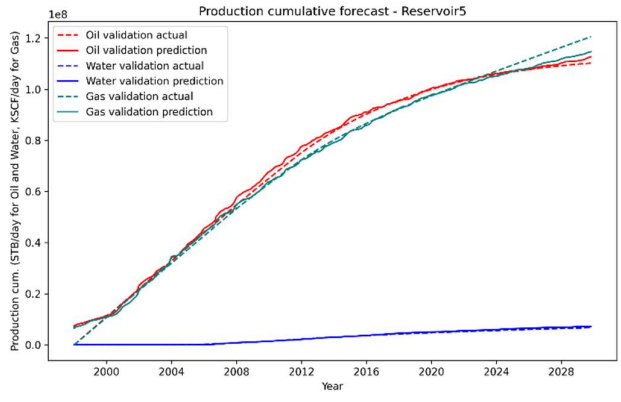
Water production tends to have higher prediction errors.

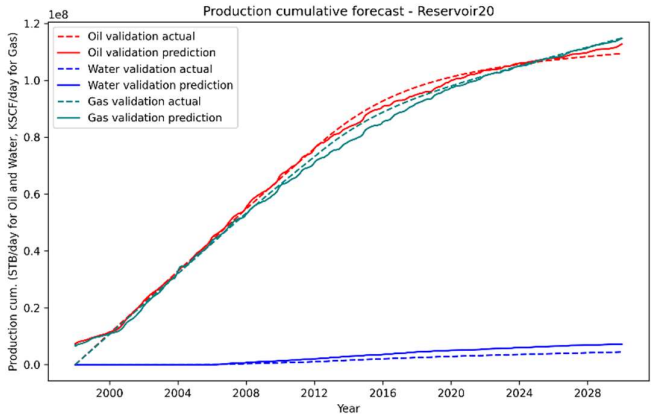
Specific Reservoir Analysis





Reservoir11 and Reservoir79 show larger deviations in both gas and water production predictions, particularly with RMSE values above 0.1, indicating poorer predictive performance for these specific reservoirs.



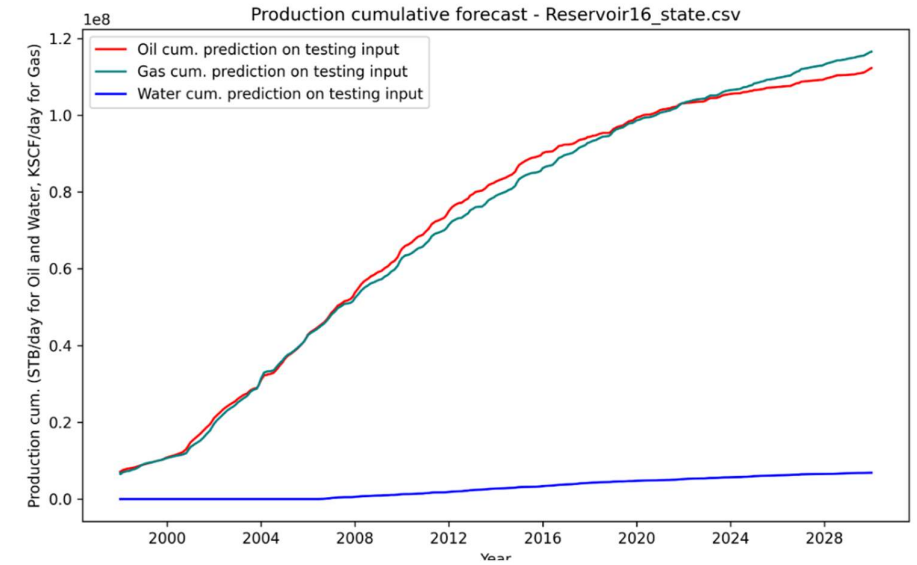


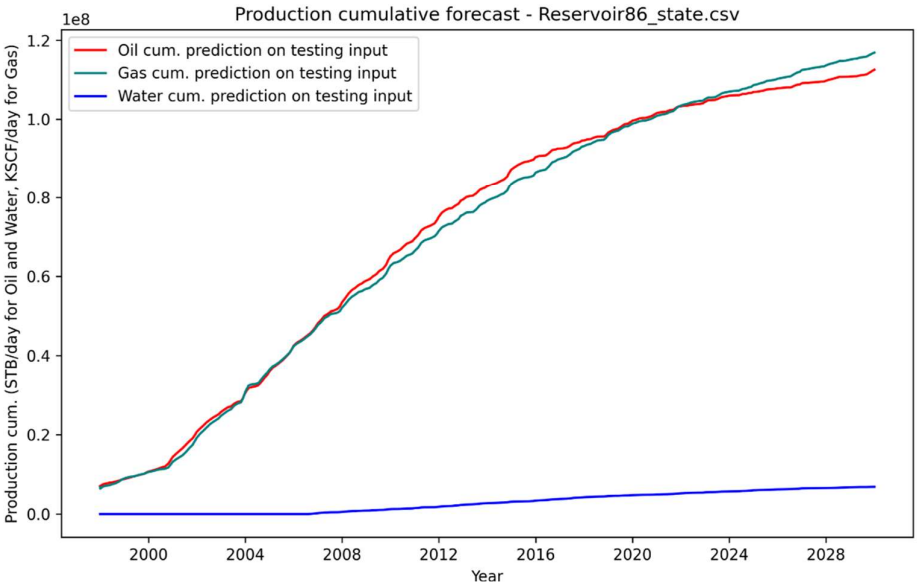
Reservoir5 and Reservoir20 have very low MAE and RMSE values across all output features, indicating strong performance in predicting gas and oil production.

Test Data

The model performance yields a reasonable prediction with only few deviations in validation dataset, the custom loss function worked to ensure that the predicted data remains cumulative and does not fluctuate lower than the previous data. Despite that, the model predictions for the test data show smoother curves with no outliers or unreasonable predictions.

Notably, the first rows are not close to 0. Hence, the presence of zeros in the first rows of temporal data is negligible. Please refer to our `../docs/appendix/` folder to see the plots of all test/validation predictions.





Conclusion

The model is successful (to our knowledge) in terms of its ability to capture the pattern between the spatial properties of a reservoir and the temporal prediction of its cumulative oil, gas, and water production rates. However there are several ways this can be improved.

Recommendations

The most important factor is the dataset. To improve the model, the dataset must be diverse, consist of more parameters, and of higher quality and quantity. As the model grows larger and more complex, the risk of overfitting is higher. This may be curbed via the addition of Dropout layers and the use of L2 regularization. Additionally, the model architecture may be revamped by attempting a physics-informed approach, where the design of the structure and loss function are based on relevant physical laws.