Task C.4 Report – Deep Learning Model Builder

Anh Vu Le
104653505

Contents

Part 1 – Code Explanation

In this part, I explain the main function build_sequence_model implemented in model_builder.py. The function allows flexible creation of deep learning models (LSTM, GRU, RNN) by specifying hyperparameters such as number of layers, units per layer, dropout, optimizer, etc. Below I highlight the key code sections and explain their role.

- **Lines 30–36 – Recurrent Layer Mapping (model_builder.py)**

```
30  # Mapping layer name to actual Keras class
31  RECURRENT_LAYER_MAP = {
32      'lstm': LSTM,
33      'gru': GRU,
34      'rnn': SimpleRNN,
35      'simplernn': SimpleRNN,
36  }
```

This dictionary provides a mapping between a string (e.g., "lstm") and the corresponding Keras layer class (LSTM).

- It ensures the function can dynamically choose which recurrent cell to use, based on user input.

- **Lines 108–115 – Handling Layer Units (model_builder.py)**

```
108     if isinstance(layer_units, int):
109         layer_units_list = [layer_units]
110     else:
111         assert len(layer_units) > 0, "layer_units list must not be empty"
112         layer_units_list = list(layer_units)
113 
114     n_layers = len(layer_units_list)
115 
```

Allows layer_units to be flexible: either a single integer (e.g., 64) or a list (e.g., [64, 32]).

- If a single int is given, it is converted into a list so the later loop can handle it uniformly.

- **Lines 116–124 – Return Sequences Strategy (model_builder.py)**

```python
    # Determine return_sequences flags
    if return_sequences_strategy == 'auto':
        return_sequences_flags = [True]*(n_layers-1) + [False]
    elif isinstance(return_sequences_strategy, list):
        assert len(return_sequences_strategy) == n_layers, "return_sequences list length mismatch"
        return_sequences_flags = return_sequences_strategy
    else:
        raise ValueError("return_sequences_strategy must be 'auto' or list[bool]")
```

- In recurrent models, intermediate layers usually return sequences (return_sequences=True) so the next layer receives the full sequence.
- Only the last layer returns a single vector (False).
- This logic automatically sets the correct flags depending on number of layers.

🔹 Lines 139–150 – Building Layers (model_builder.py)

```python
139     # Build recurrent stack
140     for i, (units, ret_seq) in enumerate(zip(layer_units_list, return_sequences_flags)):
141         layer_args = dict(units=units, return_sequences=ret_seq)
142         # Only pass recurrent_dropout if > 0 (some layers may not support otherwise)
143         if recurrent_dropout > 0:
144             layer_args['recurrent_dropout'] = recurrent_dropout
145         recurrent_layer: Layer = CellClass(**layer_args)
146         if bidirectional:
147             recurrent_layer = Bidirectional(recurrent_layer)
148         model.add(recurrent_layer)
149         if dropout > 0:
150             model.add(Dropout(dropout))
151
```

- This loop iterates through each specified layer unit.
- Dynamically creates either LSTM, GRU, or RNN layers.
- Optionally wraps the layer in Bidirectional.
- Adds Dropout after each layer to reduce overfitting.

🔹 Line 153 – Output Layer

```python
152     # Output layer
153     model.add(Dense(last_layer_units, activation=output_activation))
```

- Final dense layer outputs a single predicted value (e.g., stock price).
- Linear activation ("linear") is used for regression tasks.

🔹 Line 155 – Model Compilation (model_builder.py)

```
model.compile(optimizer=optimizer_instance, loss=loss, metrics=metrics)
```

- Configures the training process: optimizer, loss function, and metrics.
- In this project, we often used Adam or RMSProp, with mean_absolute_error (MAE) as metric.

🔹 Lines 180–185 – Returning Config (model_builder.py)

```
180     return {
181         'model': model,
182         'config': config,
183         'summary_str': summary_str,
184     }
185
```

- Returns not just the model, but also its configuration and textual summary.
- Helps track experiments systematically.

Part 2 – Inheritage from P1

P1 Reference Implementation (stock_prediction.py line 144-146):

```
144 def create_model(sequence_length, n_features, units=256, cell=LSTM, n_layers=2, dropout=0.3,
145                 loss="mean_absolute_error", optimizer="rmsprop", bidirectional=False):
146     model = Sequential()
```

Implementation (model_builder.py lines 44-59):

```
44 def build_sequence_model(
45     sequence_length: int,
46     n_features: int,
47     layer_type: str = 'lstm',
48     layer_units: Union[int, List[int]] = 64,
49     dropout: float = 0.2,
50     recurrent_dropout: float = 0.0,
51     bidirectional: bool = False,
52     last_layer_units: int = 1,
53     output_activation: Optional[str] = 'linear',
54     optimizer: str = 'adam',
55     learning_rate: Optional[float] = None,
56     loss: str = 'mean_squared_error',
57     metrics: Optional[List[str]] = None,
58     return_sequences_strategy: str = 'auto',  # 'auto' or explicit list[bool]
59     name: Optional[str] = None,
```

Comparison 1: create_mode (stock_prediction.py) vs. build_sequence_model (model_builder.py)

- **Relationship:** create_mode is a simplified, specific implementation of the general and flexible build_sequence_model function.
- **Parameter Inheritance:** create_mode inherits concepts like units, cell (layer type), optimizer, and loss, but hardcodes them with specific values (e.g., cell=LSTM, units=256) instead of accepting them as highly configurable arguments.
- **Workflow:** build_sequence_model is designed for custom model creation via many parameters. create_mode is for quickly creating a standard, pre-defined model with minimal configuration.

LAYER CREATION LOGIC INHERITANCE:

P1 Reference Pattern (lines 146-158):

```
147     for i in range(n_layers):
148         if i == 0:
149             # first layer
150             if bidirectional:
151                 model.add(Bidirectional(cell(units, return_sequences=True), batch_input_sh
152             else:
153                 model.add(cell(units, return_sequences=True, batch_input_shape=(None, sequ
154         elif i == n_layers - 1:
155             # last layer
156             if bidirectional:
157                 model.add(Bidirectional(cell(units, return_sequences=False)))
158             else:
159                 model.add(cell(units, return_sequences=False))
160         else:
161             # hidden layers
162             if bidirectional:
163                 model.add(Bidirectional(cell(units, return_sequences=True)))
164             else:
165                 model.add(cell(units, return_sequences=True))
```

Implementation (model_builder.py lines 139-150):

```
139     # Build recurrent stack
140     for i, (units, ret_seq) in enumerate(zip(layer_units_list, return_sequences_flags)):
141         layer_args = dict(units=units, return_sequences=ret_seq)
142         # Only pass recurrent_dropout if > 0 (some layers may not support otherwise)
143         if recurrent_dropout > 0:
144             layer_args['recurrent_dropout'] = recurrent_dropout
145         recurrent_layer: Layer = CellClass(**layer_args)
146         if bidirectional:
147             recurrent_layer = Bidirectional(recurrent_layer)
148         model.add(recurrent_layer)
149         if dropout > 0:
150             model.add(Dropout(dropout))
```

Loop in model_builder inherits the fundamental goal from Loop P1: to correctly build a stack of recurrent layers by managing the return_sequences flag. It provides a specific, logic-driven solution (if/else) to the general problem that Loop P1 solves with a flexible, data-driven approach.

Part 3 – Experiment Results

Using this function, I tested different model architectures (LSTM, GRU, RNN) with varying hyperparameters.

🔹 **Summary Table of Experiments**

Model	Layers	Units	Epochs	Batch	Params
LSTM	1	[64]	10	32	17985
LSTM	2	[64, 32]	20	16	30369
LSTM	1	[128]	15	64	68737
GRU	1	[64]	10	32	13697
GRU	2	[64, 32]	20	16	23073
GRU	1	[128]	15	64	51969
RNN	1	[64]	10	32	4545
RNN	2	[64, 32]	20	16	7617
RNN	1	[128]	15	64	17281

```
📈 DETAILED MODEL ARCHITECTURE EVIDENCE:
============================================================
🔹 LSTM Models:
   [64] units, 1 layers → 17985 parameters
   [64, 32] units, 2 layers → 30369 parameters
   [128] units, 1 layers → 68737 parameters
🔹 GRU Models:
   [64] units, 1 layers → 13697 parameters
   [64, 32] units, 2 layers → 23073 parameters
```

Part 4 - Conclusion

- The function correctly allows flexible construction of recurrent neural networks.
- Compared to P1, my implementation is **more modular and generalizable**.
- Experimental results confirm different architectures (LSTM/GRU/RNN) can be trained with varying hyperparameters.
- LSTM generally gave the best performance, while RNN was lighter but less accurate.