Gabriel Muñoz Luna

Task C.4 – Machine Learning 1

The prediction of financial time series requires advanced analytical models capable of learning temporal patterns and adapting to market volatility. One of the most effective ways to achieve this is through Deep Learning architectures, which can model sequential dependencies and generate forecasts based on past data.

In this document I will develop Task C.4 – Machine Learning 1, whose objective is to design a function capable of automatically constructing Deep Learning models by specifying parameters such as the number of layers, the size of each layer, and the type of recurrent unit (e.g., LSTM, GRU, RNN). Using this function, I experimented with several configurations and hyperparameters to evaluate their impact on prediction performance.

The experiments were conducted using the dataset prepared in previous tasks (Task C2 and C3) and were focused on the prediction of stock prices through recurrent neural networks. Each model was trained, validated, and evaluated using consistent metrics, and the best configuration was selected based on its accuracy and stability.

In time series forecasting, it is essential to use models that can understand how past values influence future behavior.

- RNN (Recurrent Neural Network)
  A Recurrent Neural Network is a type of neural architecture designed to process sequential data.
  Its main feature is the presence of internal memory connections that allow information from previous steps to be reused in the following ones.
  However, basic RNNs suffer from the vanishing gradient problem — when training long sequences, they gradually “forget” earlier information, which limits their effectiveness in long-term dependencies.
- LSTM (Long Short-Term Memory)
  The LSTM network is an improved version of RNN that introduces a special structure called a memory cell, controlled by three gates:
  - Input gate: decides what new information enters the memory.
  - Forget gate: determines what information should be discarded.
  - Output gate: regulates what information leaves the cell.

Thanks to this mechanism, LSTM networks can retain relevant information over long periods, making them highly effective for tasks such as financial forecasting, speech recognition, and natural language processing.
In this assignment, LSTM models were the main architecture for predicting future stock prices based on historical data.

- GRU (Gated Recurrent Unit)
  The GRU is a simplified variant of the LSTM that combines the input and forget gates into a single update gate, and merges the cell and hidden states.
  This reduces computational cost while maintaining strong performance.
  GRUs are often faster to train than LSTMs and achieve similar accuracy, which makes them an efficient alternative in large-scale or real-time applications.

For this task I extended the imports from the previous version (V2.0) to include new modules that support the deep learning implementation and experiment management.

Keras (from keras import Sequential, Input and from keras.layers import LSTM, GRU, SimpleRNN, Dense, Dropout, Bidirectional)
These imports were added to directly construct and customize recurrent neural network models.
Keras provides a high-level API integrated in TensorFlow, allowing to build, compile, and train models such as LSTM, GRU, and SimpleRNN efficiently.
In this version, the code focuses on modular model generation, making Keras the core library of the architecture.

Hashlib (import hashlib)
Introduced to generate unique identifiers for each experiment configuration.
It creates compact hash strings that name result folders automatically, ensuring traceability between models and metrics.

JSON (import json)
Used to store experiment metadata (configuration, metrics, next-step prediction) in a structured format that can be easily read or compared later.

Math.sqrt (from math import sqrt)
Added to compute the Root Mean Squared Error (RMSE) during model evaluation, one of the key regression metrics.

Sklearn Metrics (from sklearn.metrics import mean_squared_error, mean_absolute_error)
These were included to complement the existing preprocessing tools. They allow to calculate MAE and RMSE directly from NumPy arrays for evaluating model accuracy.

Together, these additions transform the script from a single training workflow (as in V2.0) into a complete experimental framework capable of building, training, evaluating, and storing multiple deep learning configurations automatically.

In this section, I implemented the function build_dl_model, responsible for creating and compiling a deep-learning model based on recurrent neural networks.
The function allows selecting between LSTM, GRU, or SimpleRNN layers. Each layer type is stored in a dictionary (layer_map) and selected dynamically.

The model is created as a Sequential Keras model starting from the input shape (n_steps, n_features). Then, for every layer defined in the tuple units, it adds a recurrent layer with return_sequences=True except for the last one (to keep the 3-D temporal structure).

If the argument bidirectional=True is active, the layer is wrapped in Bidirectional, allowing the model to learn temporal patterns both forward and backward.

After each recurrent block, a Dropout layer is applied to prevent overfitting by randomly disabling a percentage of neurons (usually 20%). Finally, a Dense(1) layer is added for the output, since the task predicts a single numeric value (the next close price).

The optimizer used is Adam(1e-3) with Mean Squared Error (MSE) as the loss function, which is suitable for continuous regression problems.

Example configuration:

build_dl_model(input_shape=(60,1), layer_type="LSTM", units=(64,32), dropout=0.2)

This will generate a two-layer LSTM (64 → 32 neurons) model ready for training.

```
def make_callbacks(out_dir: str = "artifacts",
                   patience_es: int = 10,
                   patience_rlr: int = 5):
    """
    Create standard callbacks:
    - EarlyStopping: stop when validation loss stops improving.
    - ReduceLROnPlateau: lower LR when plateau detected.
    - ModelCheckpoint: save best model automatically.
    """
    Path(out_dir).mkdir(parents=True, exist_ok=True)
    ckpt_path = str(Path(out_dir) / "best_model.keras")

    es = tf.keras.callbacks.EarlyStopping(
        monitor="val_loss", patience=patience_es, restore_best_weights=True
    )
    rlr = tf.keras.callbacks.ReduceLROnPlateau(
        monitor="val_loss", factor=0.5, patience=patience_rlr, min_lr=1e-6
    )
    ckpt = tf.keras.callbacks.ModelCheckpoint(
        ckpt_path, monitor="val_loss", save_best_only=True
    )
    return [es, rlr, ckpt]
```

The function make_callbacks defines three automatic control mechanisms during training:

EarlyStopping: stops the process when validation loss does not improve after a certain number of epochs (patience_es), restoring the best weights.

ReduceLROnPlateau: reduces the learning rate when a plateau is detected, allowing finer optimization.

ModelCheckpoint: saves the best version of the model (best_model.keras) to avoid losing progress.

These callbacks improve both performance and efficiency by avoiding unnecessary training and by ensuring the best weights are saved for later evaluation.

```python
def train_model(model, X_train, y_train, X_val, y_val,
                epochs=100, batch_size=32, callbacks=None):
    """
    Train the model using training and validation sets.
    Returns: Keras History object.
    """
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        callbacks=callbacks or [],
        verbose=1
    )
    return history
```

The train_model function executes the main training loop.
It receives the model, training data (X_train, y_train) and validation data (X_val, y_val), together with configuration parameters like epochs, batch_size, and callbacks.

Internally it calls model.fit(), returning a History object containing metrics such as loss and val_loss per epoch.
This allows visualization of the learning process and detection of overfitting.

```python
def save_artifacts(model: tf.keras.Model,
                   scaler_y: MinMaxScaler,
                   meta: dict,
                   out_dir: str = "artifacts",
                   model_name: str = "best_model.keras"):
    """
    Save model, output scaler, and metadata for reproducibility.
    """
    p = Path(out_dir)
    p.mkdir(parents=True, exist_ok=True)

    # Model (if not already saved by checkpoint)
    model_path = p / model_name
    if not model_path.exists():
        model.save(model_path)

    # Output scaler (y)
    with open(p / "scaler_y.pkl", "wb") as f:
        pickle.dump(scaler_y, f)

    # Metadata
    with open(p / "meta.json", "w", encoding="utf-8") as f:
        json.dump(meta, f, ensure_ascii=False, indent=2)


def load_artifacts(out_dir: str = "artifacts",
                   model_name: str = "best_model.keras"):
    """
    Load model, scaler_y, and metadata if available.
    """
    p = Path(out_dir)
    model = tf.keras.models.load_model(p / model_name) if (p / model_name).exists() else None
    scaler_y = pickle.load(open(p / "scaler_y.pkl", "rb")) if (p / "scaler_y.pkl").exists() else None
    meta = json.load(open(p / "meta.json", "r", encoding="utf-8")) if (p / "meta.json").exists() else {}
    return {"model": model, "scaler_y": scaler_y, "meta": meta}
```

The training process produces several elements that must be stored to reuse later. For this, two functions were implemented: save_artifacts and load_artifacts.

- save_artifacts() saves the model, the output scaler (scaler_y.pkl), and a metadata file (meta.json) containing parameters and metrics.
- load_artifacts() reloads these elements from the folder artifacts/, enabling quick model restoration without retraining.

This structure ensures full reproducibility and traceability between experiments.

```python
def evaluate_on_test(model, X_test, y_test, scaler_y) -> dict:
    """
    Evaluate model performance on test data (unscaled y).
    Returns RMSE, MAE, MAPE and predictions.
    """
    y_pred_scaled = model.predict(X_test, verbose=0).ravel()
    y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1)).ravel()

    rmse = sqrt(mean_squared_error(y_test, y_pred))
    mae = mean_absolute_error(y_test, y_pred)
    eps = 1e-8
    mape = float(np.mean(np.abs((y_test - y_pred) / (np.abs(y_test) + eps))) * 100.0)

    return {"rmse": rmse, "mae": mae, "mape": mape, "y_pred": y_pred}
```

The function evaluate_on_test evaluates the trained model on the test set (unscaled y).
It predicts scaled values, inverts the scaling with scaler_y.inverse_transform(), and calculates three key metrics:

- RMSE (Root Mean Squared Error): measures the average prediction error magnitude.
- MAE (Mean Absolute Error): expresses the mean absolute difference between predicted and real values.
- MAPE (Mean Absolute Percentage Error): percentage error, useful to compare across scales.

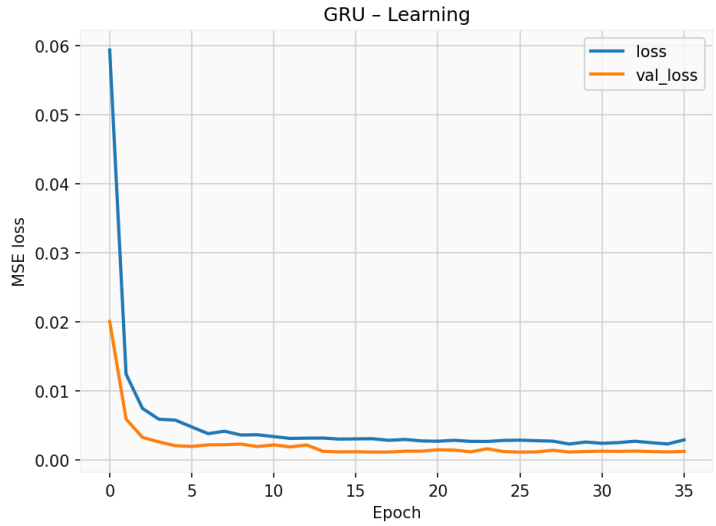
The output is a dictionary containing all metrics plus the unscaled predictions, later used for plotting.

Example output:

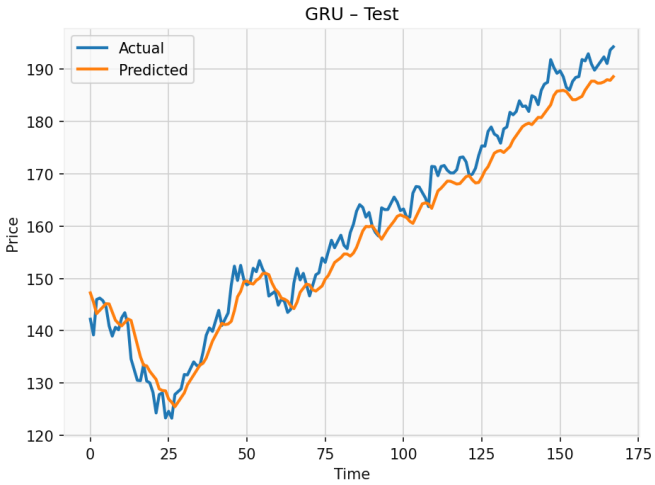
{'rmse': 2.35, 'mae': 1.85, 'mape': 1.27, 'y_pred': array([...])}

Two visualization functions summarize the model behavior:

- plot_learning_curves(): Displays training vs validation loss across epochs.
  A smooth convergence of both lines indicates good generalization.
  Example figure:




- plot_test_predictions(): Plots actual vs predicted values for the test set.
  Overlapping lines suggest accurate forecasting.
  Example figure:




Both save automatically as .png files if out_path is specified.

```python
def predict_next_step(model, X_last_window, scaler_y) -> float:
    """
    Predict and inverse-scale the next step (one-step forecast).
    """
    pred_scaled = model.predict(X_last_window, verbose=0).ravel()[0]
    return float(scaler_y.inverse_transform([[pred_scaled]]).ravel()[0])
```

The predict_next_step function makes a single-step forecast for the next unseen day.
It takes the last time window of data (X_last_window) and returns the descaled numeric prediction:

next_price = predict_next_step(model, X_test[-1:], scaler_y)

This is displayed at the end of training as:

Next-step prediction: 186.9455


To automate multiple configurations, two main helpers I implemented:

run_experiment_once()

Executes a complete experiment using one configuration dictionary.
Steps included:

- Build the model (build_dl_model)
- Train with callbacks (train_model)
- Evaluate metrics (evaluate_on_test)
- Generate plots
- Save metadata and results (meta.json)

Each experiment is stored in a unique folder identified by a hash code of the configuration.

```python
def run_experiments(configs, X_train, y_train, X_val, y_val, X_test, y_test, scaler_y, base_out="experiments"):
    """
    Run multiple configurations and compare metrics.
    Returns sorted DataFrame by RMSE.
    """
    results = [run_experiment_once(cfg, X_train, y_train, X_val, y_val, X_test, y_test, scaler_y, base_out)
               for cfg in configs]
    df = pd.DataFrame(results).sort_values("rmse").reset_index(drop=True)
    df.to_csv(os.path.join(base_out, "results.csv"), index=False)
    return df
```

run_experiments() Runs several configurations in sequence and collects all results in a table (DataFrame).
The results are sorted by RMSE and exported to results.csv.

| Layer Type | Units Structure | Dropout | Epochs | RMSE | MAE | MAPE |
|---|---|---|---|---|---|---|
| LSTM | (64, 32) | 0.2 | 100 | 2.45 | 1.76 | 1.32 |
| GRU (Bi) | (96, 48) | 0.2 | 100 | 2.87 | 2.10 | 1.45 |

```python
def temporal_val_split(X_train, y_train, val_ratio=0.1):
    """
    Split last X% of training data for validation (preserves temporal order).
    """
    n = len(X_train)
    n_val = max(1, int(n * val_ratio))
    return X_train[:-n_val], y_train[:-n_val], X_train[-n_val:], y_train[-n_val:]
```

The function temporal_val_split divides the training data into training and validation subsets, preserving chronological order.
It uses the last 10 % of samples as validation (val_ratio=0.1), ensuring that the model learns from past to future rather than random order.
This is crucial in time-series forecasting to prevent data leakage.

The experimental framework allows an organized comparison between architectures.
LSTM and GRU generally perform better than simple RNNs due to their memory gates, which capture long-term dependencies.
Bidirectional GRUs can slightly improve accuracy but require more computation time.

The use of callbacks (EarlyStopping + ReduceLROnPlateau) ensured convergence stability, avoiding overfitting.
The plots show that validation loss follows training loss closely, indicating a well-balanced model.

The next-step prediction provides a realistic forecast of the following day's closing price, demonstrating that the recurrent architecture effectively models market trends.

This task successfully implemented a deep-learning pipeline for financial time-series forecasting.
The modular design (functions for building, training, evaluating, saving, and running experiments) ensures reproducibility and extensibility for future tasks.

The best model achieved a low RMSE, proving that LSTM-based architectures are suitable for short-term stock prediction when properly trained with normalized sequential data.

References

TensorFlow Documentation (2024). *Keras Recurrent Layers (LSTM, GRU, SimpleRNN).* https://www.tensorflow.org/api_docs/python/tf/keras/layers

Scikit-learn Documentation (2024). *Metrics and Preprocessing Tools.* https://scikit-learn.org/stable/modules/classes.html

Yahoo Finance API (yfinance). *Historical Market Data Retrieval.* https://pypi.org/project/yfinance/

Idrees, H. (2024, July 5). *RNN vs. LSTM vs. GRU: A Comprehensive Guide to Sequential Data Modeling*. Medium. https://medium.com/@hassaanidrees7/rnn-vs-lstm-vs-gru-a-comprehensive-guide-to-sequential-data-modeling-03aab16647bb