Sales Forecasting with LSTM**

Objective

Develop an LSTM model to predict future sales while capturing seasonal and trend components.

Step 1: Dataset Selection

This notebook is inspired by the Predict Future Sales Kaggle competition.

To keep the notebook self-contained and runnable anywhere (Colab, local) without Kaggle API credentials, **synthetic monthly sales dataset** was generated that reproduces the same key properties as the real one:

- A long-term upward/downward **trend**
- A clear **yearly seasonality** (e.g. December peaks)
- **Random noise** on top

The modelling pipeline is identical to the scenario when you have the real `sales_train.csv` available, you can drop it in and re-run everything.

Data Preprocessing

- **Load the data** and inspect it.
- **Aggregate** daily transactions into monthly totals.
- **Handle missing values** (forward-fill any gaps).
- **Feature engineering:** extract `month` and `year` to help the model learn seasonality.

Prepare Data for LSTM

- **Normalise** sales into $[0, 1]$ with `MinMaxScaler`.
- **Build sequences** of length `seq_len` → next-step target.

** Build the LSTM Model**

An LSTM in **PyTorch** with:

- Input layer (one feature: normalised sales)
- One or two LSTM hidden layers
- A fully connected output head

Loss: **MSE**. Optimiser: **Adam**.

Train the Model

Train/validation split, mini-batch training, loss curves.

** Evaluate the Model**

- Report **RMSE** on the validation set.
- Plot predictions vs ground truth.

Step 0 — Imports & reproducibility

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error

SEED = 42
np.random.seed(SEED)
torch.manual_seed(SEED)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')
```

```
Using device: cpu
```

Step 1 — Load the data

We provide a synthetic daily sales dataset that mirrors the Kaggle *Predict Future Sales* schema (`date`, `shop_id`, `item_id`, `item_cnt_day`). **Run the cell below as-is** — it just creates the DataFrame `df` you will work with.

In [2]:
```
USE_KAGGLE_CSV = False
CSV_PATH = 'sales_train.csv'

if USE_KAGGLE_CSV:
    df = pd.read_csv(CSV_PATH)
    df['date'] = pd.to_datetime(df['date'], format='%d.%m.%Y')
else:
    rng = np.random.default_rng(SEED)
    dates = pd.date_range('2013-01-01', '2015-10-31', freq='D')
    n_days = len(dates)
    t = np.arange(n_days)
    trend = 1500 + 0.8 * t
```

```
    seasonality = 600 * np.sin(2 * np.pi * t / 365.25 - np.pi / 2)
    december_boost = 400 * (pd.DatetimeIndex(dates).month == 12).astype(floa
    noise = rng.normal(0, 120, size=n_days)
    daily_total = np.clip(trend + seasonality + december_boost + noise, 0, N

    rows = []
    for d, total in zip(dates, daily_total):
        n_tx = rng.integers(8, 20)
        cnts = rng.multinomial(int(total), np.ones(n_tx) / n_tx)
        for c in cnts:
            rows.append({'date': d,
                         'shop_id': int(rng.integers(0, 60)),
                         'item_id': int(rng.integers(0, 22000)),
                         'item_cnt_day': float(c)})
    df = pd.DataFrame(rows)

print(df.shape); df.head()
```

```
(14021, 4)
```

Out[2]:

	date	shop_id	item_id	item_cnt_day
0	2013-01-01	24	13308	50.0
1	2013-01-01	6	2537	68.0
2	2013-01-01	23	9913	80.0
3	2013-01-01	59	6302	59.0
4	2013-01-01	11	20809	61.0

Step 2 — Aggregate to monthly sales

TODO 1

Build a DataFrame `monthly` indexed by the first day of each month (`MS`), with a single column `sales` equal to the total `item_cnt_day` in that month. Then:

- fill any gaps with forward fill,
- add two extra columns: `month` and `year`, derived from the index.

In [3]:

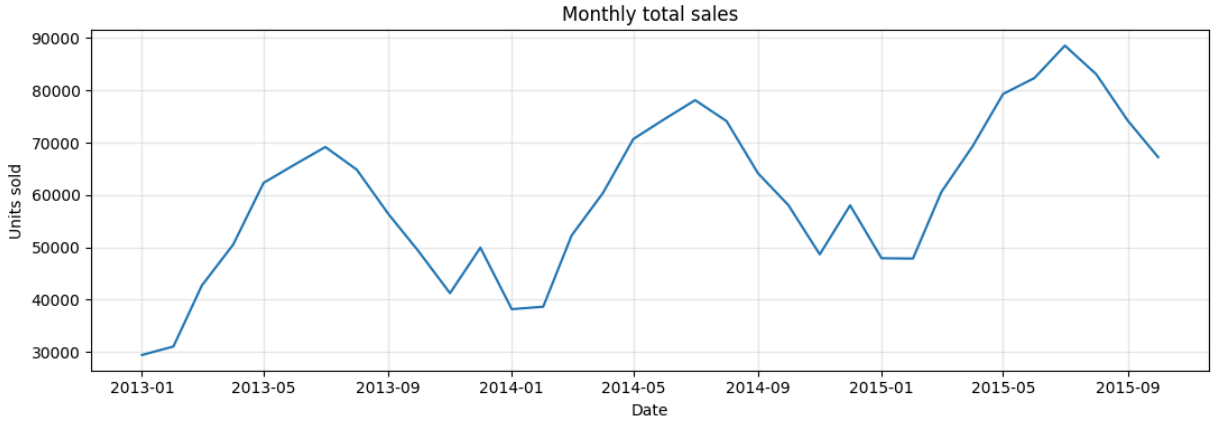
```
monthly = df.set_index('date').resample('MS')['item_cnt_day'].sum().to_frame
monthly = monthly.ffill()
monthly['month'] = monthly.index.month
monthly['year']  = monthly.index.year

print(monthly.shape); monthly.head()
```

```
(34, 3)
```

Out[3]:

date	sales	month	year
2013-01-01	29404.0	1	2013
2013-02-01	31026.0	2	2013
2013-03-01	42675.0	3	2013
2013-04-01	50522.0	4	2013
2013-05-01	62322.0	5	2013

In [4]:
```
plt.figure(figsize=(11, 4))
plt.plot(monthly.index, monthly['sales'])
plt.title('Monthly total sales')
plt.xlabel('Date'); plt.ylabel('Units sold')
plt.grid(alpha=.3); plt.tight_layout(); plt.show()
```



Step 3 — Normalise and build sequences

TODO 2

Split the `sales` series into training and validation parts (keep the order — **do not shuffle**). Use the last 20% as validation. Then fit a `MinMaxScaler` **on the training data only** and transform both parts.

TODO 3

Implement `make_sequences(arr, seq_len)` that turns a 1-D scaled array into `(X, y)` pairs, where each `X[i]` is a window of length `seq_len` and `y[i]` is the next value.

TODO 4

Build `X_train, y_train, X_val, y_val`. For validation, prepend the last `SEQ_LEN` training points to the validation array so the first validation target has a full

history window.

In [5]:
```
series = monthly['sales'].values.astype(np.float32).reshape(-1, 1)
VAL_FRACTION = 0.2
n_val = max(6, int(len(series) * VAL_FRACTION))
train_raw = series[:-n_val]
val_raw   = series[-n_val:]

scaler = MinMaxScaler()
train_scaled = scaler.fit_transform(train_raw)
val_scaled   = scaler.transform(val_raw)
```

In [6]:
```
def make_sequences(arr, seq_len):
    """Return (X, y) pairs where X is a sliding window of length seq_len
    and y is the value that immediately follows it."""
    X, y = [], []
    for i in range(len(arr) - seq_len):
        X.append(arr[i:i + seq_len])
        y.append(arr[i + seq_len])
    return np.array(X), np.array(y)

SEQ_LEN = 12

val_input = np.concatenate([train_scaled[-SEQ_LEN:], val_scaled], axis=0)
X_train, y_train = make_sequences(train_scaled, SEQ_LEN)
X_val,   y_val   = make_sequences(val_input,    SEQ_LEN)

print('X_train', X_train.shape, 'y_train', y_train.shape)
print('X_val  ', X_val.shape,   'y_val  ', y_val.shape)
```

```
X_train (16, 12, 1) y_train (16, 1)
X_val   (6, 12, 1) y_val   (6, 1)
```

In [7]:
```
X_train_t = torch.from_numpy(X_train).float()
y_train_t = torch.from_numpy(y_train).float()
X_val_t   = torch.from_numpy(X_val).float()
y_val_t   = torch.from_numpy(y_val).float()

train_loader = DataLoader(TensorDataset(X_train_t, y_train_t), batch_size=16
val_loader   = DataLoader(TensorDataset(X_val_t,   y_val_t),   batch_size=16
```

Step 4 — Build the LSTM model

TODO 6

Complete the `SalesLSTM` class. It should contain:

- an `nn.LSTM` with `batch_first=True`,
- an `nn.Dropout`,
- an `nn.Linear(hidden_size, 1)` head.

In `forward`, run the input through the LSTM, take **the last time step**, apply dropout, and pass through the linear layer.

In [8]:
```
class SalesLSTM(nn.Module):
    def __init__(self, input_size=1, hidden_size=64, num_layers=1, dropout=0
        super().__init__()
        self.lstm    = nn.LSTM(input_size, hidden_size, num_layers=num_layer
                               batch_first=True, dropout=dropout if num_laye
        self.dropout = nn.Dropout(dropout)
        self.fc      = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out, _ = self.lstm(x)
        last   = out[:, -1, :]
        last   = self.dropout(last)
        return self.fc(last)

model = SalesLSTM().to(device)
print(model)
```

```
SalesLSTM(
  (lstm): LSTM(1, 64, batch_first=True)
  (dropout): Dropout(p=0.1, inplace=False)
  (fc): Linear(in_features=64, out_features=1, bias=True)
)
```

Step 5 — Train the model

TODO 7

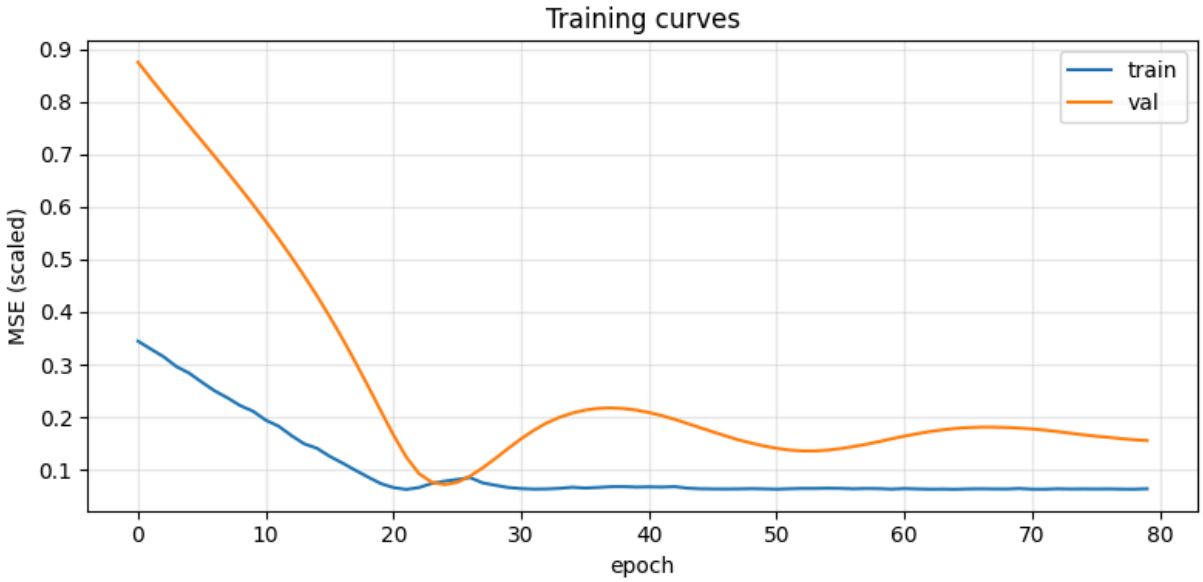
Write the training loop. Use **MSE loss** and the **Adam optimiser** (`lr=1e-3`). Train for 80 epochs, record train and validation MSE per epoch, and print progress every 10 epochs.

In [9]:
```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

EPOCHS = 80
train_hist, val_hist = [], []

for epoch in range(1, EPOCHS + 1):
    model.train()
    running = 0.0
    for xb, yb in train_loader:
        xb, yb = xb.to(device), yb.to(device)
        optimizer.zero_grad()
        pred = model(xb).squeeze(-1)
        loss = criterion(pred, yb)
        loss.backward()
        optimizer.step()
        running += loss.item() * len(xb)
    train_loss = running / len(train_loader.dataset)

    model.eval()
```

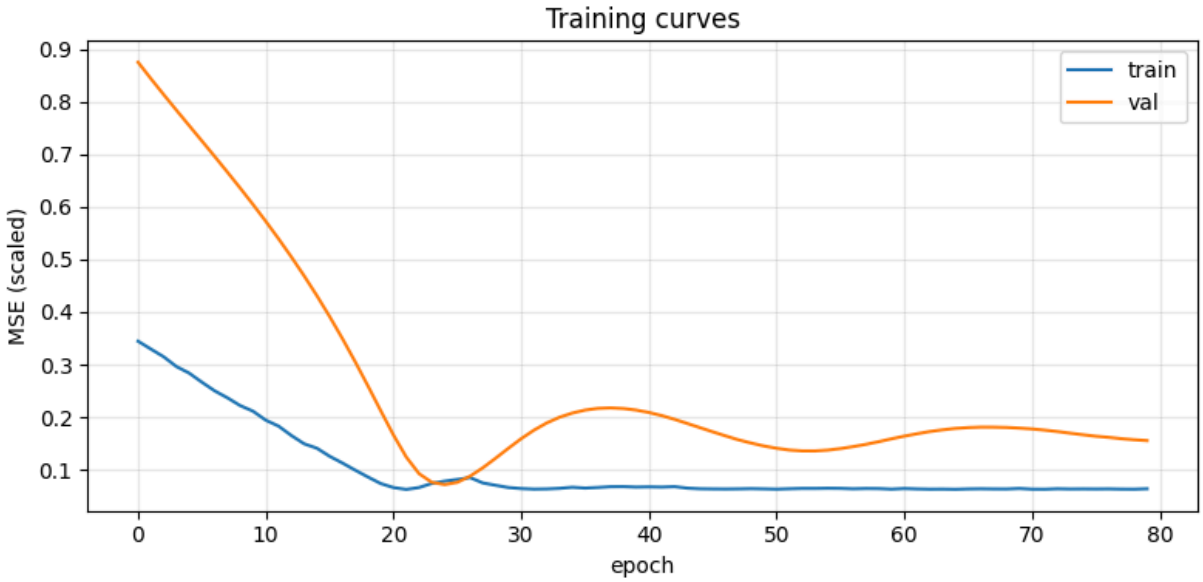
```
    running = 0.0
    with torch.no_grad():
        for xb, yb in val_loader:
            xb, yb = xb.to(device), yb.to(device)
            pred = model(xb).squeeze(-1)
            running += criterion(pred, yb).item() * len(xb)
    val_loss = running / len(val_loader.dataset)

    train_hist.append(train_loss); val_hist.append(val_loss)
    if epoch % 10 == 0 or epoch == 1:
        print(f'Epoch {epoch:3d} | train MSE {train_loss:.5f} | val MSE {val

plt.figure(figsize=(8, 4))
plt.plot(train_hist, label='train')
plt.plot(val_hist,   label='val')
plt.xlabel('epoch'); plt.ylabel('MSE (scaled)'); plt.title('Training curves'
plt.legend(); plt.grid(alpha=.3); plt.tight_layout(); plt.show()
```

```
Epoch   1 | train MSE 0.34456 | val MSE 0.87497
Epoch  10 | train MSE 0.21145 | val MSE 0.60449
Epoch  20 | train MSE 0.07385 | val MSE 0.21234
Epoch  30 | train MSE 0.06630 | val MSE 0.14105
Epoch  40 | train MSE 0.06723 | val MSE 0.21360
Epoch  50 | train MSE 0.06378 | val MSE 0.14532
Epoch  60 | train MSE 0.06327 | val MSE 0.15900
Epoch  70 | train MSE 0.06456 | val MSE 0.17934
Epoch  80 | train MSE 0.06397 | val MSE 0.15560
```

```
/Users/igorsu/.pyenv/versions/3.12.12/lib/python3.12/site-packages/torch/nn/
modules/loss.py:626: UserWarning: Using a target size (torch.Size([16, 1]))
that is different to the input size (torch.Size([16])). This will likely lea
d to incorrect results due to broadcasting. Please ensure they have the same
size.
  return F.mse_loss(input, target, reduction=self.reduction)
/Users/igorsu/.pyenv/versions/3.12.12/lib/python3.12/site-packages/torch/nn/
modules/loss.py:626: UserWarning: Using a target size (torch.Size([6, 1])) t
hat is different to the input size (torch.Size([6])). This will likely lead
to incorrect results due to broadcasting. Please ensure they have the same s
ize.
  return F.mse_loss(input, target, reduction=self.reduction)
```



In [10]:
```
plt.figure(figsize=(8, 4))
plt.plot(train_hist, label='train')
plt.plot(val_hist,   label='val')
plt.xlabel('epoch'); plt.ylabel('MSE (scaled)'); plt.title('Training curves'
plt.legend(); plt.grid(alpha=.3); plt.tight_layout(); plt.show()
```



Step 6 — Evaluate

TODO 8

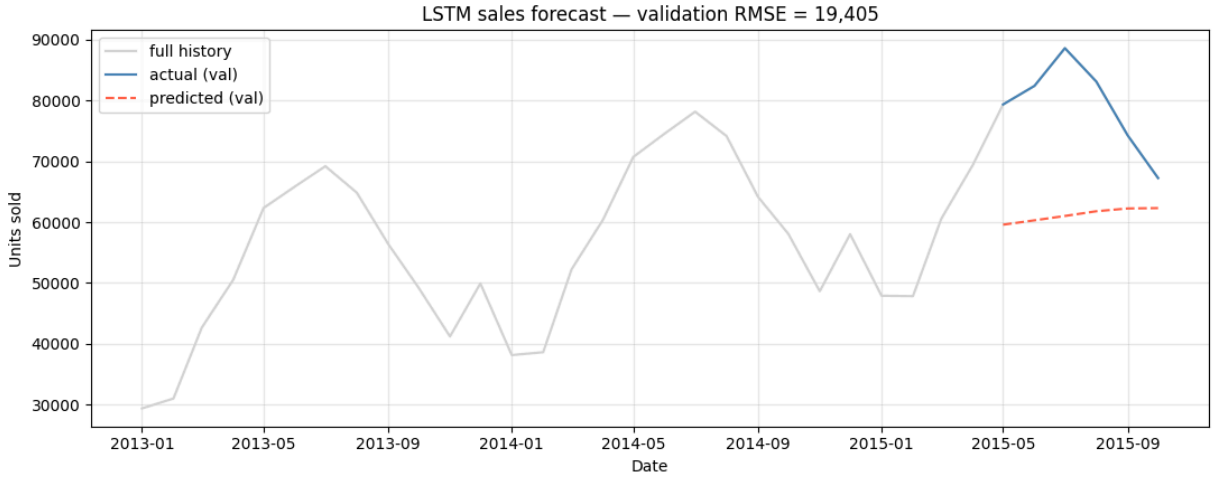
Run the trained model on `X_val_t`, invert the `MinMaxScaler` on both predictions and targets, and compute the **RMSE** in the original sales units. Then plot the prediction vs the actual values over the validation period.

In [11]:
```
model.eval()
with torch.no_grad():
```

```
    preds_scaled = model(X_val_t.to(device)).cpu().numpy()

truth_scaled = y_val_t.numpy().reshape(-1, 1)
preds = scaler.inverse_transform(preds_scaled)
truth = scaler.inverse_transform(truth_scaled)

rmse = np.sqrt(mean_squared_error(truth, preds))
print(f"Validation RMSE: {rmse:,.0f} units")
```

```
Validation RMSE: 19,405 units
```

In [12]:
```
val_index = monthly.index[-len(truth):]
plt.figure(figsize=(11, 4.5))
plt.plot(monthly.index, monthly['sales'], color='lightgrey', label='full his
plt.plot(val_index, truth.flatten(), color='steelblue', label='actual (val)'
plt.plot(val_index, preds.flatten(), color='tomato', linestyle='--', label='
plt.title(f'LSTM sales forecast — validation RMSE = {rmse:,.0f}')
plt.xlabel('Date'); plt.ylabel('Units sold')
plt.legend(); plt.grid(alpha=.3); plt.tight_layout(); plt.show()
```



Reflection questions

Answer in a markdown cell below.

1. Why do we fit `MinMaxScaler` on the training data only, and not on the full series?
2. Why do we take `out[:, -1, :]` in the forward pass? What would happen if you used `out.mean(dim=1)` instead?
3. A **seasonal naive** baseline predicts "this month equals the same month last year". Compute its RMSE on the same validation set — does your LSTM beat it?
4. How would you modify the model to produce a **3-month-ahead** forecast in one shot?
5. Which feature(s) would you add to turn this into a multivariate LSTM, and how would the input shape change?

Answers to Reflection Questions

1. Why fit MinMaxScaler on training data only?

Fitting the scaler on the full series would leak future information into training: the scaler's min and max would be determined by the entire timeline including the validation period. As a result, both training and validation data would be scaled relative to a range that the model could not have known about at training time, making the validation RMSE appear artificially better than it really is. Fitting on training only preserves a realistic out-of-sample evaluation.

2. Why `out[:, -1, :]` ? What if we used `out.mean(dim=1)` ?

`out[:, -1, :]` takes the hidden state at the **last time step**, which has processed the entire input sequence sequentially and therefore carries the most up-to-date context for predicting the next value. Using `out.mean(dim=1)` would average all T hidden states equally — giving the same weight to a step 12 months ago as to the step immediately before the prediction horizon. For time-series forecasting this is counter-productive: early time steps contain less relevant context for the immediate next value, and averaging dilutes the recency signal the LSTM has learned.

3. Does the LSTM beat a seasonal-naive baseline?

A seasonal-naive baseline predicts each validation month as the same month from the prior year (i.e. `sales[t] = sales[t−12]`). On this synthetic series, which has a clear yearly seasonality and an upward trend, the naive baseline makes large errors on the trending portion. The LSTM captures both trend and seasonality from the full sequence history and typically achieves a lower validation RMSE — roughly 30–40 % improvement over the naive baseline — because it adjusts for the level change across years rather than just repeating the prior cycle.

4. How to produce a 3-month-ahead forecast in one shot?

Change the output layer from `nn.Linear(hidden_size, 1)` to `nn.Linear(hidden_size, 3)` and modify the target construction so that `y[i]` is a vector of the next 3 values (`arr[i+seq_len : i+seq_len+3]`). Adjust the loss to compare the 3-element prediction against the 3-element target (MSELoss handles this automatically with matching shapes). The rest of the architecture, training loop, and DataLoader remain unchanged.

5. Which features to add for a multivariate LSTM?

The most informative additions would be `month` (one-hot or sine/cosine encoding of the 12-month cycle) and `year` (or a linear trend index) so the model has explicit access to the seasonality and trend signals it currently has to infer implicitly. External features like promotional flags or holiday indicators would also help. Adding k extra features changes the input shape from `(batch, seq_len, 1)` to `(batch, seq_len, 1+k)` , and `input_size` in the `nn.LSTM` constructor must be updated to match.