

SRSML24: STM Machine Learning Module

Steven R. Schofield

April 21, 2025

Overview

This module provides tools for machine learning analysis of scanning tunnelling microscopy (STM) data, including autoencoder models, clustering tools, and STM-specific preprocessing.

Getting the Code

Clone the repository from GitHub:

```
git clone https://github.com/srschofield/SRSML24.git
```

Installation

It is recommended to create a clean Python environment using `conda`. The following steps assume you are working on a macOS system with Apple Silicon:

```
# create and activate environment
conda create --name srsml24 python=3.8 -y
conda activate srsml24
```

```
# install packages
pip install -r requirements-macos.txt
```

Known Working Configuration

This module has been tested and is known to work with the following configuration on macOS 15.0.1 (Apple Silicon, M3 Pro chip):

Package	Version
python	3.8
tensorflow-macos	2.13.0
tensorflow-metal	1.0.1
numpy	1.24.3
pandas	2.0.3
matplotlib	3.7.5
scikit-learn	1.3.2
scipy	1.10.1
opencv-python	4.11.0.86
Pillow	10.4.0
joblib	1.4.2
jupyter	1.1.1
ipykernel	6.29.5
keras-core	0.1.5
spiepy	0.2.1
access2thetmatrix	0.4.4

Table 1: Verified package versions for macOS (Apple Silicon) environment

These packages can be installed using the `requirements-macos.txt` file. The Python version is critical: other versions may cause compatibility issues with TensorFlow or other packages on Apple Silicon.

Python Files

- `data_prep.py` – Functions for data preparation, including slicing STM images into windows and saving them in efficient formats.
- `model.py` – Defines convolutional autoencoder and UNET-style models.
- `utils.py` – Utility functions for loading/saving models, feature arrays, and results.

License

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0). You may share and adapt the material for non-commercial purposes, provided that appropriate credit is given and any derivatives are licensed under identical terms.

Parameter Summary

Parameter	Description
General	
job_name	Label for the run, it will be the folder name for output.
verbose	If True , enables more detailed print output.
Matrix data file processing	
flatten_method	Method used to flatten STM images before analysis. Options are 'none', 'iterate_mask', 'poly_xy'.
pixel_density	All images will be converted to this pixel density (px/nm).
pixel_ratio	Images that have ratio of fast/slow scan direction less than this will be discarded. Setting to 1 means only complete (square) images are kept.
data_scaling	Multiplicative factor for z-height data. Setting to 1.e9 means that the range 0–1 (used for training) corresponds to 1 nm.
Window generation	
window_size	Side length of square image windows (in pixels).
window_pitch	Spacing between adjacent windows during tiling.
Data saving	
(Should remain defaults but options can be useful for examining data manually.)	
save_windows	If True , saves image windows as .npy files (True).
together	If True , saves windows per image in a single file (True).
save_jpg	If True , saves full STM images as JPGs (False).
collate	If True , flattens directory structure into one folder. (False).
save_window_jpgs	If True , saves image windows as JPGs. (False)
Autoencoder	
model_name	Label used to save and load the trained autoencoder model.
batch_size	Number of windows per training batch.
buffer_size	Size of shuffle buffer.
learning_rate	Learning rate for the optimizer.
epochs	Number of training epochs.
Clustering	
cluster_model_name	Name used when saving the clustering model.
cluster_batch_size	Number of latent vectors per clustering batch.
cluster_buffer_size	Size of buffer for clustering shuffle.
num_clusters	Number of clusters to form using KMeans.
n_init	Number of initializations for KMeans.
max_iter	Max iterations for KMeans convergence.
reassignment_ratio	Fraction of centroids reassigned each step.
Image prediction	
predict_window_pitch	Window spacing during prediction step.
mtrx_train_data_limit	Max number of training MTRX files to use.
mtrx_test_data_limit	Max number of validation MTRX files to use.
train_data_limit	Limit on number of training windows.
test_data_limit	Limit on number of validation windows.

Step 1: Converting MATRIX STM Data Files

The initial step in utilizing the SRSML24 module involves converting raw STM data files from the Scienta Omicron MATRIX format into a standardized format suitable for machine learning analysis. This is accomplished using the `process_mtrx_files` function.

Function Overview

`process_mtrx_files(mtrx_paths, save_data_path, **kwargs)` is designed to batch process a list of MATRIX (`.mtrx`) files, performing the following operations:

- **Loading Data:** Reads each `.mtrx` file and extracts image data along with associated meta-data.
- **Preprocessing:** Applies flattening methods to correct for background variations and rescales images to a consistent pixel density.
- **Window Extraction:** Divides images into smaller windows of specified size and pitch, facilitating training of machine learning models.
- **Saving Outputs:** Stores processed windows and optional JPEG representations in a structured directory hierarchy under `save_data_path`.

This function ensures that STM data is preprocessed consistently, facilitating reliable training and evaluation of machine learning models within the SRSML24 framework. It will create a new directory called “windows” and subfolders under that with the names corresponding to the folders the matrix data was stored in (e.g., “training” or “testing”). Unless the “collate” variable is set, the windowed data will retain the full directory structure (dates, days, etc) of the matrix data. The individual windows for a given folder of matrix data are all stored within a single `.numpy` file, rather than separate `.numpy` files for each window, since this is much more efficient for data saving and retrieving. Two text files are also saved, one has the meta data (STM bias voltage, current, etc.). The other has the coordinates for each window in the dataset. This is useful since these are not uniform at two edges of the original image for the general case where the full image is not perfectly divided by the dimensions of the individual windows.

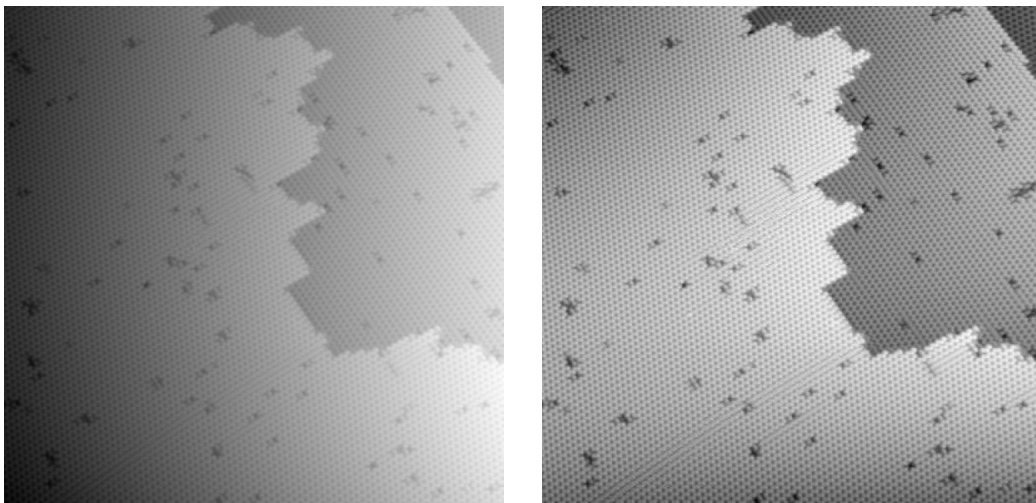


Figure 1: Typical scanning tunnelling microscopy (STM) image. (Left) raw data. (Right) After `poly_xy` background subtraction.



Figure 2: A sequence of 30×30 pixel windows extracted from the STM image in Fig. 1 with an 8 pixel pitch.

Step 2: Preparing Datasets and Training the Autoencoder

With windowed STM data in place, the next stage is to set up efficient TensorFlow data pipelines and train a convolutional autoencoder. This step handles the full model lifecycle, from data loading and batching to model training and saving the model and training history. These tools are provided by `model.py`, which we import as `m`, i.e., `import SRSML24.model as m`.

Preparing Datasets with `tf.data`

To begin, window data stored as `.npy` files in `windows_train_path` and `windows_test_path` are listed. If it is possible to choose a truncated subset of this list, e.g., to train on less data while optimising; this is done by setting the `train_data_limit` and `test_data_limit` parameters. These files are then loaded into TensorFlow `Dataset` objects for training and validation.

The function `m.create_tf_dataset_batched(file_paths, batch_size, buffer_size, window_size, is_autoencoder, shuffle)` constructs a pipeline that:

- **Loads image batches:** Reads window data from `.npy` files located at the specified paths.
- **Shuffles window order:** When `shuffle=True`, randomizes input order to enhance model generalization.
- **Batches the data:** Groups windows into batches of size `batch_size`.
- **Prefetches data:** Uses `prefetch(buffer_size)` to improve throughput by overlapping data preparation with model training.

- **Pairs inputs and targets:** For autoencoder training, each window is used as both the input and target.

Model Building and Training

Once the datasets are ready, the autoencoder can be constructed, compiled, and trained using the following workflow:

- **Model instantiation:**
 - Use `m.build_autoencoder(window_size, model_name)` to define the encoder–decoder architecture.
 - Call `model.summary()` to inspect the model structure.
- **Optimizer selection:**
 - On Apple Silicon/macOS systems, use `tf.keras.optimizers.legacy.RMSprop` for TensorFlow-metal compatibility.
 - On other systems, use the standard `tf.keras.optimizers.RMSprop`.
- **Compilation:** Compile the model with:
 - `loss='mean_squared_error'`
 - Metrics: `['mse', 'mae']`
 - Learning rate: from the `learning_rate` parameter.
- **Training:** Fit the model using:
 - `train_dataset` with `shuffle=True`
 - `validation_data=test_dataset`
 - The number of `epochs`
- **Saving outputs and diagnostics:**
 - Save the trained model using `m.save_model(..., model_train_time)`.
 - Plot and save the training history with `m.plot_history_from_file(...), m.save_history(...)`.

This produces a trained autoencoder model along with detailed training logs and performance plots, ready for use in clustering or anomaly detection tasks.

Layer (type)	Output Shape	Param #	Connected to
input (InputLayer)	[(None, 32, 32, 1)]	0	[]
conv1 (Conv2D)	(None, 32, 32, 32)	320	['input[0][0]']
drop1 (Dropout)	(None, 32, 32, 32)	0	['conv1[0][0]']
pool1 (MaxPooling2D)	(None, 16, 16, 32)	0	['drop1[0][0]']
conv2 (Conv2D)	(None, 16, 16, 64)	18496	['pool1[0][0]']
drop2 (Dropout)	(None, 16, 16, 64)	0	['conv2[0][0]']
pool2 (MaxPooling2D)	(None, 8, 8, 64)	0	['drop2[0][0]']
conv3 (Conv2D)	(None, 8, 8, 128)	73856	['pool2[0][0]']
drop3 (Dropout)	(None, 8, 8, 128)	0	['conv3[0][0]']
pool3 (MaxPooling2D)	(None, 4, 4, 128)	0	['drop3[0][0]']
bottleneck (Conv2D)	(None, 4, 4, 256)	295168	['pool3[0][0]']
up1 (UpSampling2D)	(None, 8, 8, 256)	0	['bottleneck[0][0]']
upconv1 (Conv2D)	(None, 8, 8, 128)	295040	['up1[0][0]']
skip1 (Concatenate)	(None, 8, 8, 256)	0	['upconv1[0][0]', 'conv3[0][0]']
conv4 (Conv2D)	(None, 8, 8, 128)	295040	['skip1[0][0]']
up2 (UpSampling2D)	(None, 16, 16, 128)	0	['conv4[0][0]']
upconv2 (Conv2D)	(None, 16, 16, 64)	73792	['up2[0][0]']
skip2 (Concatenate)	(None, 16, 16, 128)	0	['upconv2[0][0]', 'conv2[0][0]']
conv5 (Conv2D)	(None, 16, 16, 64)	73792	['skip2[0][0]']
up3 (UpSampling2D)	(None, 32, 32, 64)	0	['conv5[0][0]']
upconv3 (Conv2D)	(None, 32, 32, 32)	18464	['up3[0][0]']
skip3 (Concatenate)	(None, 32, 32, 64)	0	['upconv3[0][0]', 'conv1[0][0]']
conv6 (Conv2D)	(None, 32, 32, 32)	18464	['skip3[0][0]']
output (Conv2D)	(None, 32, 32, 1)	33	['conv6[0][0]']
=====			
Total params: 1162465 (4.43 MB)			
Trainable params: 1162465 (4.43 MB)			
Non-trainable params: 0 (0.00 Byte)			

Figure 3: Example UNET autoencoder model summary.

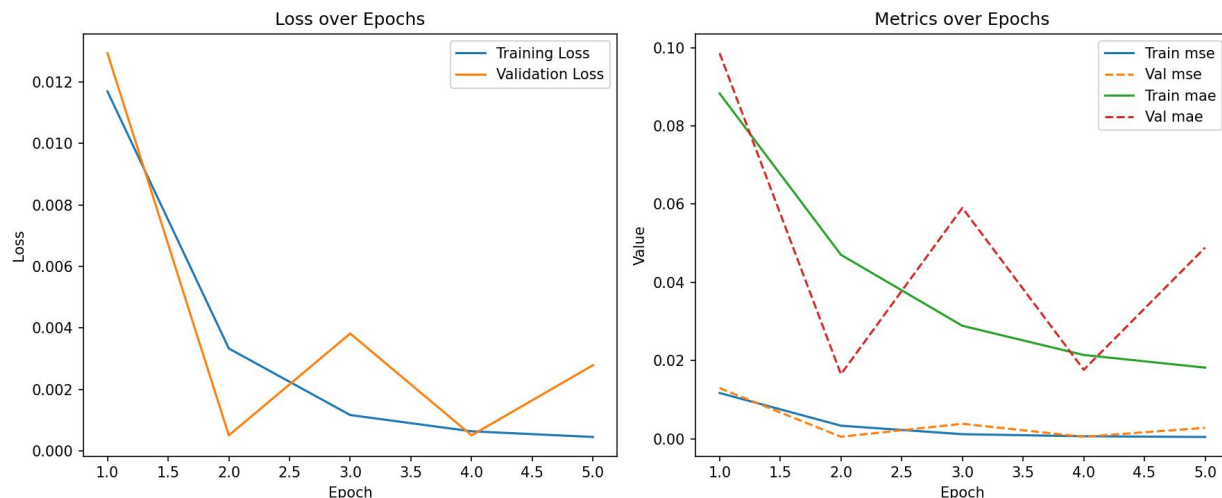


Figure 4: Example plot of training history.

Step 3: Extracting Latent Features and Clustering

Once the autoencoder has been trained, the next step is to extract compressed latent representations from its encoder and apply clustering to identify recurring patterns in the STM image windows. This step uses memory-efficient TensorFlow pipelines and supports large-scale datasets by processing in batches and saving intermediate results to disk.

Extracting Latent Features and Saving to Disk

The encoder is applied to the training dataset to compute latent vectors for each image window. These are saved to disk in a structured format using:

```
m.extract_latent_features_to_disk_from_prebatched_windows(
    autoencoder_model,
    train_dataset,
    latent_features_path,
    features_name='latent_features_train',
    return_array=False,
    verbose=False)
```

This function stores the latent features in `.npy` files under `latent_features_path`, with names beginning with `latent_features_train`. Saving in batches avoids memory bottlenecks. The saved latent features are later used for training a cluster model.

Preparing the Latent Feature Dataset

The saved latent feature files are listed and loaded into a TensorFlow data pipeline, ready for clustering:

```
latent_features_files, num_latent_files = dp.list_files_by_extension(
```



```
latent_features_path, 'numpy')

latent_features_dataset = m.create_latent_features_tf_dataset(
    latent_features_files,
    batch_size=cluster_batch_size,
    shuffle=True,
    shuffle_buffer_size=cluster_buffer_size)
```

Clustering with MiniBatch KMeans

Clustering is performed using the MiniBatch KMeans algorithm, which is scalable to large datasets and performs incremental updates to the centroids:

```
cluster_model, convergence_history = m.train_kmeans(
    latent_features_dataset,
    batch_size=cluster_batch_size,
    num_clusters=num_clusters,
    n_init=n_init,
    max_iter=max_iter,
    reassignment_ratio=reassignment_ratio)
```

The training history includes convergence diagnostics and can be plotted to assess stability across epochs.

Saving the Clustering Model

The trained clustering model is saved using:

```
m.save_cluster_model(cluster_model, cluster_model_path, model_name=cluster_model_name)
```

This ensures that cluster labels can be reused for prediction, visualization, or interpretation in downstream analysis.

Outputs

This step produces the following key results:

- **Latent feature vectors:** Encoded representations saved as batched `.numpy` files.
- **Trained KMeans model:** Saved clustering model for reuse or deployment.
- **Convergence history:** Optional diagnostics on clustering stability and progress.

Figure 5: Cluster assignments in latent space using MiniBatch KMeans. Each color represents a distinct pattern or feature set discovered from STM data.

Step 4: Making Cluster Predictions on New STM Images

In the final step, trained models are applied to new STM image data. This allows the latent features learned by the autoencoder, and the clustering patterns identified via KMeans, to be mapped onto previously unseen scans—highlighting recurring patterns, regions of interest, or anomalies.

Preparing Prediction Data

STM images in the Scienta Omicron MATRIX format are first preprocessed into windowed image data using the same function as in Step 1:

```
dp.process_mtrx_files(  
    mtrx_predict_file_list,  
    job_data_path,  
    flatten_method = flatten_method,  
    pixel_density = pixel_density,  
    pixel_ratio = pixel_ratio,  
    data_scaling = data_scaling,  
    window_size = window_size,  
    window_pitch = predict_window_pitch,  
    save_windows = save_windows,  
    save_window_jpgs = save_window_jpgs,  
    save_jpg = save_jpg,  
    together = together,  
    collate = collate,  
    verbose = verbose)
```

The windows and coordinate files are saved in the same structure as the training set, enabling seamless integration with the trained models.

Loading Trained Models

Both the autoencoder and clustering model are reloaded from disk:

```
autoencoder_model = m.load_model(model_path, model_name=model_name)  
cluster_model = m.load_cluster_model(cluster_model_path, model_name=cluster_model_name)
```

Cluster Prediction Workflow

For each preprocessed STM image:

1. **Window data and coordinates** are loaded from disk.
2. A full image is reconstructed from the original window tiles.
3. A TensorFlow data pipeline is built to pass the windows through the autoencoder.
4. The autoencoder encoder generates latent features for each window.
5. The clustering model assigns each window to a cluster.
6. Cluster labels are mapped back to image coordinates and rendered as a cluster map.
7. The result is optionally padded to match the original image dimensions.
8. A side-by-side visualization of the raw and clustered image is saved to disk.

This workflow is handled in the following loop:

```

for prediction_file, coords_file in zip(predict_data_files_list, image_windows_coordinates_file_
...
predict_dataset = m.create_tf_dataset_batched(...)
latent_features = m.extract_latent_features_to_disk_from_prebatched_windows(
    autoencoder_model, predict_dataset, '', return_array=True)
cluster_predictions = cluster_model.predict(latent_features)
cluster_img = dp.reconstruct_cluster_image(...)
cluster_img = ut.pad_cluster_image(...)
m.display_reconstructed_and_cluster_images(...)

```

Output

Each prediction run saves the following outputs in the `predictions_path` folder:

- **Cluster map:** A color-coded image showing predicted cluster labels for each image region.
- **Overlay (optional):** A composite image showing the raw STM image overlaid with transparency or color labels.
- **Reconstruction log:** Coordinate-based window reconstructions for reproducibility.

Figure 6: Cluster prediction output. (Left) reconstructed raw STM image. (Right) predicted cluster labels, highlighting nanoscale structure.

This concludes the SRSML24 processing pipeline: from raw data preprocessing, through autoencoder training and latent-space clustering, to visualization of predictions on new STM images.