Dynamic Intercropping Compatibility Model Documentation

---

1. Overview

This Python script implements a deep learning model to predict intercropping compatibility and expected yield impact based on crop traits and textual historical/compatibility information. It allows dynamic input feature sizes, meaning the model automatically adapts to any number of trait columns specified in preprocessing.

The model architecture combines: - LSTM for trait sequence encoding (numerical features) - Embedding + transformer layers for textual crop information - Fully connected layers for joint classification (compatibility) and regression (yield impact)

It is also flexible in the number of outputs, so new regression or classification targets can be added by changing only a few parts of the code.

---

2. Libraries and Dependencies

The script uses: - **PyTorch**: neural network implementation (`nn.Module`, `DataLoader`, `Dataset`) - **scikit-learn**: for MinMax scaling and K-Fold validation - **pandas**: data handling from JSON - **numpy**: numerical operations - **joblib**: saving scaler and tokenizer - **matplotlib**: optional visualization

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset, TensorDataset
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import joblib
import numpy as np
import random
import json
import os
```

---

3. Reproducibility

Ensures that every run produces the same results by fixing random seeds:

```
SEED = 42
random.seed(SEED)
```

```
np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)
```

- Also ensures deterministic behavior for CUDA operations.

---

4. Dataset and Tokenizer

4.1 IntercropDataset

A PyTorch `Dataset` class to wrap traits, text, and yield labels:

```
class IntercropDataset(Dataset):
    def __getitem__(self, idx):
        return self.traits[idx], self.text[idx], self.yield_label[idx]
```

- Allows indexing like `dataset[i]`.

4.2 SimpleTokenizer

A basic word-level tokenizer:

```
class SimpleTokenizer:
    def __init__(self, texts, vocab_size=5000):
        ...
    def encode(self, text, max_len=32):
        ...
```

- Builds a vocabulary of the top `vocab_size` words. - Maps unknown words to `<UNK>` and pads sequences to `max_len`.

---

5. Data Preprocessing

The preprocessing function transforms raw JSON data into tensors suitable for the model.

5.1 Steps:

1. **Read JSON**: load crop pairs, compatibility, and historical data.
2. **Extract nutrient features**: N, P, K min/max values for both crops.

3. **Extract shared traits**: Water overlap, Root overlap, Pest sharing, Nutrient competition, Space utilization, Soil resource sharing.
4. **Parse yield impact**: convert percentages or ranges into float between 0.0-1.0.
5. **Text preparation**: combine `Canopy_Cover_Synergy`, `Height_Compatibility`, and `Farmer_Feedback` into one string.

```
df = pd.DataFrame(records)
trait_cols = [col for col in df.columns if col not in ['Text_Info',
'Yield_Impact']]
scaler = MinMaxScaler()
trait_tensor = torch.tensor(scaler.fit_transform(df[trait_cols].values),
dtype=torch.float32)
```

- `trait_cols` dynamically defines the input features (`trait_dim`). - `trait_tensor` is the normalized numerical input.

```
text_tensor = torch.tensor([tokenizer.encode(t) for t in df['Text_Info']],
dtype=torch.long)
yield_tensor = torch.tensor(df['Yield_Impact'].values.astype(float),
dtype=torch.float32).unsqueeze(1)
```

- `text_tensor` is tokenized text input. - `yield_tensor` is regression target.

---

6. Model Architecture and Layer Details

```
class CompatibilityModel(nn.Module):
    def __init__(self, trait_dim, text_dim=64, hidden_dim=128, num_outputs=2):
        ...
```

6.1 LSTM Layer (Traits)

```
self.lstm = nn.LSTM(trait_dim, hidden_dim, batch_first=True, bidirectional=True,
num_layers=2, dropout=0.25)
```

- Processes numerical trait features. - **bidirectional**: captures forward and backward dependencies. - **num_layers=2**: stacked LSTM for deeper representation. - **dropout=0.25**: prevents overfitting. - Output: concatenated forward/backward hidden states `[B, 2*hidden_dim]`.

6.2 Embedding Layer (Text)

```
self.embedding = nn.Embedding(5000, text_dim)
self.text_proj = nn.Linear(text_dim, hidden_dim * 2)
```

- Converts tokens to dense vectors. - `text_proj` maps embedding to same size as LSTM hidden states for addition.

6.3 Transformer Encoder

```
self.transformer_layer = nn.TransformerEncoderLayer(d_model=hidden_dim * 2,
nhead=4, dropout=0.25, batch_first=True)
self.transformer = nn.TransformerEncoder(self.transformer_layer, num_layers=4)
```

- Processes the combined LSTM + text features. - **nhead=4**: multi-head attention captures multiple interactions. - **num_layers=4**: stacked transformer layers.

6.4 Fully Connected MLP (Sequential)

```
self.mlp = nn.Sequential(
    nn.Linear(hidden_dim * 2, 256),
    nn.BatchNorm1d(256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, 64),
    nn.ReLU(),
    nn.Linear(64, num_outputs)
)
```

- **Why this sequence:** - **Linear → BatchNorm → ReLU**: ensures stable, normalized activations and faster training. - **Dropout**: prevents overfitting. - **Multiple Linear + ReLU layers**: allows learning complex non-linear relationships. - **Final Linear**: flexible output layer, can be classification logits, regression, or both. - **Alternative sequences**: could use `LeakyReLU`, `tanh`, or LayerNorm, but the current configuration is standard and effective.

6.5 Activation Functions

- **ReLU**: used in hidden layers for non-linearity and to avoid vanishing gradients.
- **Sigmoid**: applied on classification logit to get probability between 0–1.
- **Linear**: regression outputs are raw numbers (no activation).

6.6 Forward Pass

```
def forward(self, trait_seq, text_vec):
    _, (hidden, _) = self.lstm(trait_seq)
    bi_hidden = torch.cat((hidden[-2], hidden[-1]), dim=-1)
    text_emb = self.embedding(text_vec).mean(dim=1)
    combined = bi_hidden + self.text_proj(text_emb)
    transformed = self.transformer(combined.unsqueeze(1)).squeeze(1)
    preds = self.mlp(transformed)
    return preds
```

- Combines LSTM-encoded traits and transformer-processed text. - Returns predictions `[B, num_outputs]`.

6.7 Dynamic Output Handling

- To change number of outputs, adjust `num_outputs` in `CompatibilityModel`.
- Update slicing in forward pass and training loop for corresponding loss calculations.

---

7. Training Loop

- Loads dataset via `DataLoader`.
- Defines separate loss functions per output.
- Combines losses and optimizes with Adam.
- Early stopping monitors classification accuracy.

```
pred = final_model(trait.unsqueeze(1), text)
y_loss = criterion_yield(pred[:, 1], label)
c_loss = criterion_comp(pred[:, 0], comp_t)
loss = y_loss + 0.5 * c_loss
```

- For additional outputs, slice `pred[:, 2:]` and define corresponding loss.

---

8. Output Interpretation

- `pred[:, 0]`: logit for compatibility → apply `torch.sigmoid(pred[:, 0])` for probability.
- `pred[:, 1]`: regression target (yield impact, 0.0–1.0).
- Additional outputs correspond to added targets, in the order defined in the MLP final layer.

---

9. Summary of Key Concepts

- **Sequential model (`nn.Sequential`)**: simple way to chain layers with defined order.

• **LSTM**: handles sequential/numerical inputs capturing temporal dependencies.
• **Transformer**: captures relationships in text embeddings.
• **Embedding layer**: converts tokens into vectors.
• **ReLU**: non-linear activation.
• **Sigmoid**: converts logits to probabilities for binary classification.
• **BatchNorm**: normalizes activations to stabilize learning.
• **Dropout**: regularization to prevent overfitting.
• **Dynamic** `trait_dim` **and** `num_outputs` allow model adaptation to