PRISM-DNN User Guide

Introducing PRISM-DNN

PRISM-DNN (Pretrained pIxel-based Spatial-Temporal Deep Neural Network) is a deep learning model designed to make accurate predictions for each satellite image pixel by learning from its **spatial neighbors** and **temporal history**. Whenever your task involves predicting something for every pixel over space and time, PRISM-DNN can help.

This ability is especially valuable when you have:

- **Few ground-truth labels**
- **Massive archives of satellite images**

Why Is It Unique?

1. Trained to Understand Space and Time Together

Most models treat each pixel separately. PRISM-DNN is different: It learns how **what happens nearby (in space) and what happened before and after (in time)** affect the pixel you're looking at now.

2. Uses Large-Scale Unlabeled Data for Smarter Learning

When real-world labels are limited, PRISM-DNN first trains on **unlabeled satellite images** to learn general spatial–temporal patterns. This pretraining helps the model **make better use of limited labeled samples**, improving both **prediction accuracy** and **data efficiency** in low-resource scenarios.

3. Highly Transferable Across Tasks

Once pretrained, the model can be adapted to many tasks: retrieving different pollutants, various aerosol components, or other related variables—you name it.

Step-by-Step Tutorial for Using the Model

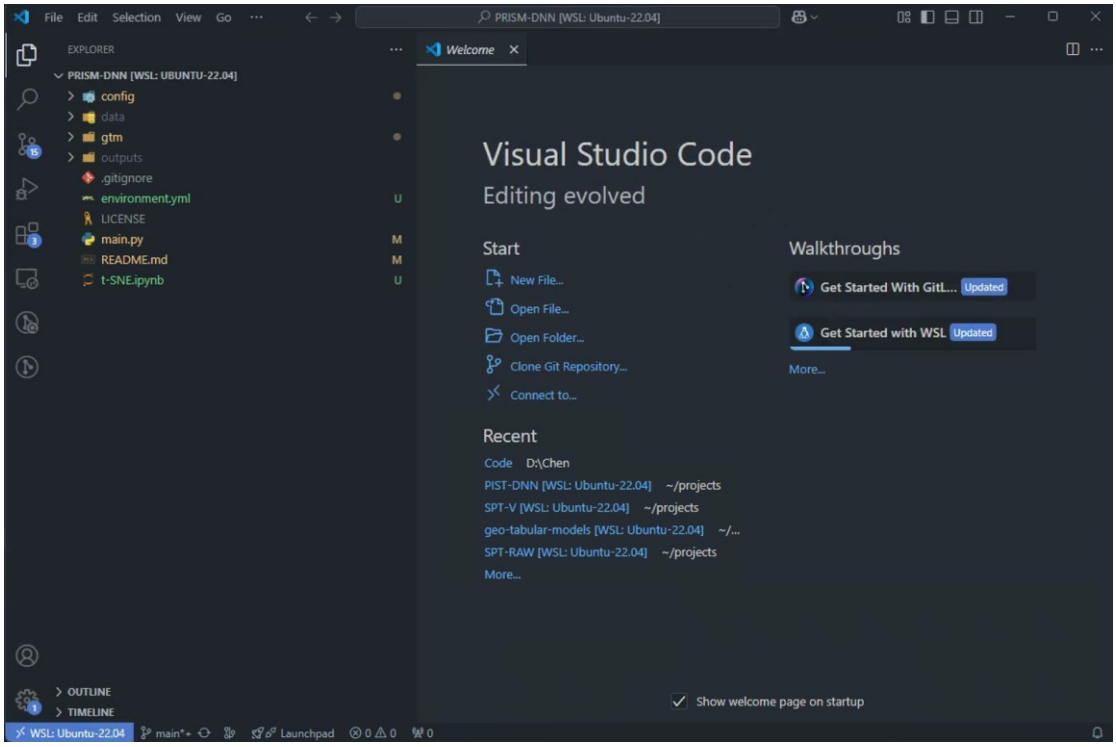
Step 1: Model Setup

1.1 Open the Project in VSCode

1st. **Download the project source code** from GitHub, then unzip or clone it to a folder on your computer.

2nd. **Open VSCode**.

3rd. In VSCode, click on *File* > *Open Folder* and select the project folder.

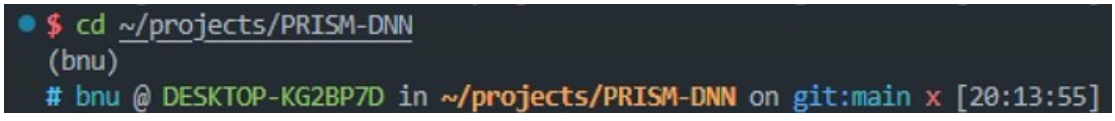


1.2 Set Up Python Environment (Python ≥ 3.8)

1st. Make sure you have installed **Miniconda** or **Anaconda** on your system.

2nd. Open the VSCode terminal (*View* → *Terminal*), and navigate to your PRISM-DNN project folder:

```
cd ~/projects/PRISM-DNN
```

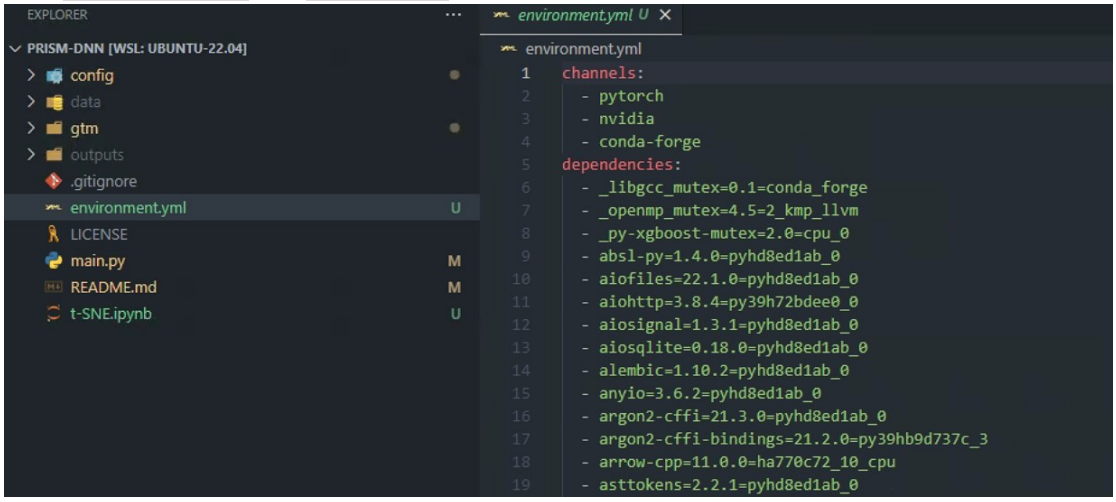


3rd. Create the Conda environment using the provided configuration file:

```
conda env create -f environment.yml -n bnu
```

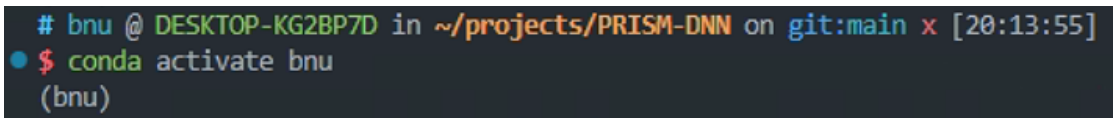
This will automatically create a new environment (You can replace *bnu* with any environment name you prefer) and install all necessary packages listed in the

environment.yml file. You can see that the *environment.yml* file includes libraries from *pytorch, nvidia,* and *conda-forge* channels, ensuring GPU support and compatibility.



4th. After installation, activate the environment (bnu):

```
conda activate bnu
```

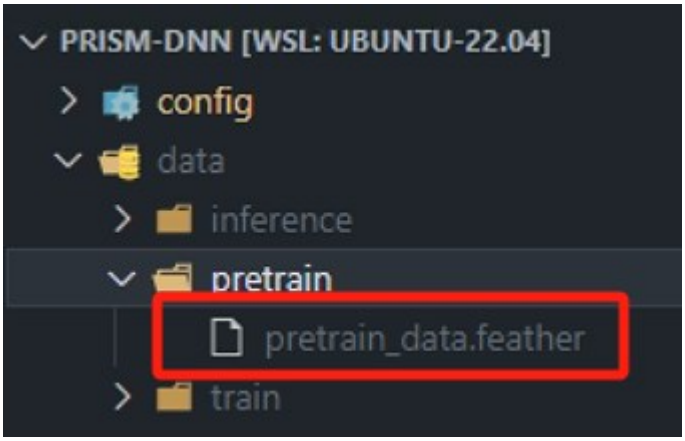


✅ Now you're ready to run the code.

Step 2: Pretraining the Model

2.1 Preparing Inputs for Pretraining

(e.g., pretrain_data.feather in data/pretrain/)



1st. Two Types of Input Variables (No Missing Values Allowed):

- **Categorical** variables (e.g. *Month*, *Season*, *Land cover type*)
- **Continuous** variables (e.g. *Satellite reflectance*, *Brightness temperature*)

2nd. For each pixel (called a **target pixel**), PRISM-DNN needs:

- The **current value** of the target pixel

- The values of its **8 surrounding neighbors** (in space, 3×3 window)
- The same set of variables from the **previous and next day** (in time)

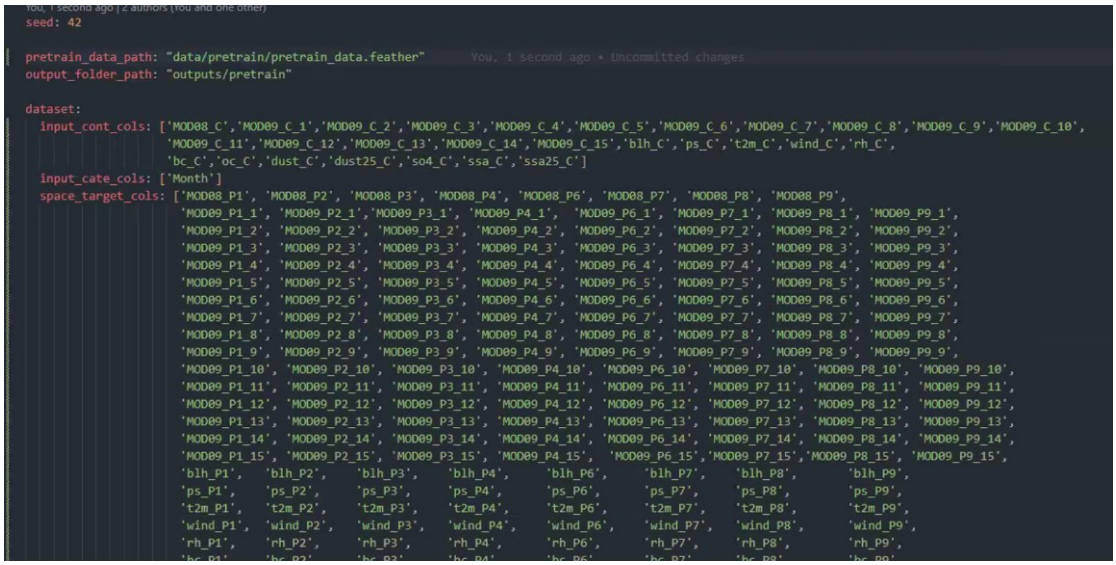
3rd. Export your final data table to **CSV, PKL, or Feather** format (recommended: Feather for large-scale data)

2.2 Running Pretraining

Once your input file (e.g. *pretrain_data.feather*) is ready, you can start pretraining the PRISM-DNN model. The model reads all its settings from a configuration file: *config/pretrain.yaml.*

1st. Locate the Configuration File

In your project folder, open: *config/pretrain.yaml*, You'll see a structure like this:



2nd. Key Configuration Sections

* These column names must exactly match the ones in your input file.

- *pretrain_data_path*:

  Path to the input file for pretraining. Should point to your *feather / .csv / .pkl* file.
- *output_folder_path*:

  Where to save model weights and logs after training. Suggested: leave as "*outputs/pretrain*" unless you want a custom folder.
- *input_cont_cols*:

  All **continuous input variables** from the central pixel at the current time step, used for learning spatial–temporal patterns (e.g., satellite bands, temperature, etc.).
- *input_cate_cols*:

  List of **categorical inputs**, such as Month, Land cover, Sensor ID, etc.
- *space_target_cols*:

  These columns contain values from the **8 neighboring pixels** around the target pixel (in a 3×3 window).
- *space_target_cols*:

These columns contain values from the **8 neighboring pixels** around the target pixel.

- *time_target_cols*:

  Columns used as temporal prediction targets, representing the values of the central pixel from the neighboring time steps (e.g. **previous and next day)**.

- Dataloader:

  Controls batch size and data loading speed:

```
batch_size: 1024        # Number of samples per batch (adjust based on memory)
pin_memory: True          # Speeds up GPU transfer
persistent_workers: True # Recommended for large datasets
```

- Model:

  Defines model architecture:

```
d_model: 128            # Embedding size
n_tf_head: 4           # Number of transformer heads
n_tf_layer: 4           # Number of transformer layers
p_tf_drop: 0.2          # Dropout rate in transformer
p_mlp_layer: 2          # MLP depth
p_mlp_drop: 0.2         # Dropout in MLP
lr: 0.0003              # Learning rate
```

- Callbacks:

  Tells the model when and how to stop:

```
save_top_k: 1              # Save the best model only
monitor: "valid_loss"       # Monitor validation loss
mode: "min"                # Lower is better
patience: 10               # Stop early if no improvement after 10 epochs
```

- Trainer:

  Sets how the model will run:

```
max_epochs: 200          # Max training epochs
```

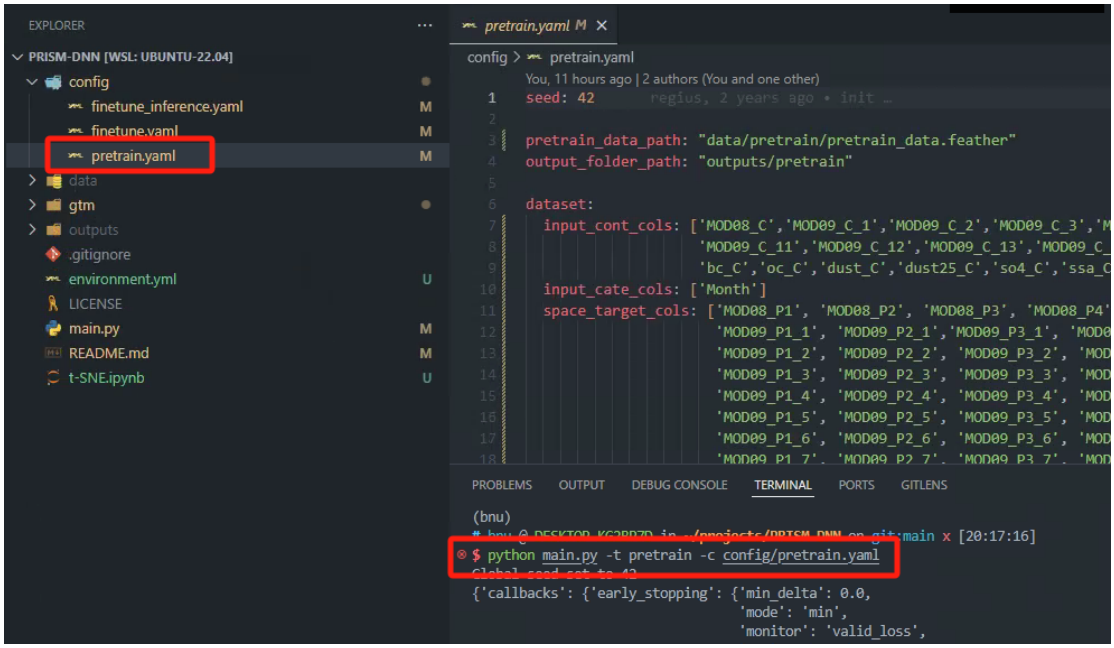
```
accelerator: "gpu"       # Use GPU (recommended)

devices: 1               # Number of GPUs (set to 1 unless using multi-GPU)
```

3rd. Run Pretraining in Terminal

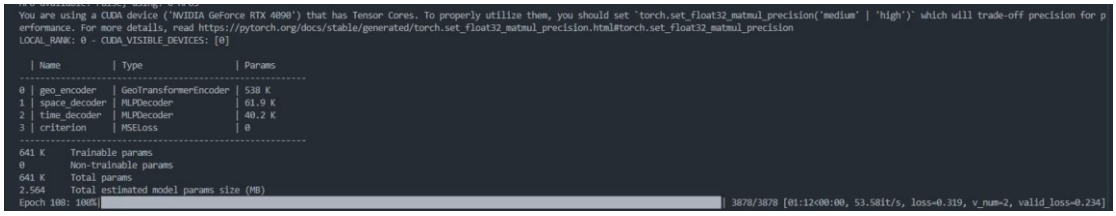
Make sure your Conda environment is activated, then run:

```
python main.py -t pretrain -c config/pretrain.yaml
```

The model will start reading the input file, building the model, and saving output to *outputs/pretrain/*.



Running screenshot:

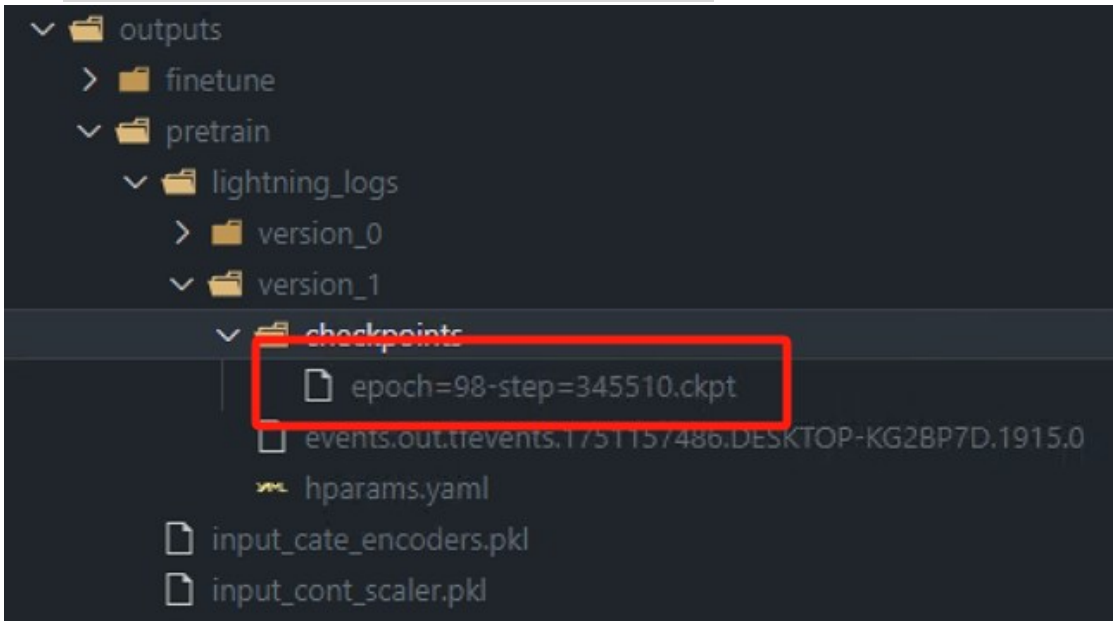


4th. Pretrain Finished

When the pretraining process completes, the following files will be generated automatically in the *outputs/pretrain/* directory:

- *input_cate_encoders.pkl* – encoders for categorical variables;
- *input_cont_scaler.pkl* – scalers for continuous variables;
- *lightning_logs/* – training logs and versioned model runs.

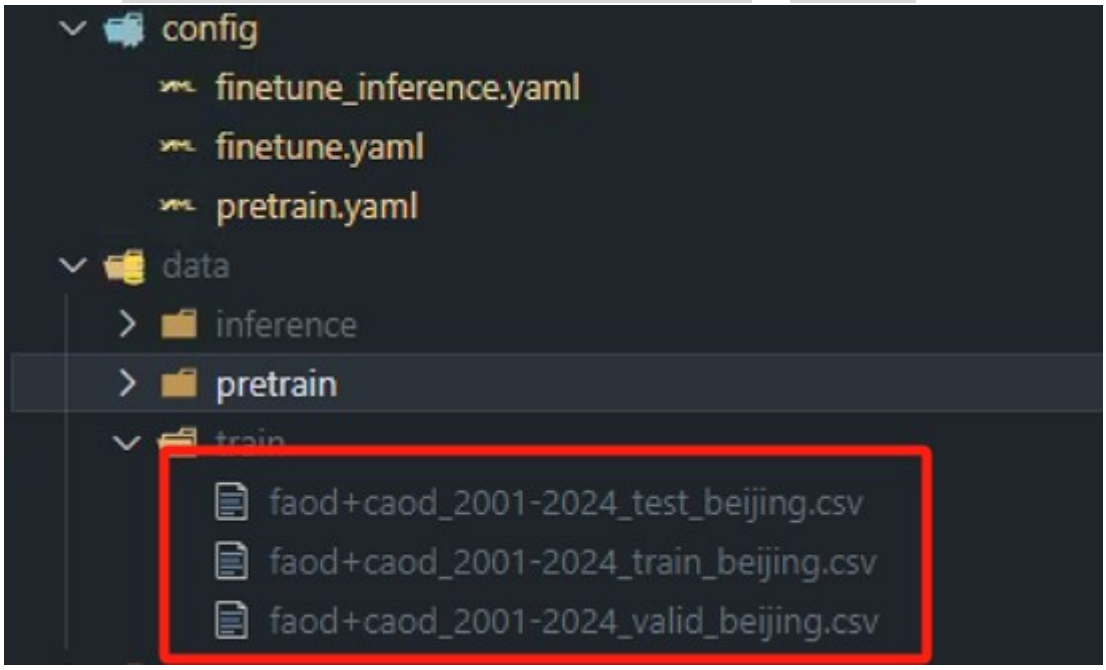
*Each time you run the pretraining process, a new subfolder (e.g., version_0, version_1, ...) will be created under outputs/pretrain/lightning_logs/. For each version, the best-performing model checkpoint is saved inside: *outputs/pretrain/lightning_logs/version_X/checkpoints/*



Step 3: Fine-tuning for a Target Task

3.1 Preparing Inputs for Fine-tuning

(e.g., faod+caod_2001-2024_[train/valid/test]_beijing.csv in data/train/)



1st. Two Types of Input Variables (No Missing Values Allowed):

- **Categorical** variables (e.g. *Month*, *Season*, *Land cover type*)
- **Continuous** variables (e.g. *Satellite reflectance*, *Brightness temperature*)

2nd. Fine-tuning requires a labeled dataset, where each sample includes both:

- **Input features** (from the central pixel only)
- **Target value** (the real variable you want the model to learn to predict, e.g. fAOD, cAOD, FMF)

*This step is different from pretraining. Here, we use a **standard supervised learning format,** similar to how models like LightGBM or Random Forest are trained.

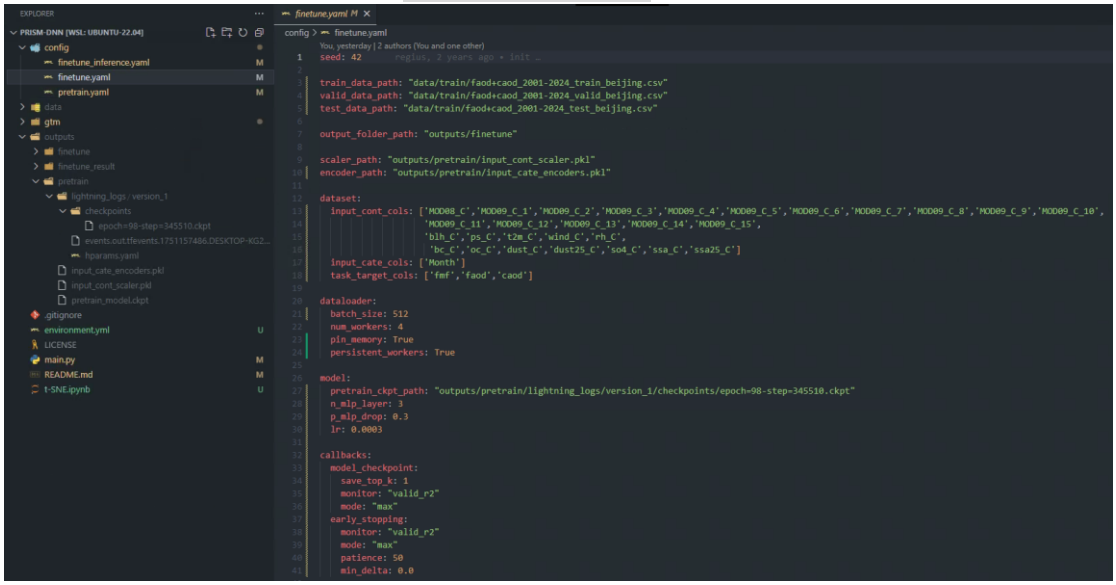
3rd. Export your final data table to **CSV** format.

3.2 Running Fine-tuning

Once you have prepared your labeled dataset and filled out the fine-tuning config file (*config/finetune.yaml*), you can run supervised training to adapt the pretrained model to your specific prediction task—such as estimating fAOD, cAOD, FMF or other variables.

1st. Locate the Configuration File

In your project folder, open: *config/ finetune.yaml*, You'll see a structure like this:



2nd. Key Configuration Sections

* These column names must exactly match the ones in your input file.

- *train_data_path*, *valid_data_path*, *test_data_path*:

  Path to the input file for fine-tuning. Should point to you *.csv* file.
- *scalar_path*, *encoder_path*:

  These files are generated during pretraining. By default, they are saved in the directory: *outputs/pretrain/*
- *input_cont_cols*:
  All **continuous input variables** (e.g. MODIS bands, temperature).
- *input_cate_cols*:
  All **categorical variables** (e.g. Month).
- *task_target_cols*:
  Your **label columns** (e.g. faod, caod, fmf).

- Dataloader:

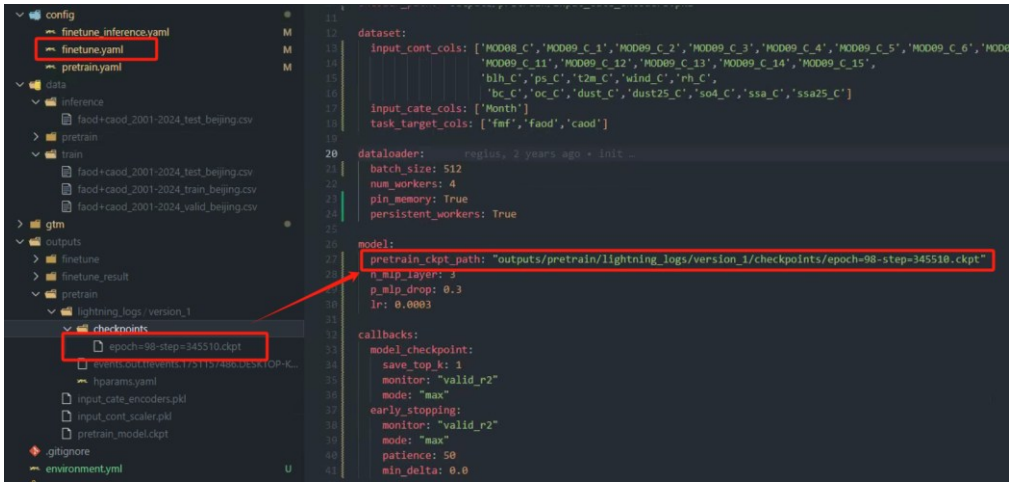
  Controls batch size and data loading speed:

```
batch_size: 1024          # Number of samples per batch (adjust based on memory)

pin_memory: True            # Speeds up GPU transfer

persistent_workers: True # Recommended for large datasets
```

- Model:

  Model structure and learning rate:

```
pretrain_ckpt_path:
outputs/pretrain/lightning_logs/version_1/checkpoints/epoch-98-
step=345510.ckpt        # replace to the best checkpoint from your pretraining

n_mlp_layer: 3          # the number of fully connected (MLP) layer

p_mlp_drop: 0.3         # Dropout in MLP

lr: 0.0003              # Learning rate
```



- Callbacks:

  Tells the trainer when to stop:

```
model_checkpoint:

  monitor: "valid_r2"    # Save model with best R² score

early_stopping:

  patience: 50            # Stop early if no improvement after 50 epochs
```

- Trainer:

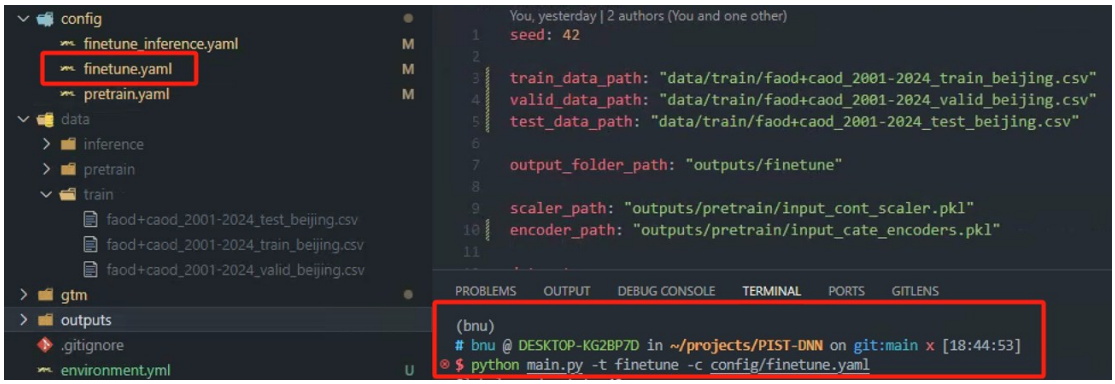
  Runs the training loop:

```
max_epochs: 150          # Max training epochs

accelerator: "gpu"       # Use GPU (recommended)

devices: 1               # Number of GPUs (set to 1 unless using multi-GPU)
```

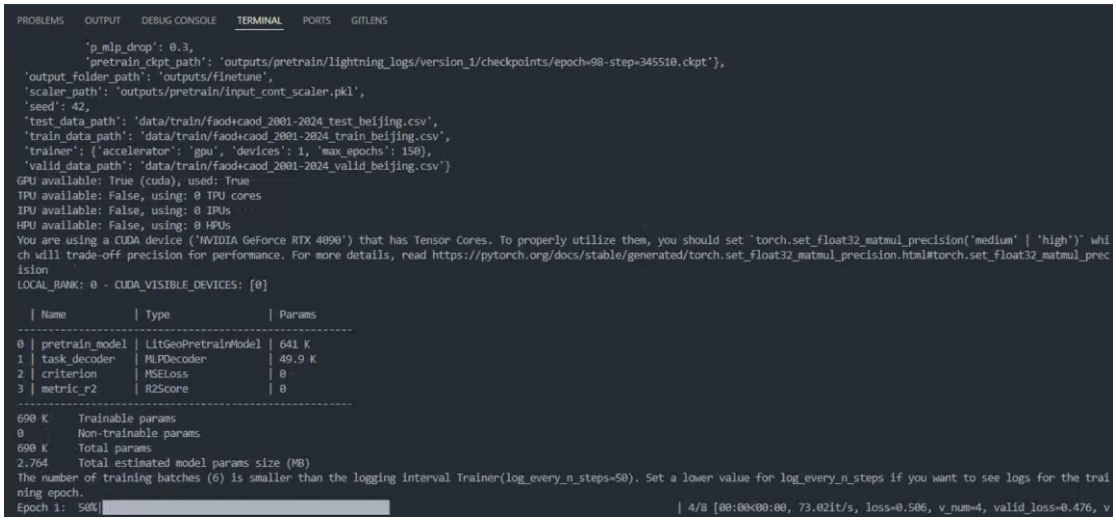
3rd. Run Fine-tuning in Terminal

Once your config is ready and your environment is active, run this command:

```
python main.py -t finetune -c config/finetune.yaml
```

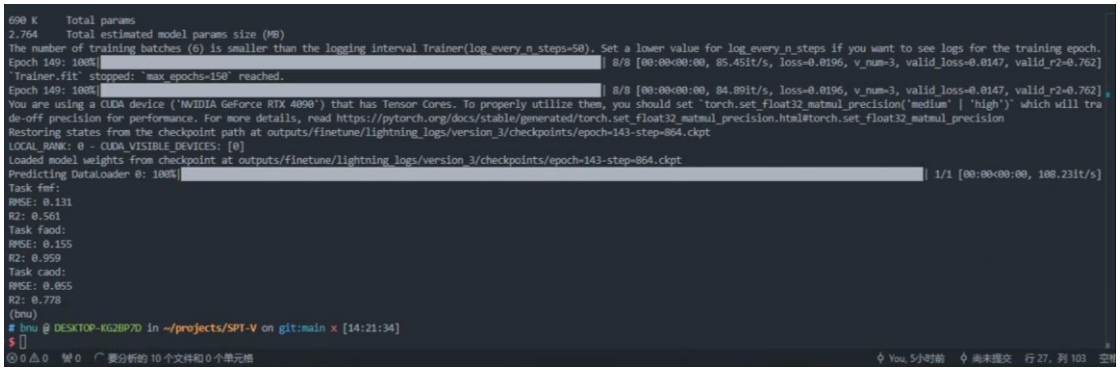


Running screenshot:



4th. Fine-tune Finished

When the fine-tuning process completes, the terminal will display a summary of the prediction performance for each target variable. This includes metrics like R^2 and RMSE, providing a quick overview of how well the model performed on the test set.



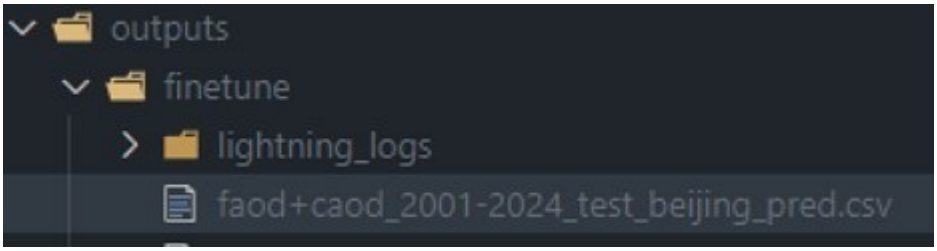
After that, the following files will be automatically saved in the *outputs/finetune/* directory:

- *faod+caod_2001-2024_test_beijing_pred.csv*

  This file includes predictions for each target (e.g., faod_PRED, caod_PRED, fmf_PRED); 128-dimensional pretrained feature embeddings for each sample (e.g., x_feature_0 to x_feature_127)

- lightning_logs/

  Training logs, including validation scores, learning rate, epoch summaries, and checkpoints.



That's it! The full training workflow is now complete.

Step 4: Global-Scale Inference without Ground Truth

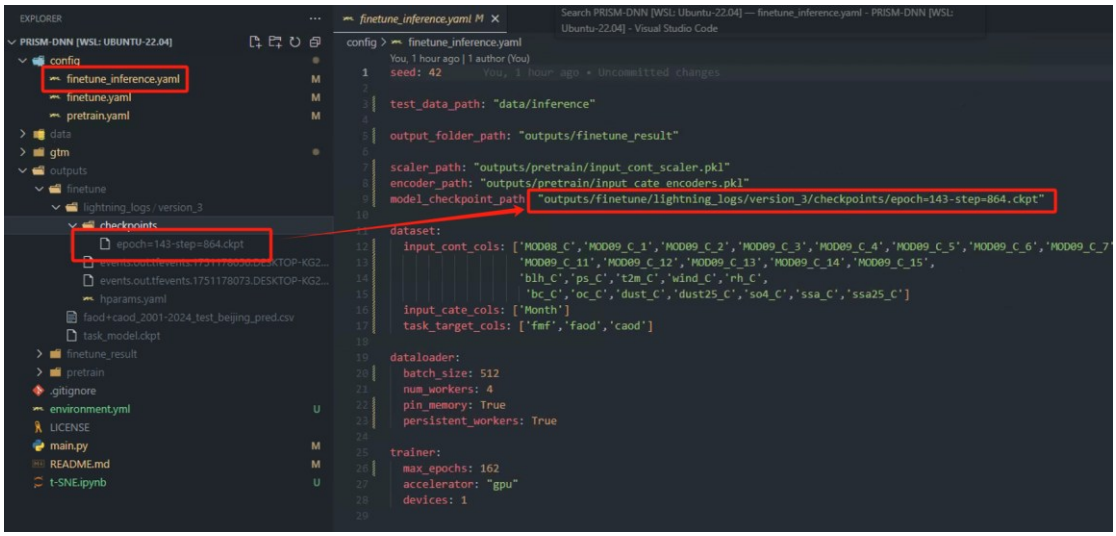
To perform inference in areas where no ground truth is available (e.g., pixels without station-based measurements), first place your input data in the *data/finetune_inference/* directory. This input file must have the **exact same column names and structure** as the one used during fine-tuning. Since there are no true target values, you should still include the target columns (e.g., faod, caod, fmf) but fill them with a constant value such as 0. These columns are required for compatibility with the model input format, but their values will not affect prediction results—as long as they are present and **non-empty**.

In your YAML configuration file (*finetune_inference.yaml*), set the *model_checkpoint_path* to the relative path of the best fine-tuned model checkpoint (e.g., *outputs/finetune/lightning_logs/version_3/checkpoints/epoch=143-step=864.ckpt*). The

settings under dataset: **must match exactly with those in the original fine-tuning** YAML to ensure consistent data preprocessing and feature encoding.

Then, execute the inference with the following command:

```
python main.py -t finetune_inference -c config/finetune_inference.yaml
```

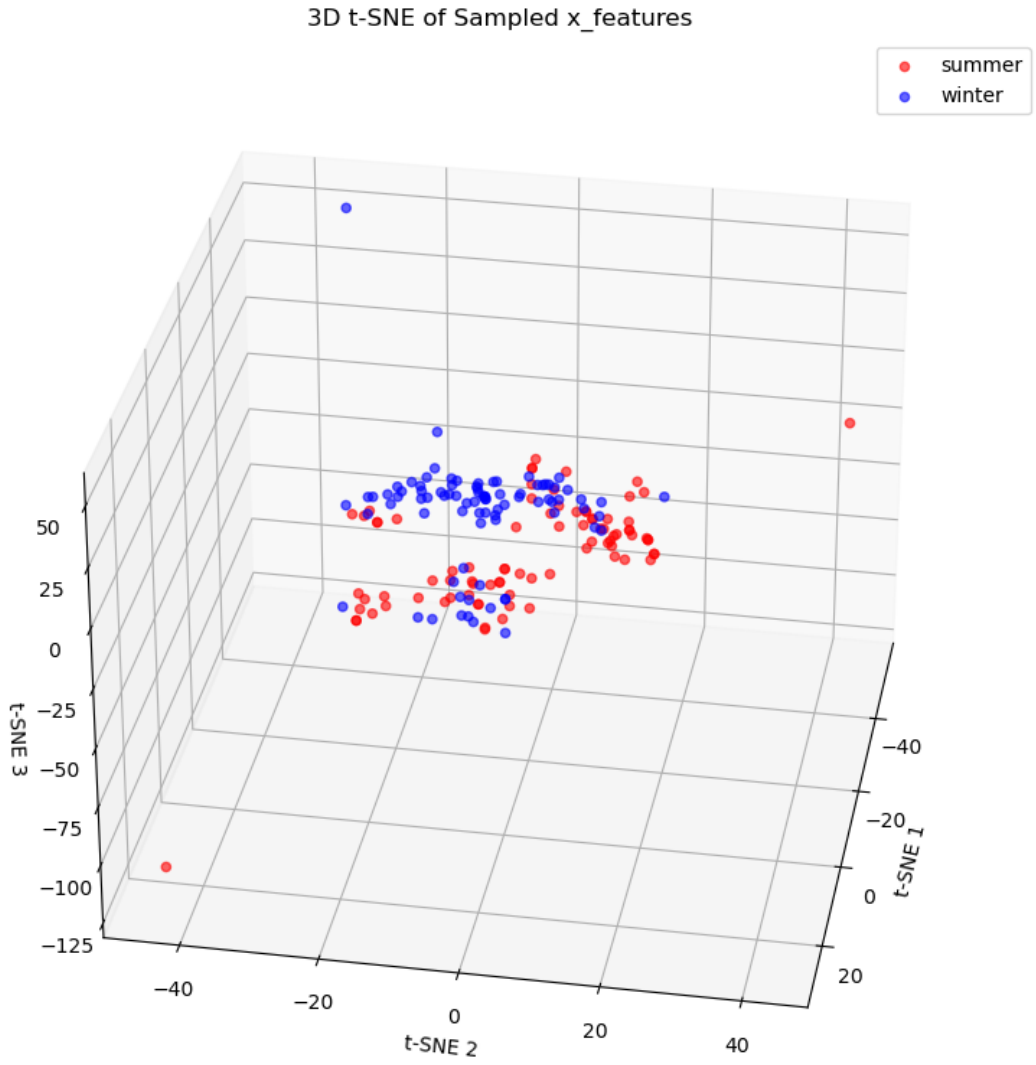


The model will output predictions for every pixel in the input file, enabling global-scale mapping without requiring any labeled data.

Step 5: Visualizing Pretrained Features with t-SNE

To better understand what the model has learned during pretraining, you can **visualize the 128-dimensional feature embeddings** from the test predictions using t-SNE, a commonly used dimensionality reduction technique.

We provide a Jupyter Notebook (*t-SNE.ipynb*) that demonstrates how to load the prediction results, extract pretrained features, and visualize them in 3D space.


3D t-SNE of Sampled x_features
summer
winter
t-SNE 3
50
25
0
−25
−50
−75
−100
−125
t-SNE 2
−40
−20
0
20
40
t-SNE 1
−40
−20
0
20