CNN BiLSTM Attention

August 14, 2025

1 Local–Global Hybrid Forecasting: CNN → BiLSTM → Multi-Head Attention

Goal:

Forecast weekly sales while preserving seasonal shape, and select the most reliable approach via a transparent baseline comparison.

Method (2-stage):

1) Learn a **week-of-year seasonal profile** on the train set; compute **residuals** as:

$$\text{seasonal\_mean}(w) \;=\; \frac{1}{N_w} \sum_{i:\,\text{woy}(t_i)=w} y_i, \quad r_i \;=\; y_i - \text{seasonal\_mean}(\text{woy}(t_i)).$$

Note: For calendar week w, average all historical values that occurred in that same week (across years). Here, N_w is the count of such points and $\text{woy}(t)$ returns the week-of-year of timestamp t.

2) Train a **CNN→BiLSTM→Attention** on r_t and **add seasonality back** to predict sales:

$$\hat{y}_t \;=\; \hat{r}_t \;+\; \text{seasonal\_mean}(\text{woy}(t)).$$

Note: Predict residuals with the model, then add the week's seasonal mean to reconstruct the sales forecast.

Validation & Model Selection (time-series discipline):

- **Time-aware** split (no shuffling).
- **Seasonal Naïve** baseline (lag=52):

$$\hat{y}_t^{(\text{SN})} \;=\; y_{t-52}.$$

Note: Use last year's same-week value as this week's forecast.

- **Calibration** of the deep model on validation (remove constant bias or fit a linear map):

$$\text{constant: } \hat{y}^{(\text{cal})} = \hat{y} - b, \qquad \text{linear: } \hat{y}^{(\text{cal})} = a\,\hat{y} + b.$$

Note: Fix systematic bias by shifting predictions by b (constant) or scaling by a and then shifting by b (linear).

- **Blending** (choose α on validation to minimize WMAPE):

$$\hat{y}^{(\mathrm{blend})} \;=\; \alpha\,\hat{y}^{(\mathrm{DL,\,cal})} \;+\; (1-\alpha)\,\hat{y}^{(\mathrm{baseline})}, \quad \alpha \in [0,1].$$

  Note: Combine the calibrated DL forecast and the baseline with a weight α; pick α that yields the lowest validation WMAPE.

- **Champion** = lowest validation WMAPE (ties → choose the simpler approach).

Deliverables:

- Full metrics: **RMSE, MAE, MAPE, WAPE/WMAPE, sMAPE, MASE, ME/MPE,** R^2.
- Plots: Actual vs Pred (**Champion**), Residuals over time, Residual histogram, Actual vs Pred scatter, Rolling 13-week WAPE, Zoom last 52 weeks.
- Excel export: Forecast, Validation view, Validation metrics, Skill vs Baseline, Champion details.

1.1 Environment & Dependencies

This project uses a robust set of Python libraries and configurations to ensure efficient data processing, model development, and visualization. Below is an overview of the key dependencies and setup requirements.

Dependencies:

- **Frameworks**:
  - **TensorFlow/Keras**: Core libraries for building, training, and optimizing deep learning models, including convolutional and recurrent neural networks, with support for mixed precision training to enhance performance.
  - **Keras Tuner**: Utilized for hyperparameter optimization to fine-tune model architectures and training configurations efficiently.
- **Data Handling**:
  - **pandas**: Provides high-performance data structures and analysis tools for efficient manipulation of tabular data.
  - **numpy**: Supports numerical computations and array operations, essential for preprocessing and model input preparation.
- **Visualization**:
  - **matplotlib**: A versatile plotting library for creating customizable visualizations, configured with a high DPI for sharp outputs.
  - **seaborn**: Enhances visualization with aesthetically pleasing and statistically informative graphics, built on top of matplotlib.
- **Export**:
  - **openpyxl**: Enables exporting data and results to Excel for seamless sharing and reporting.

Environment Setup:

- **Virtual Environment**: Create a fresh virtual environment to ensure reproducibility and avoid dependency conflicts. Use tools like `venv` or `virtualenv` to isolate the project's dependencies.
- **TensorFlow Configuration**:

- TensorFlow logging is set to ERROR level to minimize console output (TF_CPP_MIN_LOG_LEVEL='2').
  - OneDNN optimizations are disabled (TF_ENABLE_ONEDNN_OPTS='0') for compatibility.
  - Deterministic operations are enabled where supported (tf.config.experimental.enable_op_determinism()) to ensure reproducible results.
  - A fixed random seed is set (tf.keras.utils.set_random_seed(42)) for consistent model initialization and training.
- **Mixed Precision Training**: Enabled via Keras' mixed_float16 policy to optimize memory usage and accelerate training on compatible hardware.
- **Visualization Settings**:
  - Matplotlib is configured with the fivethirtyeight style and a DPI of 130 for high-quality plots.
  - Seaborn is set to the whitegrid style for clean and professional visualizations.
  - A custom formatter (FuncFormatter) is applied to add comma separators to numerical axis labels for readability.
- **Pandas Configuration**: Floating-point numbers are displayed with two decimal places (pd.set_option('display.float_format', '{:.2f}'.format)) for consistent data presentation.

[29]:
```
import os
import gc
import logging
from datetime import timedelta
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter
from unittest.mock import patch
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import r2_score
import tensorflow as tf
from keras.models import Model
from keras.layers import Input, Conv1D, Add, LayerNormalization, Activation,␣
↪GlobalAveragePooling1D, MultiHeadAttention, Bidirectional, LSTM, Dense,␣
↪Dropout
from keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
from keras.regularizers import l1_l2
from keras.optimizers import Adam
from keras.mixed_precision import Policy, set_global_policy
from keras_tuner import HyperModel, Hyperband
import keras.backend as K

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'
logging.getLogger('tensorflow').setLevel(logging.ERROR)
```

```python
tf.get_logger().setLevel(logging.ERROR)

tf.keras.utils.set_random_seed(42)
try:
    tf.config.experimental.enable_op_determinism()
except Exception:
    pass

policy = Policy('mixed_float16')
set_global_policy(policy)

sns.set_style('whitegrid')
plt.style.use("fivethirtyeight")
plt.rcParams["figure.dpi"] = 130

pd.set_option('display.float_format', '{:.2f}'.format)

def fmt_commas(x, pos):
    try: return '{:,}'.format(int(x))
    except: return x

formatter = FuncFormatter(fmt_commas)
```

[30]:
```python
DATA_PATH  = '/home/linux/Source/VS Code Projects/Advanced Deep Learning␣
 ↪Forecast/CNN BiLSTM Attention/Dataset/Dummy Data.csv'
MODEL_DIR  = '/home/linux/Source/VS Code Projects/Advanced Deep Learning␣
 ↪Forecast/CNN BiLSTM Attention/Model/'
PRED_DIR   = '/home/linux/Source/VS Code Projects/Advanced Deep Learning␣
 ↪Forecast/CNN BiLSTM Attention/Prediction/'
ARTIFACTS_DIR = os.path.join(PRED_DIR, 'Artifacts')

for d in [MODEL_DIR, PRED_DIR, ARTIFACTS_DIR]:
    os.makedirs(d, exist_ok=True)

channel  = 'Carnivore'
category = 'Kraken'

WEEKLY_SEASON_LAG = 52
SEQ_LEN           = 156
VAL_TAIL_FRAC     = 0.20
FORECAST_YEARS    = 6
PROFILE_MODE      = 'trimmed'

print("Config:",
      f"\n  SEQ_LEN={SEQ_LEN}",
      f"\n  VAL_TAIL_FRAC={VAL_TAIL_FRAC}",
      f"\n  WEEKLY_SEASON_LAG={WEEKLY_SEASON_LAG}",
```

```
      f"\n  PROFILE_MODE={PROFILE_MODE}",
      f"\n  FORECAST_YEARS={FORECAST_YEARS}")
```

```
Config:
  SEQ_LEN=156
  VAL_TAIL_FRAC=0.2
  WEEKLY_SEASON_LAG=52
  PROFILE_MODE=trimmed
  FORECAST_YEARS=6
```

1.2 Data Loading & Initial Checks

- Parse dates
- Remove invalid/missing rows
- Sort by date
- Print basic stats for context

[31]:
```
df = pd.read_csv(DATA_PATH, parse_dates=['Date'])
df['Sales'] = pd.to_numeric(df['Sales'], errors='coerce')
df = df.dropna(subset=['Date', 'Sales'])
df = df[df['Sales'] != 0].sort_values('Date').reset_index(drop=True)

start = pd.Timestamp('2019-01-12')
end   = df['Date'].max()
df    = df[(df['Date'] >= start) & (df['Date'] <= end)].reset_index(drop=True)

latest_date = df['Date'].max()
future_dates = pd.date_range(
    start=latest_date + pd.Timedelta(weeks=1),
    periods=FORECAST_YEARS*52,
    freq='W-SAT'
)
forecast_date = (latest_date + pd.Timedelta(days=9)).strftime('%Y-%m-%d')

print(f"\n\033[92mLatest Date: {latest_date.date()} \033[0m\n")
print("Head:\n", df.head(), "\n")
print("Summary stats:\n", df.describe(include='all'), "\n")
print("Missing values:\n", df.isnull().sum(), "\n")
print(f"Total observations: {len(df):,}")
```

```
Latest Date: 2025-06-14

Head:
         Date        Sales
0 2019-01-12 143720043.60
1 2019-01-19 143945495.30
2 2019-01-26 143791091.50
3 2019-02-02 144329039.80
```

```
4 2019-02-09 142930646.80

Summary stats:
                       Date        Sales
count                   336       336.00
mean   2022-03-29 12:00:00 148657439.80
min    2019-01-12 00:00:00 142230477.00
25%    2020-08-20 06:00:00 145367562.95
50%    2022-03-29 12:00:00 148526073.90
75%    2023-11-05 18:00:00 151778546.40
max    2025-06-14 00:00:00 156518947.80
std                     NaN   3602821.28

Missing values:
 Date     0
Sales    0
dtype: int64

Total observations: 336
```

1.3 Seasonal Profile (Week-of-Year)

Three options to estimate the seasonal mean on the **training window**:

1. **Mean**

$$\text{seasonal\_mean}(w) = \frac{1}{N_w} \sum_{i:\,\text{woy}(t_i)=w} y_i.$$

   Note: For calendar week w, average all historical values that occurred in that same week (across years). Here, N_w is the count of such points and $\text{woy}(t)$ returns the week-of-year of timestamp t.

2. **Trimmed Mean** (robust to outliers; trim proportion p from both ends)

$$\text{trimmed\_mean}(w) = \text{mean}\big(\text{trim}_p\{\, y_i : \text{woy}(t_i) = w \,\}\big).$$

   Note: For week w, sort the historical values for that week, drop the lowest $p\%$ and highest $p\%$ (e.g., $p = 10\% \to$ remove bottom/top 10%), then average what remains. This reduces the influence of spikes and dips.

3. **Exponentially Weighted Mean** (recency weighting; half-life h weeks)

$$\text{ew\_mean}(w) = \frac{\sum_{i:\,\text{woy}(t_i)=w} y_i \, \exp\!\left(-\frac{\Delta_i \ln 2}{h}\right)}{\sum_{i:\,\text{woy}(t_i)=w} \exp\!\left(-\frac{\Delta_i \ln 2}{h}\right)}.$$

   Note: For week w, weight newer observations more than older ones. Δ_i is the age (in weeks) of observation i (larger $\Delta_i \to$ older data). With half-life h, the weight halves every h weeks. The denominator normalizes weights so they sum to 1.

Residuals:

$$r_i = y_i - \text{seasonal\_mean}(\text{woy}(t_i)).$$

Note: Subtract the seasonal level for the matching week-of-year from each actual to get the residual (the part not explained by seasonality). If you use the trimmed or EW estimator, plug that in for seasonal_mean(·).

[32]:
```python
df_channel = df[['Date', 'Sales']].groupby('Date', as_index=False)['Sales'].
↪sum()
df_channel['Date'] = pd.to_datetime(df_channel['Date'])
df_channel['woy']  = df_channel['Date'].dt.isocalendar().week.astype(int)

train_len = int(np.ceil(len(df_channel) * 0.80))
train_cutoff_date = df_channel['Date'].iloc[train_len - 1]
train_mask = df_channel['Date'] <= train_cutoff_date
train_only = df_channel.loc[train_mask].copy()

def trimmed_mean(x, p=0.10):
    x = np.sort(x.values)
    k = int(len(x)*p)
    if len(x) > 2*k:
        x = x[k:len(x)-k]
    return x.mean()

def ew_mean(s, halflife_weeks=52):
    n = len(s)
    w = np.exp(-np.arange(n)[::-1] * np.log(2) / halflife_weeks)
    return np.average(s.values, weights=w)

if PROFILE_MODE == 'mean':
    seasonal_profile = train_only.groupby('woy')['Sales'].mean()
elif PROFILE_MODE == 'trimmed':
    seasonal_profile = train_only.groupby('woy')['Sales'].apply(trimmed_mean)
elif PROFILE_MODE == 'ewm':
    seasonal_profile = train_only.groupby('woy').apply(ew_mean)
else:
    raise ValueError("PROFILE_MODE must be one of {'mean','trimmed','ewm'}")

df_channel['seasonal_mean'] = df_channel['woy'].map(seasonal_profile)
df_channel['resid'] = df_channel['Sales'] - df_channel['seasonal_mean']

print("Seasonal profile (week-of-year) summary:")
print(seasonal_profile.describe(), "\n")
print(f"Seasonal amplitude (max-min): {seasonal_profile.max() -␣
↪seasonal_profile.min():,.0f}\n")
print(f"Total observations: {len(df_channel)} | Train cutoff index:␣
↪{train_len}")
```

```
Seasonal profile (week-of-year) summary:
```

```
count          53.00
mean    148805468.14
std       3572545.09
min     143849208.42
25%     145202361.24
50%     148078519.52
75%     152601331.68
max     154470019.08
Name: Sales, dtype: float64

Seasonal amplitude (max-min): 10,620,811

Total observations: 336 | Train cutoff index: 269
```

1.4 Sequences & Time-Aware Validation

Create sliding windows of length $L = 156$ on residuals:

$$\mathbf{X}_t = [\, r_{t-L}, \dots, r_{t-1} \,], \qquad y_t = r_t.$$

Note: Each training example uses the previous L residuals as features ($\mathbf{X}_t$) and the current residual as the target (y_t). Slide the window forward by one time step to generate the next example. (If data are weekly, $L = 156$ 3 years of history.)

Validation: Use the **tail 20% of the training windows** (no shuffling).

Note: Split by time—train on the earlier 80% of windows and validate on the latest 20% of windows to avoid leakage and keep temporal order.

[33]:
```python
def create_sequences(data, sequence_length=SEQ_LEN):
    x, y = [], []
    for i in range(sequence_length, len(data)):
        x.append(data[i-sequence_length:i, 0])
        y.append(data[i, 0])
    return np.array(x), np.array(y)

data_resid = df_channel[['resid']].values

scaler = MinMaxScaler()
train_data = scaler.fit_transform(data_resid[:train_len])
test_data  = scaler.transform(data_resid[train_len - SEQ_LEN:])

x_train, y_train = create_sequences(train_data, sequence_length=SEQ_LEN)
x_test,  y_test  = create_sequences(test_data,  sequence_length=SEQ_LEN)

val_size = int(VAL_TAIL_FRAC * x_train.shape[0])
x_tr, y_tr = x_train[:-val_size], y_train[:-val_size]
x_val, y_val = x_train[-val_size:], y_train[-val_size:]

x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
```

```
x_tr    = x_tr.reshape(x_tr.shape[0], SEQ_LEN, 1)
x_val   = x_val.reshape(x_val.shape[0], SEQ_LEN, 1)
x_test  = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

print(f"x_train total: {x_train.shape}, x_tr: {x_tr.shape}, x_val: {x_val.
 ↪shape}, x_test: {x_test.shape}")
```

```
x_train total: (113, 156, 1), x_tr: (91, 156, 1), x_val: (22, 156, 1), x_test:
(67, 156, 1)
```

1.5 CNN → BiLSTM → Multi-Head Attention

- **CNN** (causal padding, residual blocks) with dilation rates $d \in \{1, 2, 4\}$
- **BiLSTM** to capture long-range context
- **Multi-Head Attention** to focus on the most relevant timesteps
- **Dense** head with (L_1/L_2) regularization

Loss: **MAE** (less smoothing).

[34]:
```
class CNNBiLSTMHyperModel(HyperModel):
    def __init__(self, input_shape):
        self.input_shape = input_shape

    def _residual_block(self, x, filters, k, dilation, hp):
        shortcut = x
        x = Conv1D(filters=filters, kernel_size=k, dilation_rate=dilation,
                   padding='causal', activation=None)(x)
        x = LayerNormalization()(x)
        x = Activation('relu')(x)
        x = Dropout(hp.Float('cnn_dropout', 0.1, 0.4, step=0.1))(x)
        if shortcut.shape[-1] != x.shape[-1]:
            shortcut = Conv1D(filters=filters, kernel_size=1,␣
 ↪padding='same')(shortcut)
        return Add()([x, shortcut])

    def build(self, hp):
        inp = Input(shape=self.input_shape)
        filters = hp.Int('filters', 16, 64, step=16)
        k = hp.Choice('kernel_size', [3,5,7])

        x = inp
        for d in [1, 2, 4]:
            x = self._residual_block(x, filters, k, dilation=d, hp=hp)

        units = hp.Int('lstm_units', 100, 300, step=50)
        x = Bidirectional(LSTM(units,
```

```
                              return_sequences=True,
                              dropout=hp.Float('lstm_dropout', 0.1, 0.4,␣
↪step=0.1),
                              recurrent_dropout=hp.Float('recurrent_dropout',␣
↪0.0, 0.2, step=0.1)))(x)

        heads   = hp.Choice('mha_heads', [2,4,8])
        key_dim = hp.Choice('mha_key_dim', [16,32,64])
        x = MultiHeadAttention(num_heads=heads, key_dim=key_dim,
                              dropout=hp.Float('mha_dropout', 0.0, 0.2, step=0.
↪1))(x, x)
        x = LayerNormalization()(x)

        if hp.Boolean('second_bilstm'):
            x = Bidirectional(LSTM(units, return_sequences=False,
                                  dropout=hp.Float('lstm2_dropout', 0.1, 0.3,␣
↪step=0.1)))(x)
        else:
            x = GlobalAveragePooling1D()(x)

        x = Dense(hp.Int('dense_units', 32, 128, step=32), activation='relu',
                  kernel_regularizer=l1_l2(
                      l1=hp.Float('l1_reg', 1e-5, 1e-3, sampling='LOG'),
                      l2=hp.Float('l2_reg', 1e-5, 1e-3, sampling='LOG')))(x)
        x = Dropout(hp.Float('dense_dropout', 0.2, 0.4, step=0.1))(x)
        out = Dense(1)(x)

        model = Model(inp, out)
        model.compile(optimizer=Adam(hp.Float('learning_rate', 1e-4, 3e-3,␣
↪sampling='LOG')),
                      loss='mae', metrics=['mae'])
        return model
```

[35]:
```
early_stopping = EarlyStopping(monitor='val_loss', patience=15,␣
↪restore_best_weights=True)
lr_reduction   = ReduceLROnPlateau(monitor='val_loss', factor=0.3, patience=7)

tuner = Hyperband(
    CNNBiLSTMHyperModel(input_shape=(SEQ_LEN, 1)),
    objective='val_loss',
    max_epochs=150,
    factor=3,
    directory=MODEL_DIR,
    project_name=f"CNNBiLSTM - {channel} - {category}"
)

ckpt_path = os.path.join(MODEL_DIR, 'best_cnn_bilstm.keras')
```

```python
checkpoint = ModelCheckpoint(ckpt_path, monitor='val_loss', save_best_only=True)

tuner.search(
    x_tr, y_tr,
    epochs=150,
    validation_data=(x_val, y_val),
    callbacks=[early_stopping, lr_reduction, checkpoint],
    shuffle=False
)

best_hp    = tuner.get_best_hyperparameters(1)[0]
best_model = tuner.get_best_models(1)[0]

print("Best Hyperparameters:")
for k, v in best_hp.values.items():
    print(f"  - {k}: {v}")

best_model.compile(optimizer=Adam(1e-3), loss='mae', metrics=['mae'])
print(f"\nParameters (best model): {best_model.count_params():,}")
```

```
Reloading Tuner from /home/linux/Source/VS Code Projects/Advanced Deep Learning
Forecast/CNN BiLSTM Attention/Model/CNNBiLSTM - Carnivore - Kraken/tuner0.json
Best Hyperparameters:
  - filters: 64
  - kernel_size: 5
  - cnn_dropout: 0.2
  - lstm_units: 100
  - lstm_dropout: 0.2
  - recurrent_dropout: 0.2
  - mha_heads: 2
  - mha_key_dim: 32
  - mha_dropout: 0.2
  - second_bilstm: True
  - dense_units: 96
  - l1_reg: 1.173607447336231e-05
  - l2_reg: 1.1858185678151422e-05
  - dense_dropout: 0.2
  - learning_rate: 0.0017752767384601404
  - lstm2_dropout: 0.1
  - tuner/epochs: 150
  - tuner/initial_epoch: 50
  - tuner/bracket: 4
  - tuner/round: 4
  - tuner/trial_id: 0143

Parameters (best model): 486,169

/home/linux/miniforge3/envs/dl/lib/python3.11/site-
```

```
packages/keras/src/saving/saving_lib.py:797: UserWarning: Skipping variable
loading for optimizer 'adam', because it has 2 variables whereas the saved
optimizer has 82 variables.
  saveable.load_own_variables(weights_store.get(inner_path))
```

1.6 Metrics

1.6.1 RMSE

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Note: Root-mean-square error penalizes big misses more (squares the errors). Good when care about large spikes.

1.6.2 MAE

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Note: Average absolute miss in the original units. More robust to outliers than RMSE.

1.6.3 MAPE %

$$\text{MAPE}(\%) = \frac{100}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{|y_i|}$$

Note: Average relative error. Undefined when any $y_i = 0$; in practice use a small ε or report only where $|y_i| > \varepsilon$.

1.6.4 sMAPE %

$$\text{sMAPE}(\%) = \frac{100}{n}\sum_{i=1}^{n}\frac{2|y_i - \hat{y}_i|}{|y_i| + |\hat{y}_i|}$$

Note: Symmetric version that caps influence when y_i is near 0. Range is $[0, 200]$.

1.6.5 WAPE / WMAPE %

$$\text{WAPE/WMAPE}(\%) = 100 \cdot \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{\sum_{i=1}^{n}|y_i|}$$

Note: "Total miss divided by total actuals." Equivalent to MAE/$\bar{y}$ when data are nonnegative and evenly weighted.

1.6.6 R^2

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Note: Fraction of variance explained vs. predicting the mean $\bar{y}$. Can be negative on validation/test if the model is worse than using $\bar{y}$.

1.6.7 MASE (relative to seasonal naive)

$$\text{MASE} = \frac{\text{MAE}_{\text{model}}}{\frac{1}{N-m}\sum_{t=m+1}^{N}|y_t - y_{t-m}|}, \quad m = 52$$

Note: Compares model MAE to the in-sample MAE of a seasonal naïve with period m (here weekly seasonality $m = 52$). Interpretable across series: MASE < 1 beats seasonal naïve; > 1 is worse.

1.6.8 Seasonal Naïve (lag = 52)

$$\hat{y}_t^{(\text{SN})} = y_{t-52}$$

Note: Baseline that copies last year's same week.

1.6.9 Skill vs. baseline (e.g., WMAPE)

$$\text{Improvement}(\%) = \frac{\text{WMAPE}_{\text{baseline}} - \text{WMAPE}_{\text{model}}}{\text{WMAPE}_{\text{baseline}}} \times 100$$

Note: Positive values mean the model improves on the baseline; 20% means "20% lower WMAPE than the baseline."

[36]:
```
def _safe_div(a, b):
    return np.divide(a, b, out=np.zeros_like(a, dtype=float), where=(b!=0))

def compute_metrics(y_true, y_pred, train_series_for_mase=None,␣
 ↪m_season=WEEKLY_SEASON_LAG):
    err = y_pred - y_true
    abs_err = np.abs(err)
    mae  = abs_err.mean()
    rmse = np.sqrt(np.mean(err**2))
    mape = np.mean(_safe_div(abs_err, np.abs(y_true))) * 100
    smape = np.mean(2.0 * abs_err / (np.abs(y_true) + np.abs(y_pred))) * 100
    wape = abs_err.sum() / np.abs(y_true).sum() * 100
    me   = err.mean()
    mpe  = np.mean(_safe_div(err, np.where(y_true==0, np.nan, y_true))) * 100
    r2   = r2_score(y_true, y_pred)
    mase = np.nan
    if train_series_for_mase is not None and len(train_series_for_mase) >␣
 ↪m_season:
        naive_diffs = np.abs(train_series_for_mase[m_season:] -␣
 ↪train_series_for_mase[:-m_season])
        denom = naive_diffs.mean()
        mase = mae / denom if denom != 0 else np.nan
    return {"RMSE": rmse, "MAE": mae, "MAPE_%": mape, "sMAPE_%": smape,
            "WAPE_or_WMAPE_%": wape, "MASE": mase, "ME": me, "MPE_%": mpe, "R2":
 ↪ r2}
```

```python
def seasonal_naive_on_dates(df_date_sales, dates, lag_weeks=WEEKLY_SEASON_LAG,␣
↪fallback_seasonal=None):
    """Return baseline array for given dates: y(t-52). If missing, fall back to␣
↪seasonal mean for that WOY."""
    hist_map = dict(zip(df_date_sales['Date'], df_date_sales['Sales']))
    res = []
    for d in dates:
        prev = hist_map.get(d - pd.Timedelta(weeks=lag_weeks))
        if pd.isna(prev) or prev is None:
            if fallback_seasonal is None:
                res.append(np.nan)
            else:
                w = int(pd.Timestamp(d).isocalendar().week)
                res.append(float(fallback_seasonal.get(w, np.nan)))
        else:
            res.append(float(prev))
    return np.array(res, dtype=float)

def best_alpha(y_true, y_dl, y_base, step=0.02, train_series=None):
    """Grid-search alpha in [0,1] to minimize WMAPE on validation."""
    best = (0.0, 1e9)
    for alpha in np.arange(0, 1+1e-9, step):
        y_hat = alpha*y_dl + (1-alpha)*y_base
        wape = compute_metrics(y_true, y_hat,␣
↪train_series_for_mase=train_series)['WAPE_or_WMAPE_%']
        if wape < best[1]:
            best = (alpha, wape)
    return best
```

[37]:

```python
pred_resid_scaled = best_model.predict(x_test, verbose=0)
pred_resid = scaler.inverse_transform(pred_resid_scaled.reshape(-1,1)).flatten()

valid_df = df_channel[train_len:].copy()
valid_df['Predictions_DL'] = np.nan
idx = valid_df.index[-len(pred_resid):]
sales_pred_dl = pred_resid + valid_df.loc[idx, 'seasonal_mean'].values
valid_df.loc[idx, 'Predictions_DL'] = sales_pred_dl
sales_true = valid_df.loc[idx, 'Sales'].values

baseline = seasonal_naive_on_dates(df_channel[['Date','Sales']], valid_df.
↪loc[idx, 'Date'], fallback_seasonal=seasonal_profile)
mask = ~np.isnan(baseline)
y_true_masked = sales_true[mask]
y_base_masked = baseline[mask]
y_dl_masked   = sales_pred_dl[mask]

train_series = df_channel.loc[:train_len-1, 'Sales'].values
```

```python
m_raw  = compute_metrics(y_true_masked, y_dl_masked, train_series_for_mase=train_series)
m_base = compute_metrics(y_true_masked, y_base_masked, train_series_for_mase=train_series)

bias = (y_dl_masked - y_true_masked).mean()
y_dl_const = y_dl_masked - bias

a_lin, b_lin = np.polyfit(y_dl_masked, y_true_masked, deg=1)
y_dl_lin = a_lin * y_dl_masked + b_lin

m_const = compute_metrics(y_true_masked, y_dl_const, train_series_for_mase=train_series)
m_lin   = compute_metrics(y_true_masked, y_dl_lin,   train_series_for_mase=train_series)

dl_candidates = {'DL_raw': (y_dl_masked, m_raw),
                 'DL_const_cal': (y_dl_const, m_const),
                 'DL_linear_cal': (y_dl_lin, m_lin)}

best_dl_name, (best_dl_series, best_dl_metrics) = min(
    dl_candidates.items(), key=lambda kv: kv[1][1]['WAPE_or_WMAPE_%']
)

print("Deep model calibration choice:", best_dl_name)
print("DL (raw) WMAPE:", m_raw['WAPE_or_WMAPE_%'])
print("DL (const-cal) WMAPE:", m_const['WAPE_or_WMAPE_%'])
print("DL (linear-cal) WMAPE:", m_lin['WAPE_or_WMAPE_%'])
print("Baseline WMAPE:", m_base['WAPE_or_WMAPE_%'])

metrics_df = pd.DataFrame({
    'DL_raw':      m_raw,
    'DL_const_cal':m_const,
    'DL_linear_cal':m_lin,
    'Baseline_SN': m_base
}).T

metrics_df = metrics_df.apply(lambda s: s.round(4) if np.issubdtype(s.dtype, np.number) else s)
display(metrics_df)

def pct_improve(model_val, base_val):
    return (base_val - model_val) / base_val * 100 if base_val not in (0, None, np.nan) else np.nan

skill = {
```

```
    'WAPE/WMAPE improvement vs baseline (%)':␣
 ↪pct_improve(best_dl_metrics['WAPE_or_WMAPE_%'], m_base['WAPE_or_WMAPE_%']),
    'sMAPE improvement vs baseline (%)':      ␣
 ↪pct_improve(best_dl_metrics['sMAPE_%'],           m_base['sMAPE_%']),
    'MAE improvement vs baseline (%)':        ␣
 ↪pct_improve(best_dl_metrics['MAE'],               m_base['MAE']),
    'RMSE improvement vs baseline (%)':       ␣
 ↪pct_improve(best_dl_metrics['RMSE'],              m_base['RMSE']),
}
print("\nRelative improvement of best-calibrated DL vs Baseline:")
for k, v in skill.items():
    print(f"  • {k}: {v:.2f}")

calibration = {'type': best_dl_name, 'bias': float(bias), 'a': float(a_lin),␣
 ↪'b': float(b_lin)}
calibration
```

```
2025-08-13 15:52:47.853344: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
```

```
Deep model calibration choice: DL_linear_cal
DL (raw) WMAPE: 0.6082768720341996
DL (const-cal) WMAPE: 0.4538482496772074
DL (linear-cal) WMAPE: 0.3791760914187811
Baseline WMAPE: 0.5274840077547783

                    RMSE       MAE  MAPE_%  sMAPE_%  WAPE_or_WMAPE_%  MASE  \
DL_raw        1105332.03 902033.71    0.60     0.60             0.61  0.60
DL_const_cal   887177.56 673026.44    0.45     0.45             0.45  0.45
DL_linear_cal  721561.75 562292.65    0.38     0.38             0.38  0.37
Baseline_SN    944726.35 782223.32    0.53     0.53             0.53  0.52

                     ME  MPE_%   R2
DL_raw        659298.77   0.44 0.85
DL_const_cal      -0.00  -0.00 0.90
DL_linear_cal     -0.00   0.00 0.94
Baseline_SN   314281.11   0.21 0.89


Relative improvement of best-calibrated DL vs Baseline:
```

- WAPE/WMAPE improvement vs baseline (%): 28.12
- sMAPE improvement vs baseline (%): 27.94
- MAE improvement vs baseline (%): 28.12
- RMSE improvement vs baseline (%): 23.62

[37]:
```
{'type': 'DL_linear_cal',
 'bias': 659298.7746917792,
 'a': 0.8416684305079865,
 'b': 22924596.376746863}
```

[38]:
```python
alpha_opt, wape_opt = best_alpha(y_true_masked, best_dl_series, y_base_masked,␣
 ↪step=0.02, train_series=train_series)
y_blend = alpha_opt*best_dl_series + (1-alpha_opt)*y_base_masked
m_blend = compute_metrics(y_true_masked, y_blend,␣
 ↪train_series_for_mase=train_series)

print(f"\nOptimal blend alpha: {alpha_opt:.2f}")
print("Blend WMAPE:", m_blend['WAPE_or_WMAPE_%'])

candidates = {
    'Baseline_SN':    (y_base_masked, m_base,   {'alpha': 0.0}),
    best_dl_name:     (best_dl_series, best_dl_metrics, {'alpha': 1.0,␣
 ↪**calibration}),
    'Blend':          (y_blend, m_blend, {'alpha': alpha_opt, **calibration})
}
champion_name, (champ_series, champ_metrics, champ_info) = min(
    candidates.items(), key=lambda kv: kv[1][1]['WAPE_or_WMAPE_%']
)

print("\n=== Champion (validation) ===")
print("Name:", champion_name)
for k, v in champ_metrics.items():
    print(f"  {k}: {v:.4f}")
print("Details:", champ_info)

valid_df['Predictions_Champion'] = np.nan
valid_df.loc[idx[mask], 'Predictions_Champion'] = champ_series
display(valid_df[['Date','Sales','Predictions_Champion']].tail(10))
```

```
Optimal blend alpha: 1.00
Blend WMAPE: 0.3791760914187811

=== Champion (validation) ===
Name: DL_linear_cal
  RMSE: 721561.7469
  MAE: 562292.6527
  MAPE_%: 0.3786
```

```
  sMAPE_%: 0.3788
  WAPE_or_WMAPE_%: 0.3792
  MASE: 0.3740
  ME: -0.0000
  MPE_%: 0.0024
  R2: 0.9353
Details: {'alpha': 1.0, 'type': 'DL_linear_cal', 'bias': 659298.7746917792, 'a':
0.8416684305079865, 'b': 22924596.376746863}
```

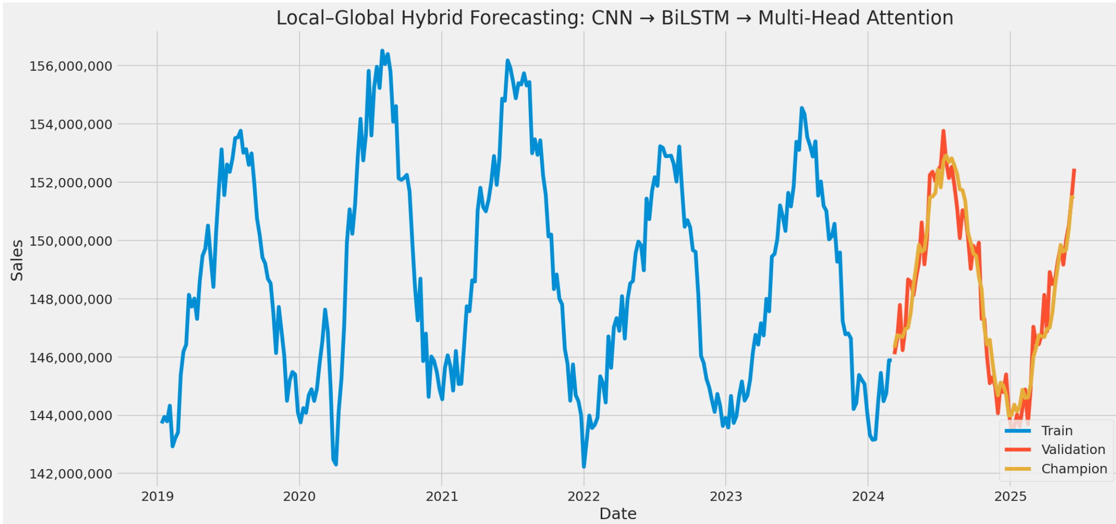
```
          Date        Sales  Predictions_Champion
326 2025-04-12 148917630.80          147009011.86
327 2025-04-19 148502266.80          147540911.55
328 2025-04-26 148680861.60          148470607.09
329 2025-05-03 149312608.70          149125769.03
330 2025-05-10 149796640.00          149847900.11
331 2025-05-17 149169487.40          149612742.21
332 2025-05-24 150001083.40          149689357.17
333 2025-05-31 150519921.00          150395772.86
334 2025-06-07 151409087.10          151485048.14
335 2025-06-14 152464663.30          151507803.63
```

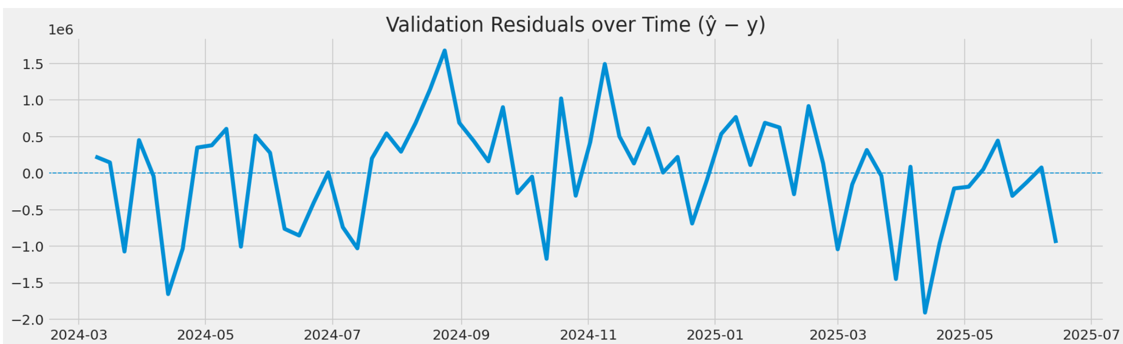
1.7 Visual Diagnostics

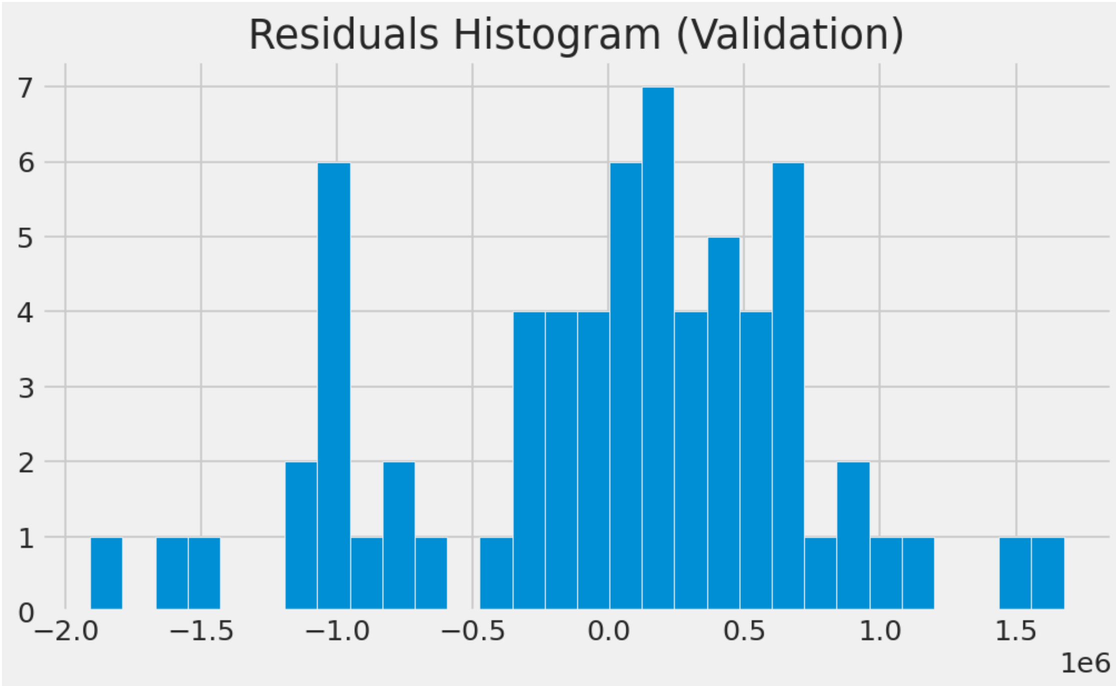
- Actual vs Pred (**Champion**)
- Residuals over time
- Residual histogram
- Actual vs Pred scatter
- Rolling 13-week WAPE
- Zoom last 52 weeks

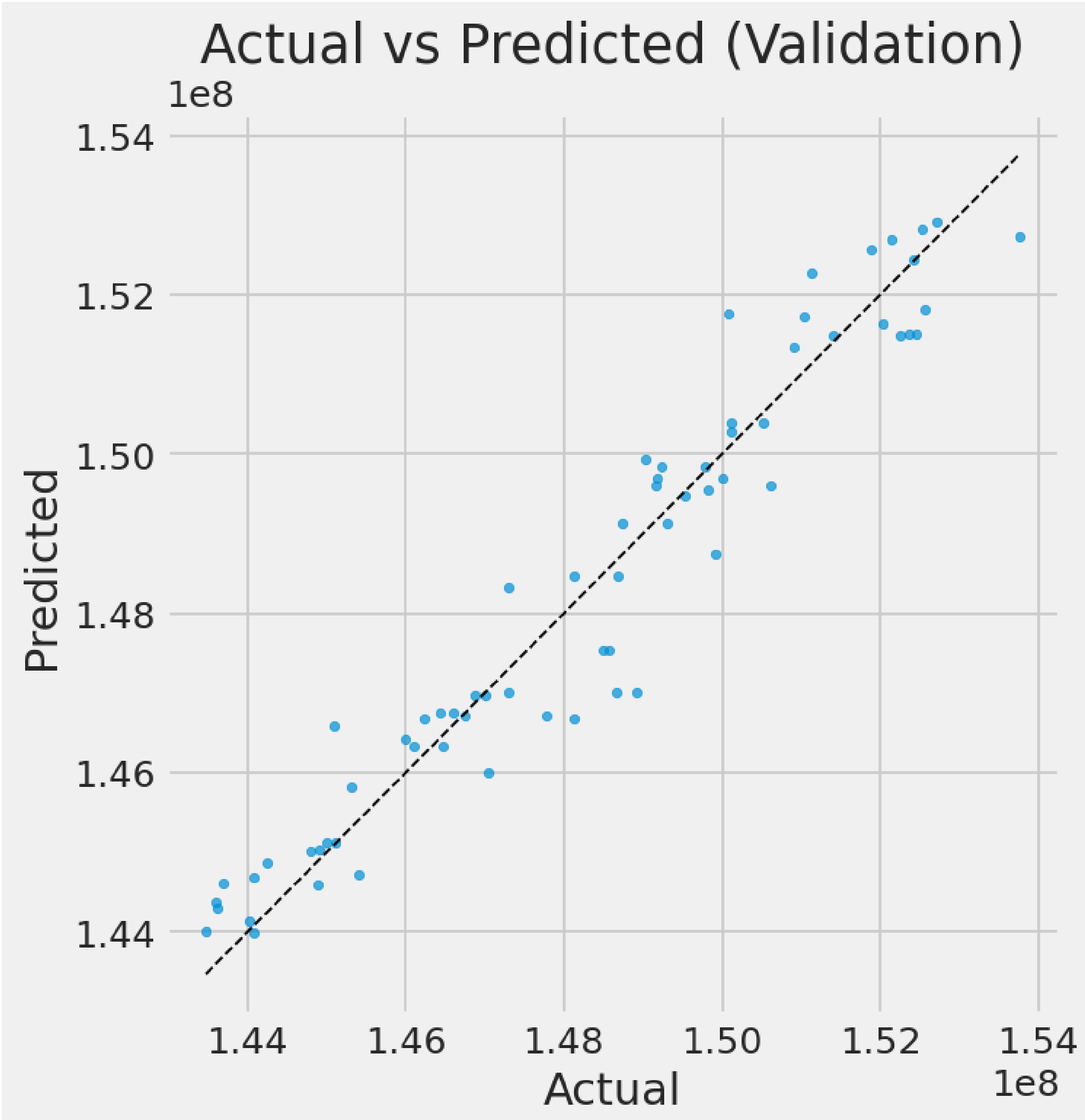
[39]:
```
def plot_actual_vs_pred(train_df, valid_df, title_suffix=''):
    plt.figure(figsize=(16,8))
    plt.title(f'Local-Global Hybrid Forecasting: CNN → BiLSTM → Multi-Head␣
 ↪Attention {title_suffix}')
    plt.xlabel('Date'); plt.ylabel('Sales')
    plt.plot(train_df['Date'], train_df['Sales'], label='Train')
    plt.plot(valid_df['Date'], valid_df['Sales'], label='Validation')
    plt.plot(valid_df['Date'], valid_df['Predictions_Champion'],␣
 ↪label='Champion')
    plt.legend(loc='lower right'); plt.tight_layout(); plt.gca().yaxis.
 ↪set_major_formatter(formatter)
    plt.show()


def plot_residuals_time(y_true, y_pred, dates):
    resid = y_pred - y_true
    plt.figure(figsize=(16,5))
    plt.title('Validation Residuals over Time (ŷ - y)')
```

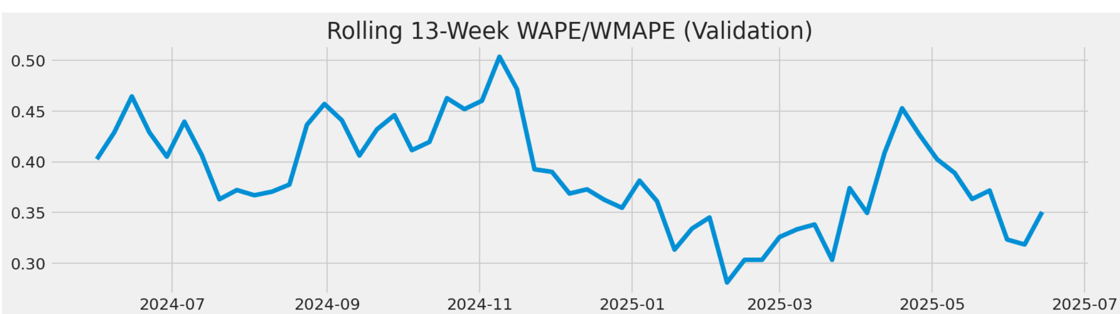
```python
    plt.plot(dates, resid)
    plt.axhline(0, linestyle='--', linewidth=1)
    plt.tight_layout(); plt.show()

def plot_residual_hist(y_true, y_pred):
    resid = y_pred - y_true
    plt.figure(figsize=(8,5))
    plt.title('Residuals Histogram (Validation)')
    plt.hist(resid, bins=30)
    plt.tight_layout(); plt.show()

def plot_scatter(y_true, y_pred):
    lo, hi = min(y_true.min(), y_pred.min()), max(y_true.max(), y_pred.max())
    plt.figure(figsize=(6,6))
    plt.title('Actual vs Predicted (Validation)')
    plt.scatter(y_true, y_pred, s=12, alpha=0.7)
    plt.plot([lo,hi], [lo,hi], 'k--', linewidth=1)
    plt.xlabel('Actual'); plt.ylabel('Predicted')
    plt.tight_layout(); plt.show()

def plot_rolling_wape(valid_df, window=13):
    df_tmp = valid_df.dropna(subset=['Predictions_Champion']).copy()
    df_tmp['abs_err'] = (df_tmp['Predictions_Champion'] - df_tmp['Sales']).abs()
    df_tmp['roll_wape'] = (df_tmp['abs_err'].rolling(window).sum()
                          / df_tmp['Sales'].abs().rolling(window).sum()) * 100
    plt.figure(figsize=(14,4))
    plt.title(f'Rolling {window}-Week WAPE/WMAPE (Validation)')
    plt.plot(df_tmp['Date'], df_tmp['roll_wape'])
    plt.tight_layout(); plt.show()

def plot_zoom_last52(valid_df):
    dfz = valid_df.tail(52).copy()
    plt.figure(figsize=(16,6))
    plt.title('Zoom: Last 52 Validation Weeks')
    plt.plot(dfz['Date'], dfz['Sales'], label='Validation')
    plt.plot(dfz['Date'], dfz['Predictions_Champion'], label='Champion')
    plt.legend(); plt.tight_layout(); plt.show()

train_df = df_channel[:train_len]
plot_actual_vs_pred(train_df, valid_df)
y_true_plot = valid_df.loc[idx[mask], 'Sales'].values
y_pred_plot = valid_df.loc[idx[mask], 'Predictions_Champion'].values
plot_residuals_time(y_true_plot, y_pred_plot, valid_df.loc[idx[mask], 'Date'])
plot_residual_hist(y_true_plot, y_pred_plot)
plot_scatter(y_true_plot, y_pred_plot)
plot_rolling_wape(valid_df, window=13)
plot_zoom_last52(valid_df)
```

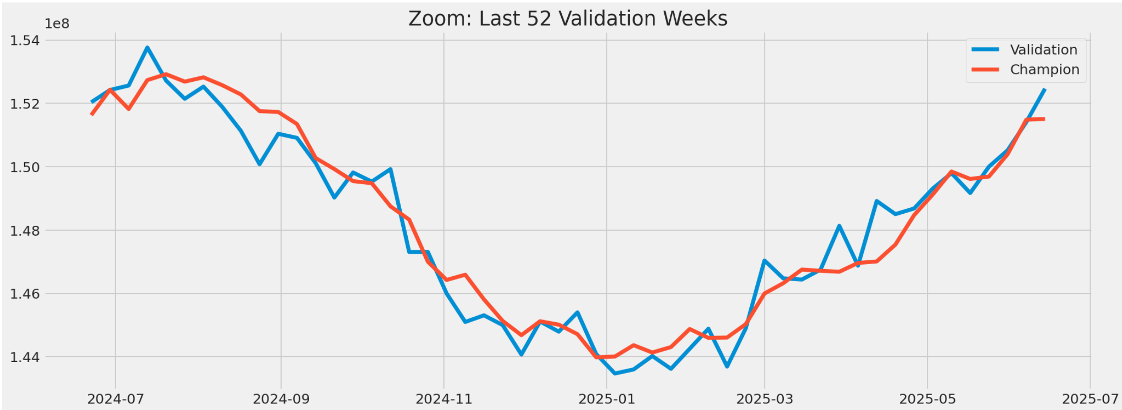

Local–Global Hybrid Forecasting: CNN → BiLSTM → Multi-Head Attention
Sales
156,000,000
154,000,000
152,000,000
150,000,000
148,000,000
146,000,000
144,000,000
142,000,000
2019
2020
2021
2022
2023
2024
2025
Date
Train
Validation
Champion


Validation Residuals over Time (ŷ − y)
1e6
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
2024-03
2024-05
2024-07
2024-09
2024-11
2025-01
2025-03
2025-05
2025-07


Residuals Histogram (Validation)
7
6
5
4
3
2
1
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
1e6


Actual vs Predicted (Validation)
1e8
1.54
1.52
1.50
1.48
1.46
1.44
Predicted
1.44
1.46
1.48
1.50
1.52
1.54
Actual
1e8


Rolling 13-Week WAPE/WMAPE (Validation)
0.50
0.45
0.40
0.35
0.30
2024-07
2024-09
2024-11
2025-01
2025-03
2025-05
2025-07



1.8 Future Forecasting

Residual rollout in scaled space with seasonality add-back:

$$\hat{r}_{t+1} = f(r_{t-L+1:t}), \qquad \hat{y}^{(\mathrm{DL})}_{t+1} = \hat{r}_{t+1} + \mathrm{seasonal\_mean}(\mathrm{woy}(t+1)).$$

Note: Feed the last L residuals into the model f to predict the next residual $\hat{r}_{t+1}$. Then add the seasonal mean for week-of-year $\mathrm{woy}(t+1)$ to reconstruct the sales forecast $\hat{y}^{(\mathrm{DL})}_{t+1}$. If you trained on scaled residuals, apply the inverse transform to $\hat{r}_{t+1}$ before adding back seasonality.

Apply the same calibration found on validation:

- **Constant calibration**

$$\hat{y}^{(\mathrm{cal})} = \hat{y} - b$$

  Note: Shifts forecasts by a constant bias b learned on the validation set.

- **Linear calibration**

$$\hat{y}^{(\mathrm{cal})} = a\,\hat{y} + b$$

  Note: Scales by a and shifts by b to remove proportional and constant bias.

Baseline (lag = 52) for future week t:

$$\hat{y}^{(\mathrm{SN})}_t = y_{t-52}$$

Note: Copy last year's same-week value. If y_{t-52} is unavailable (e.g., new series), fall back to $\mathrm{seasonal\_mean}(\mathrm{woy}(t))$.

Blend (if chosen):

$$\hat{y}^{(\mathrm{blend})} = \alpha\,\hat{y}^{(\mathrm{DL,\,cal})} + (1-\alpha)\,\hat{y}^{(\mathrm{SN})}$$

Note: Weighted average of the calibrated deep-learning forecast and the seasonal-naïve baseline. Choose $\alpha \in [0, 1]$ on validation to minimize WMAPE.

[40]:
```
last_sequence = scaler.transform(df_channel[['resid']].values)[train_len -
↪SEQ_LEN:]
future_steps = FORECAST_YEARS * 52
```

```
future_scaled = []
for _ in range(future_steps):
    seq = last_sequence.reshape(1, last_sequence.shape[0], 1)
    next_scaled = best_model.predict(seq, verbose=0)
    future_scaled.append(next_scaled.flatten()[0])
    last_sequence = np.append(last_sequence[1:], next_scaled).reshape(-1, 1)

future_resid = scaler.inverse_transform(np.array(future_scaled).reshape(-1,1)).
 ↪flatten()

future_woy = pd.Series(future_dates).dt.isocalendar().week.astype(int).values
seasonal_mean_avg = seasonal_profile.mean()
future_seasonal = np.array([seasonal_profile.get(int(w), seasonal_mean_avg) for␣
 ↪w in future_woy])
y_future_dl = future_resid + future_seasonal

def apply_calibration(y, calib):
    if calib['type'] == 'DL_const_cal':
        return y - calib['bias']
    elif calib['type'] == 'DL_linear_cal':
        return calib['a']*y + calib['b']
    else:
        return y

y_future_dl_cal = apply_calibration(y_future_dl, calibration)

future_baseline = seasonal_naive_on_dates(df_channel[['Date','Sales']],␣
 ↪future_dates, fallback_seasonal=seasonal_profile)

if champion_name == 'Baseline_SN':
    y_future_final = future_baseline
elif champion_name == 'Blend':
    alpha = champ_info.get('alpha', 0.5)
    y_future_final = alpha * y_future_dl_cal + (1 - alpha) * future_baseline
else:
    y_future_final = y_future_dl_cal

final_predictions = pd.DataFrame({
    'Date': future_dates.strftime('%Y-%m-%d'),
    'Forecast Date': forecast_date,
    'Channel': f"{channel} - {category} (Validation WMAPE:␣
 ↪{champ_metrics['WAPE_or_WMAPE_%']:.2f}%)",
    'Prediction': y_future_final
})
display(final_predictions.head(12))
```

```
2025-08-13 15:52:53.745226: E tensorflow/core/framework/node_def_util.cc:680]
```

```
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:52:58.712807: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:03.677722: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:08.716855: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:13.685269: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
```

```
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:18.700298: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:23.652290: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:29.705869: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:34.663048: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
```

```
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:39.804701: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:44.784039: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:49.715054: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:53:54.664762: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:00.823560: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
```

```
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:05.702853: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:10.774191: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:15.778926: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:21.096299: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
```

```
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:26.114993: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:32.237694: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:37.371651: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:42.239928: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
```

```
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:47.144356: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:52.092075: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:54:57.331840: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:55:02.476642: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2025-08-13 15:55:08.768945: E tensorflow/core/framework/node_def_util.cc:680]
NodeDef mentions attribute use_unbounded_threadpool which is not in the op
definition: Op<name=MapDataset; signature=input_dataset:variant,
```

```
other_arguments: -> handle:variant; attr=f:func;
attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1;
attr=use_inter_op_parallelism:bool,default=true;
attr=preserve_cardinality:bool,default=false;
attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This
may be expected if your graph generating binary is newer  than this binary.
Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}

          Date Forecast Date                                         Channel  \
0   2025-06-21    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
1   2025-06-28    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
2   2025-07-05    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
3   2025-07-12    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
4   2025-07-19    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
5   2025-07-26    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
6   2025-08-02    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
7   2025-08-09    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
8   2025-08-16    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
9   2025-08-23    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
10  2025-08-30    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)
11  2025-09-06    2025-06-23  Carnivore - Kraken (Validation WMAPE: 0.38%)

     Prediction
0   151628830.88
1   152431215.88
2   151822294.83
3   152734959.71
4   152920251.85
5   152686571.30
6   152821673.56
7   152575375.25
8   152281441.40
9   151755060.13
10  151727460.51
11  151347423.31
```

[41]:

```
xlsx_name = f'{channel} - {category} - {df["Date"].max().date()}.xlsx'
xlsx_path = os.path.join(PRED_DIR, xlsx_name)

valid_export = valid_df[['Date','Sales','Predictions_Champion']].copy()
valid_export['Error']    = valid_export['Predictions_Champion'] -␣
 ↪valid_export['Sales']
valid_export['AbsError'] = valid_export['Error'].abs()
valid_export['APE_%']    = _safe_div(valid_export['AbsError'],␣
 ↪valid_export['Sales'].abs()) * 100

metrics_table = pd.DataFrame({
```

```
    'Champion': champ_metrics
}).T
baseline_table = pd.DataFrame({'Baseline_SN': m_base}).T
dl_table = pd.DataFrame({
    'DL_raw':       m_raw,
    'DL_const_cal': m_const,
    'DL_linear_cal':m_lin
}).T

for t in [metrics_table, baseline_table, dl_table]:
    t[:] = t.apply(lambda s: s.round(4) if np.issubdtype(s.dtype, np.number)␣
 ↪else s)

skill_final = {
    'WAPE/WMAPE improvement vs baseline (%)': (m_base['WAPE_or_WMAPE_%'] -␣
 ↪champ_metrics['WAPE_or_WMAPE_%']) / m_base['WAPE_or_WMAPE_%'] * 100,
    'sMAPE improvement vs baseline (%)':      (m_base['sMAPE_%']         -␣
 ↪champ_metrics['sMAPE_%'])           / m_base['sMAPE_%']           * 100,
    'MAE improvement vs baseline (%)':        (m_base['MAE']             -␣
 ↪champ_metrics['MAE'])               / m_base['MAE']               * 100,
    'RMSE improvement vs baseline (%)':       (m_base['RMSE']            -␣
 ↪champ_metrics['RMSE'])              / m_base['RMSE']              * 100,
}
skill_df = pd.DataFrame(skill_final, index=['Champion_vs_Baseline']).T.round(2)

with pd.ExcelWriter(xlsx_path, engine='openpyxl') as writer:
    final_predictions.to_excel(writer, index=False, sheet_name='Forecast')
    valid_export.to_excel(writer, index=False, sheet_name='Validation_View')
    metrics_table.to_excel(writer, sheet_name='Champion_Metrics')
    baseline_table.to_excel(writer, sheet_name='Baseline_Metrics')
    dl_table.to_excel(writer, sheet_name='DL_Metrics')
    skill_df.to_excel(writer, sheet_name='Skill_vs_Baseline')
    pd.DataFrame([{
        'Champion': champion_name, **champ_info
    }]).to_excel(writer, index=False, sheet_name='Champion_Details')

print(f'\033[92mExcel created: {xlsx_path}\033[0m')

SAVE_FIGS = True
if SAVE_FIGS:
    os.makedirs(ARTIFACTS_DIR, exist_ok=True)

    # Disable display just for this block (plt.show becomes a no-op)
    with patch.object(plt, 'show', lambda *a, **k: None):
        plot_actual_vs_pred(train_df, valid_df, title_suffix='(Saved)')
        plt.savefig(os.path.join(ARTIFACTS_DIR, '01_actual_vs_champion.png'),␣
 ↪bbox_inches='tight')
```

```
        plt.close()

        plot_residuals_time(y_true_plot, y_pred_plot, valid_df.loc[idx[mask],␣
↪'Date'])
        plt.savefig(os.path.join(ARTIFACTS_DIR, '02_residuals_over_time.png'),␣
↪bbox_inches='tight')
        plt.close()

        plot_residual_hist(y_true_plot, y_pred_plot)
        plt.savefig(os.path.join(ARTIFACTS_DIR, '03_residual_hist.png'),␣
↪bbox_inches='tight')
        plt.close()

        plot_scatter(y_true_plot, y_pred_plot)
        plt.savefig(os.path.join(ARTIFACTS_DIR, '04_actual_vs_pred_scatter.
↪png'), bbox_inches='tight')
        plt.close()

        plot_rolling_wape(valid_df, window=13)
        plt.savefig(os.path.join(ARTIFACTS_DIR, '05_rolling_13w_wape.png'),␣
↪bbox_inches='tight')
        plt.close()

        plot_zoom_last52(valid_df)
        plt.savefig(os.path.join(ARTIFACTS_DIR, '06_zoom_last_52w.png'),␣
↪bbox_inches='tight')
        plt.close()

    print(f"Figures saved to: {ARTIFACTS_DIR}")
```

```
Excel created: /home/linux/Source/VS Code Projects/Advanced Deep Learning

Forecast/CNN BiLSTM Attention/Prediction/Carnivore - Kraken -

2025-06-14.xlsx
Figures saved to: /home/linux/Source/VS Code Projects/Advanced Deep Learning
Forecast/CNN BiLSTM Attention/Prediction/Artifacts
```

[42]:
```
del best_model, tuner
del x_train, y_train, x_tr, y_tr, x_val, y_val, x_test, y_test
del pred_resid_scaled, pred_resid, last_sequence
del df, df_channel, train_df, valid_df, data_resid, train_series
K.clear_session()
plt.close('all')
gc.collect()
```

[42]: 0