Powder Stream Distribution Analysis

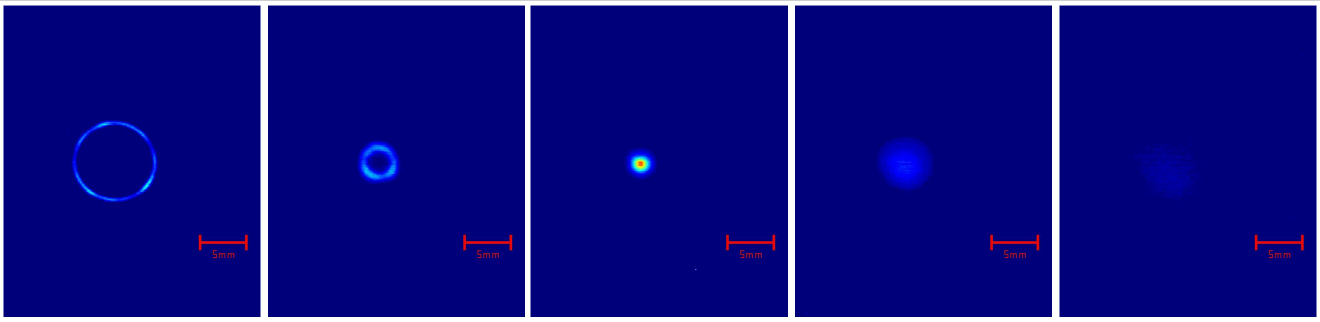
- CNNs: models/cnn_encoder_decoder.py
- used model on process parameters: models/process_parameter_models.py

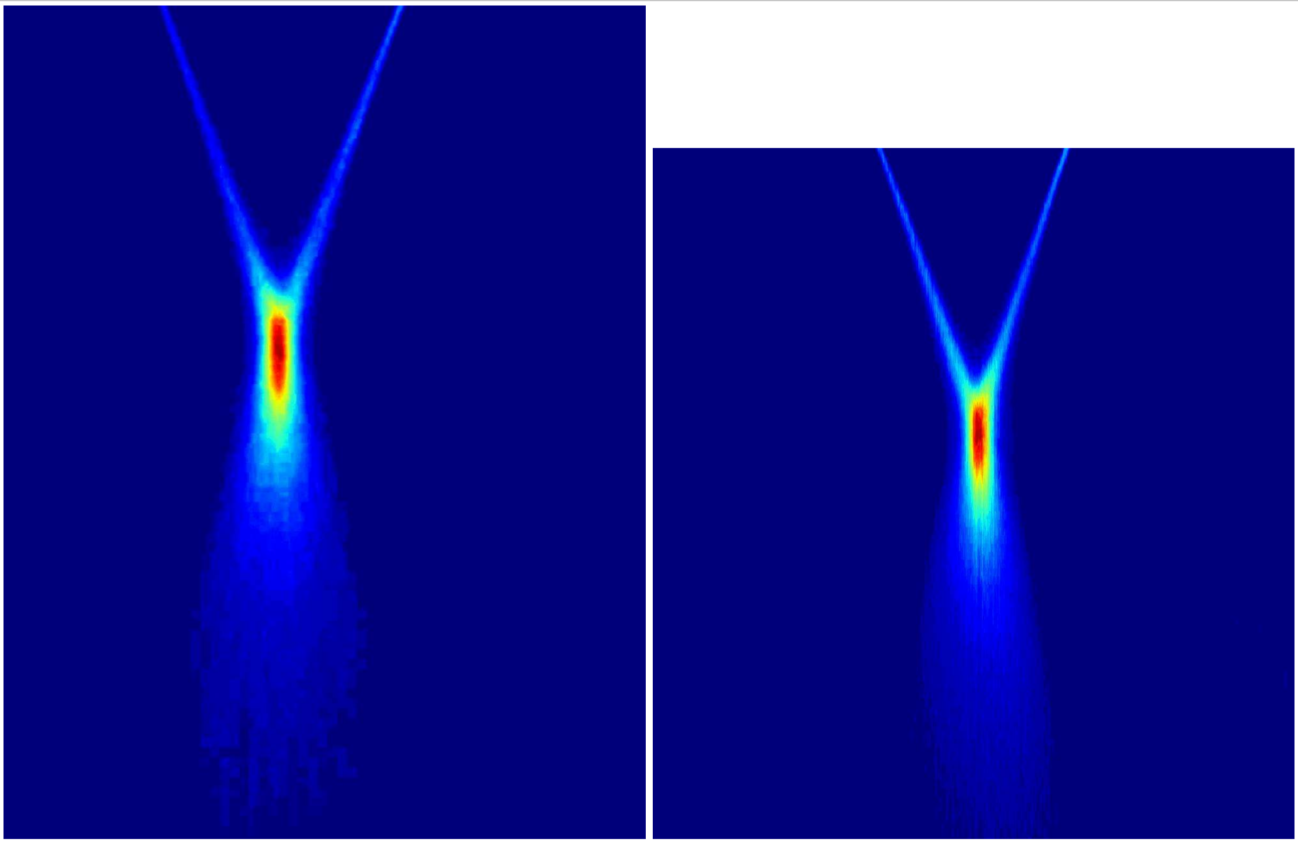
Table of Contents

Data Overview

Understanding the Powder Stream Data

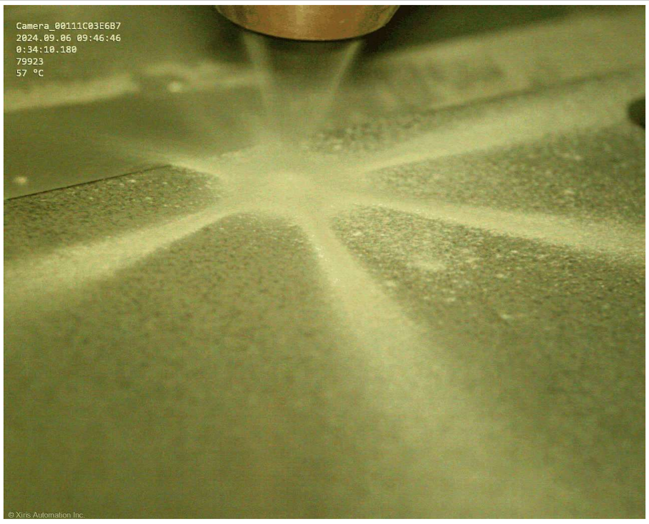
The powder stream evolves as it travels from the nozzle (top) to the substrate (bottom). To understand this evolution, we can visualize different cross-sections of the 3D volume:

XY Cross-Sections (Horizontal Slices at Different Heights)

Progression from top (near nozzle) to bottom shows how the powder distribution changes
Vertical Cross-Sections


Vertical cross-sections show the full powder stream trajectory

Below are two videos showing the powder stream and deposition process:

Powder Stream



Deposition Process (Example)

Note: The deposition video is provided as an example to give a general idea of how the process looks. This project focuses specifically on analyzing and predicting the powder stream distribution shown in the left video (without any melting).

Related Research: For more information on laser-powder alignment and its influence on part properties in high-speed directed energy deposition, see the research paper included in `_paper/Platz et al. - 2024 - Investigation on different laser-powder alignments...`.

3D Volume Data

The raw data consists of 3D numpy arrays stored at `/scratch/schuermm-shared/powder-spy-arrays`. These volumetric captures represent the powder stream distribution at various distances from the nozzle.

The 3D volumes are organized as follows:

- The vertical dimension (Z-axis) represents the distance from the nozzle
- The X and Y dimensions represent the horizontal cross-section of the powder stream
- Higher slice indices correspond to positions closer to the nozzle (top of the volume)

Slice Extraction

The 3D volumes were processed into 2D slices using the `_elwe_stuff/array_slice_extractor.py` script, which extracts slices along the vertical dimension (dim 2). The extracted slices are stored in `/scratch/schuermm-shared/powder-spy-arrays/slice_data_dim2`.

```
# Example of how slices are extracted (from array_slice_extractor.py)
for slice_idx in range(array.shape[2]):
    slice_data = array[:, :, slice_idx]  # Taking slice along dim 2
    slice_filename = f"slice_{slice_idx:04d}.npy"
    slice_path = os.path.join(array_output_dir, slice_filename)
    np.save(slice_data, slice_path)
```

For efficient training, the slice data has been consolidated into an optimized HDF5 format using `_elwe_stuff/create_h5_dataset.py` and `_elwe_stuff/rechunk_it.py`.

Important Note: Since the top of the 3D volume corresponds to where the powder stream starts (closest to the nozzle has the highest index), we need to process the slices in reverse order when modeling the powder flow from top to bottom. This ordering is handled in the `data_modules/batch_slice_data_module.py` file:

```
# From batch_slice_data_module.py
def _create_slice_pairs(self, array_dirs: List[Path]) -> List[Tuple[str, str]]:
    """Create consecutive slice pairs from array directories.
    Arrays are sorted in reverse order because the highest slice index
    corresponds to the first slice at the top of the volume.
    """
    pairs = []
    for array_dir in array_dirs:
        # Get all slice files in this array directory
        slice_files = sorted(array_dir.glob("slice_*.npy"), reverse=True)
        # Create consecutive pairs
        for i in range(len(slice_files) - 1):
            current_slice = str(slice_files[i])
            next_slice = str(slice_files[i + 1])
            pairs.append((current_slice, next_slice))
    return pairs
```

Project Structure

```
powder-stream-analysis/
├── README.md                        # This file
├── run.py                          # Main training script
├── run_fbk.sh                      # SLURM script for FBK partition
├── run_v100.sh                     # SLURM script for V100 GPU
├── visualization_callback.py        # Comprehensive visualization callback
│
├── models/                         # Model architectures
│   ├── __init__.py
│   ├── base_slice_predictor.py     # Base model class
│   ├── cnn_encoder_decoder.py      # Advanced CNN architectures
│   └── cnn_simple.py               # Simple CNN baseline
│
├── data_modules/                   # Data loading and processing
│   ├── __init__.py
│   ├── batch_slice_data_module.py  # Optimized batch data loader (current)
│   └── single_slice_data_module.py # Original single-slice loader (legacy)
│
├── *elwe*stuff/                    # Data preprocessing utilities
│   ├── array_slice_extractor.py    # Extract 2D slices from 3D volumes
│   ├── create_h5_dataset.py          # Convert slices to HDF5 format
│   └── rechunk_it.py                 # Optimize HDF5 chunking for performance
│
├── _paper/                           # Related research papers
│   └── Platz et al. - 2024 - Investigation on different laser-powder alignments...
│
└── _assets/                          # Documentation assets
    ├── powder_spy_examples/           # Example slice visualizations
    ├── process_videos/                # Process demonstration videos
    └── visualization_callback_example/ # Example visualization outputs
```

Data Processing Pipeline

The data pipeline is managed by the `BatchOptimizedDataModule` class (`data_modules/batch_slice_data_module.py`), which provides significant performance improvements over the original single-slice loader:

Key Features:

- **HDF5 Backend**: Uses optimized HDF5 storage for fast batch loading
- **Batch Loading**: Supports `__getitems__` for efficient multi-slice loading
- **Memory Optimization**: Rechunked data for optimal I/O performance
- **Flexible Encoding**: Supports position and slice index encoding
- **Comprehensive Splitting**: Handles train/validation/test splits with configurable ratios

Pipeline Steps:

1. **Data Consolidation**: Individual slice files → HDF5 dataset (`create_h5_dataset.py`)
2. **Optimization**: Rechunk HDF5 for optimal access patterns (`rechunk_it.py`)
3. **Pair Creation**: Generate consecutive slice pairs for training
4. **Batch Loading**: Efficient multi-slice loading during training
5. **Normalization**: Data normalization and augmentation

Model Architecture

Available Models

The project includes multiple model architectures in the `models/` directory:

- **SimpleCNN** (`cnn_simple.py`): Baseline CNN for direct slice-to-slice prediction
- **LatentCNN** (`cnn_encoder_decoder.py`): Encoder-decoder with latent space representation
- **VariationalCNN** (`cnn_encoder_decoder.py`): Variational autoencoder for probabilistic predictions

Model Selection

Models are selected via command-line argument:

```
python run.py --model SimpleCNN        # Baseline model
python run.py --model LatentCNN        # Latent space model
python run.py --model VariationalCNN   # Variational model
```

Input/Output Specifications

- **Input Size**: 1024×1024 pixels (configurable)
- **Channels**: Configurable based on encoding options
    - Base: 1 channel (grayscale image)
    - +Position encoding: +2 channels (x, y coordinates)
    - +Slice index: +1 channel (normalized slice position)
- **Output**: Same dimensions as input (next slice prediction)

Prediction Methods

Direct Prediction

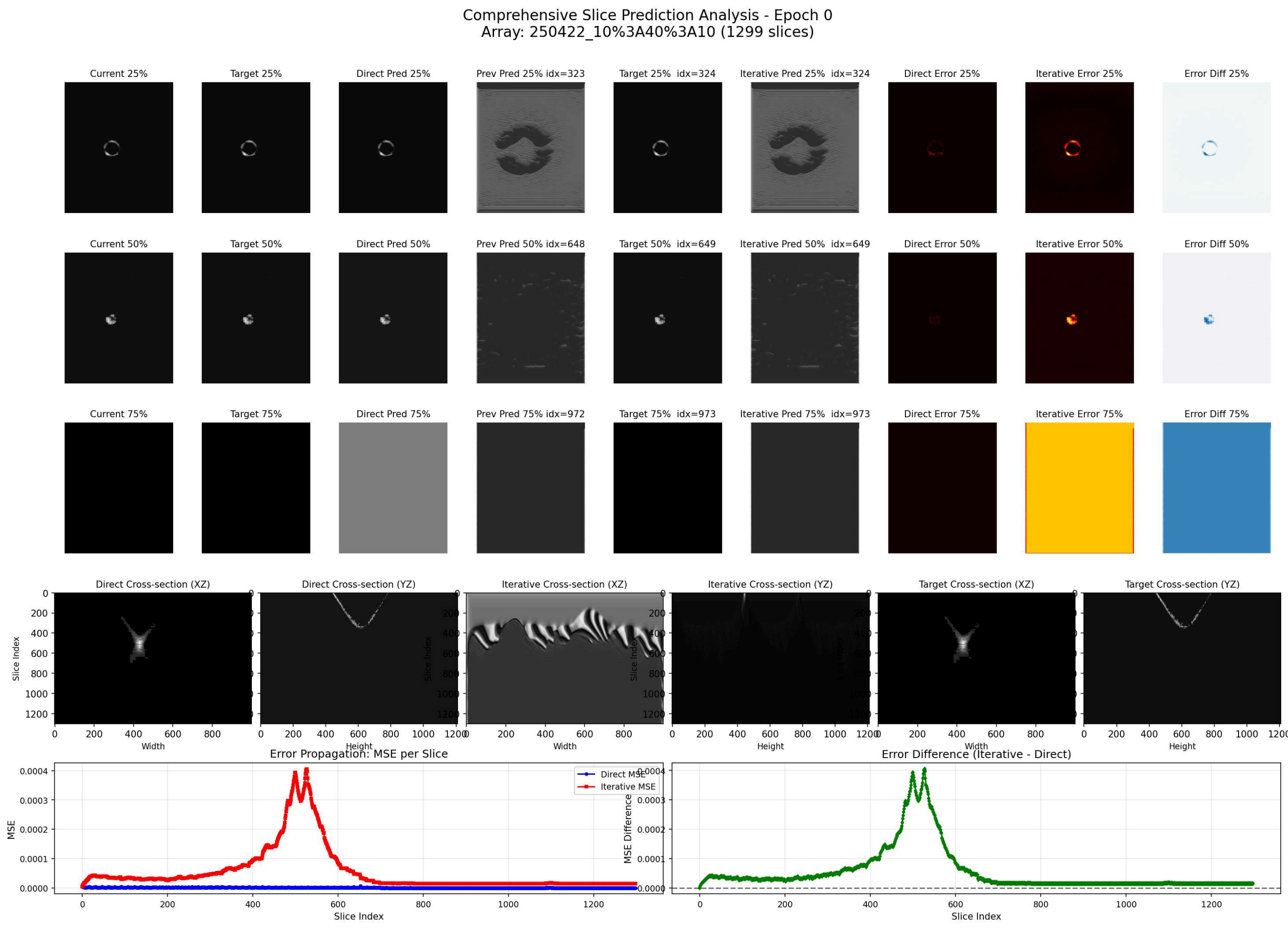
- Each slice is predicted directly from ground truth input
- More accurate for individual predictions but doesn't model propagating errors
- Used for pre-training and baseline evaluation

Iterative Prediction

- Uses previous predictions as inputs for subsequent predictions
- Subject to error propagation over multiple predictions

Visualization and Analysis

The `VisualizationCallback` provides comprehensive analysis of model predictions:



Running the Code

Basic Training

```
python run.py \
    --batch_size 16 \
    --learning_rate 0.001 \
    --max_epochs 100 \
    --model SimpleCNN \
    --num_workers 8
```

Advanced Options

```
python run.py \
    --model LatentCNN \
    --encode_positions \              # Add position encoding
    --encode_slice_index 1500 \       # Add slice index encoding
    --batch_size 32 \
    --max_epochs 200
```

SLURM Execution

For cluster execution, use the provided SLURM scripts:

```
# FBK partition
sbatch run_fbk.sh

# V100 GPU
sbatch run_v100.sh
```

Development Roadmap

Completed

- ✅ **Basic CNN Architecture**: Direct slice-to-slice prediction
- ✅ **Data Pipeline Optimization**: HDF5-based batch loading
- ✅ **Comprehensive Visualization**: Multi-method comparison and analysis
- ✅ **Multiple Model Architectures**: Simple, Latent, and Variational CNNs
- ✅ **Performance Optimization**: Efficient data loading and GPU utilization

Future Development

1. **LSTM Integration**: Incorporate recurrent architecture to better capture temporal dynamics
2. **Enriched Latent Space**:
    - Integrate process parameters (powder feed rate, gas flow rates)
    - Material type and mixing percentages for multi-material experiments