Report on the Final Project – Deep Learning

Soumya Swaraj Mondal

In this project, we tried to implement a model that would increase reliability to our lab operation where Aerosol Jet Printing is done.

Workflow

Aerosol Jet Printing **(AJP)** is an advanced additive manufacturing technique used to deposit fine patterns of functional materials onto a variety of substrates. It's especially popular in **printed electronics**, **biomedical devices**, and **flexible hybrid electronics** because it allows high-resolution printing of conductive inks, polymers, and even biological materials.

Work Flow:

1. **Atomization**: The ink (liquid containing nanoparticles or functional molecules) is aerosolized—converted into a mist of tiny droplets—using either ultrasonic or pneumatic methods.
2. **Aerosol Transport**: A carrier gas (usually nitrogen) transports the aerosol to the print head.
3. **Focusing Nozzle**: A sheath gas surrounds and focuses the aerosol stream into a tight jet (as small as 10–100 micrometers wide).
4. **Deposition**: The focused jet is directed onto the substrate, which can be flat, curved, or irregular.

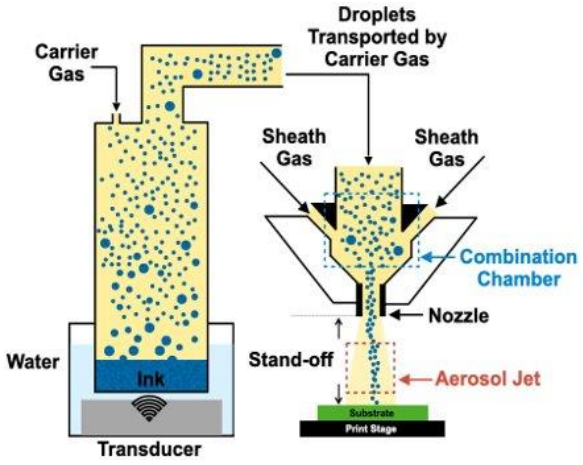

Fig: Aerosol Jet Printing work flow

Attributes:

- **High resolution**: Can print features as small as 10 µm.
- **Non-contact**: Good for delicate or 3D surfaces.
- **Versatile materials**: Can handle conductive inks, dielectrics, biomaterials, etc.
- **3D capability**: Can build up vertical structures layer-by-layer.

Fig: Batches of AJP printed lines

Fig: Close look on AJP printed conductive line

These features have made this process quite lucrative, leading to numerous multifaceted applications:

- Printing conductive traces for flexible electronics.
- Sensors and antennas on curved surfaces.
- Biomedical devices (e.g., drug delivery microstructures).
- Photovoltaics and OLED displays.

Problem Statement:

We can create the AJP-based print according to the shape and the size given as the input. Even though it might look perfect in terms of the shape, **NO TWO PRINTS ARE IDENTICAL!** In terms of the physical Properties, the deposit morphology is not ideal and can vary from the intended design. This variation in the morphology can lead to differences in its properties

(conductivity, voltage capacity etc.) and reduce the reliability in the process itself.

Our model contributes to the under-mentioned things in the domain of Direct Ink Printing:

- Reproducibility
- Move from Trial and Error based approach to Data Driven Approach.
- Increase Dataset balance and quality.

These contributions were made by our work through **CGAN (Conditional Generative Adversarial Network) .**

Personal Contribution:

The approach we took to build this model and solve the issues we faced while working was **Discussion.** We discussed as a group and tried to build different parts and solve the issues with them. After data augmentation, one of the first things was to introduce this data in a suitable format to the training setup. After this, I designed **Discriminator[1].** The key areas I focused while making it:

- **Conditional Discrimination**: The Discriminator is a conditional GAN component, using label embeddings (ATM, CR, FR, PS) to condition its classification on printing parameters, aligning with the AIP-cGAN's design.
- **Feature Extraction and Classification**: The convolutional layers downsample the input, extracting spatial features, while the final linear and Sigmoid layers provide a binary real/fake classification.
- **Training Stability**: Batch Normalization and Dropout enhance training stability and prevent overfitting, while LeakyReLU ensures gradient flow.

[1] https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/cgan/cgan.py

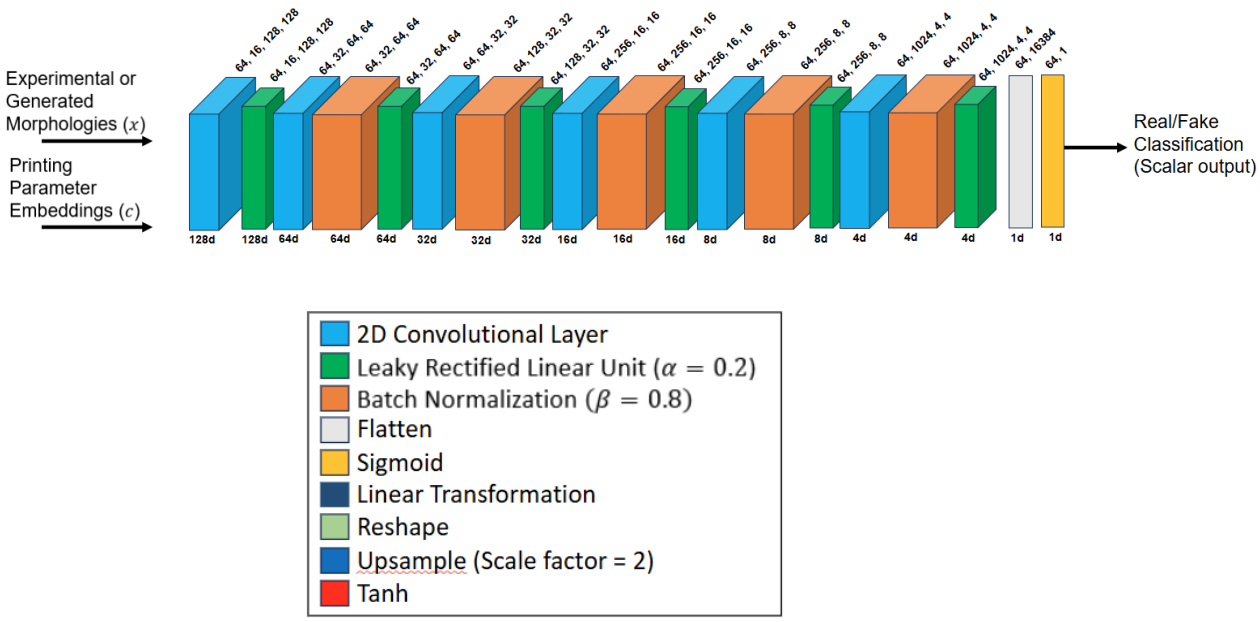

Fig: Discriminator architecture in CGAN

Detailed Analysis:

Input Channels (1 + 4):

- The first nn.Conv2d(1 + 4, 16, ...) indicates the input has *1+4=51 + 4* = 51+4=5 channels:
    - 1 channel for the grayscale image (e.g., 256x256).
    - 4 channels from the embedded label vectors (ATM, CR, FR, PS), which are concatenated later.

Convolutional Layers (nn.Conv2d):

- Five nn.Conv2d layers with parameters:
    - Input channels, output channels (16 → 32 → 64 → 128 → 256), kernel size (3x3), stride (2), padding (1).
    - Stride=2 and padding=1 downsample the spatial dimensions by half each time (e.g., 256x256 → 128x128 → 64x64 → 32x32 → 16x16 → 8x8), while increasing the number of feature maps.
    - Output shape after the last convolution: [batch_size, 256, 4, 4]

LeakyReLU (nn.LeakyReLU(0.2, inplace=True)):

- Applies a Leaky ReLU activation with a slope of 0.2 for negative values, introducing non-linearity and allowing gradient flow for negative inputs.
- inplace=True modifies the input tensor directly, saving memory.

Batch Normalization (nn.BatchNorm2d):
- Normalizes the output of each convolutional layer (except the first) across the batch, stabilizing training with a momentum parameter (default or implied).

Dropout (nn.Dropout2d(0.25)):
- Applies 2D dropout with a 0.25 probability, randomly zeroing out features during training to prevent overfitting.

Flatten (nn.Flatten):
- Converts the 4D tensor [batch_size, 256, 4, 4] into a 2D tensor [batch_size, 256 * 4 * 4] (4096 features) for the fully connected layer.

Linear Layer (nn.Linear(256 * 4 * 4, 1)):
- A fully connected layer that reduces 4096 features to 1 output per sample, representing the real/fake score.

Sigmoid (nn.Sigmoid):
- Applies the sigmoid activation to convert the output to a probability between 0 (fake) and 1 (real).

After this, **Label Embedding** is Done. It defines embedding layers to convert discrete label values into spatial feature maps that can be concatenated with the image input. The work in this portion is outlined:

Parameters:
- **nn.Embedding(num_classes_X, img_size * img_size):** Creates an embedding layer where:
  - num_classes_X (e.g., num_classes_ATM) is the number of unique categories for each label (e.g., 10 possible ATM values).
  - img_size * img_size (e.g., 256 * 256 = 65536) is the embedding dimension, matching the image size to create a 2D feature map.

Function: Maps each label index (e.g., an integer for ATM_value) to a vector of size img_size * img_size, which is later reshaped into a 2D grid (e.g., [1, 256, 256]) to align with the image dimensions.

Assumption: Variables num_classes_ATM, num_classes_CR, num_classes_FR, num_classes_PS, and img_size are defined elsewhere (e.g., img_size = 256), representing the number of classes and image resolution.

After this, the **Forward Pass** is implemented. Defines how the Discriminator processes input data and produces an output during the forward pass. The details are:

Inputs:

- **img:** Tensor of shape [batch_size, 1, height, width] (e.g., [64, 1, 256, 256]) representing the input image.
- **labels_ATM, labels_CR, labels_FR, labels_PS:** Tensors of shape [batch_size] containing integer indices for each label.

Label Embedding and Reshaping:

- **self.label_embedding_X(labels_X)**: Converts each label index into an embedding vector of size img_size * img_size.
- **.view(labels_X.size(0), 1, img.size(2), img.size(3)):** Reshapes the embedding into a 4D tensor:
    - labels_X.size(0): Batch size.
    - 1: Single channel (since each label is embedded into a 2D map).
    - img.size(2), img.size(3): Height and width of the image (e.g., 256, 256), ensuring spatial alignment.
- This creates four 2D feature maps (one per label), each matching the image's spatial dimensions.

Concatenation:

- **torch.cat((img, label_ATM_flat, ...), dim=1):** Concatenates the image (1 channel) and the four label embeddings (4 channels) along the channel dimension (dim=1), resulting in a tensor of shape [batch_size, 5, height, width] (e.g., [64, 5, 256, 256]).
- This combines the image and conditional information into a single input for the model.

Model Processing:

- **validity = self.model(d_in)**: Passes the concatenated input through the self.model sequential network, producing a tensor of shape [batch_size, 1] with probabilities.

Output:

- **return validity:** Returns the Sigmoid output, where each value represents the probability that the corresponding input is real (0 to 1).

Even though the Generator and Discriminator portion was designed separately by me and Humaun Kabir, we sat down to discuss our code snippets from time to time and optimized it's features so that these two blocks might work synchronously.
The dependencies of each other in this work:

- The output of the generator is used as the input for the Discriminator.
- Hyperparameters of the training should be aligned in both of the code blocks
- Data Normalization should be aligned.
- While the Generator consistently up samples it's inputs to go to a higherresolution output (picture), the Discriminator would have to down sample to extract the features from the input pictures and predict REAL/FAKE.

How did we implement this in our code:

- The Discriminator's forward method depends on the Generator's forward method to provide fake_images during training (e.g., in the Discriminator's optimization block).
- The Generator's optimization depends on the Discriminator's forward method to evaluate its generated images (e.g., in the Generator's optimization block)
- There is upsampling and down sampling in the two blocks. So they are listed as below:

  - **Upsampling in Generator**:
    - Initial size: 32x32 (from self.init_size = img_size // 8 with img_size = 256).
    - First upsampling: 32x32 → 64x64 (via nn.Upsample(scale_factor=2) in self.conv_blocks).
    - Second upsampling: 64x64 → 128x128 (via second nn.Upsample(scale_factor=2)).
    - Third upsampling: 128x128 → 256x256 (via third nn.Upsample(scale_factor=2)).
    - Final output: 256x256 (after nn.Conv2d(32, 1, ...) and nn.Tanh).
- **Downsampling in Discriminator**:
    - Initial size: 256x256 (input img shape).
    - First downsampling: 256x256 → 128x128 (via nn.Conv2d(1 + 4, 16, ..., stride=2)).
    - Second downsampling: 128x128 → 64x64 (via nn.Conv2d(16, 32, ..., stride=2)).
    - Third downsampling: 64x64 → 32x32 (via nn.Conv2d(32, 64, ..., stride=2)).

- Fourth downsampling: 32x32 → 16x16 (via nn.Conv2d(64, 128, ..., stride=2)).
- Fifth downsampling: 16x16 → 8x8 (via nn.Conv2d(128, 256, ..., stride=2)).
- Implied downsampling: 8x8 → 4x4

Second Part of our work consisted of designing the remaining part of the code which :

- Defines the optimizer (Adam)
- Loss function define (BCE) and plotting for generator and discriminator
- Label generation

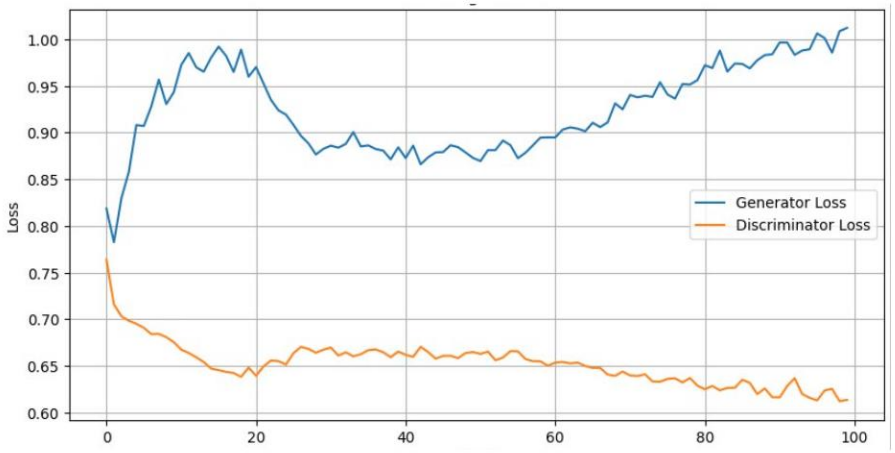

Fig: Loss vs Epoch Graph

From here, we see that our model has the optimum region at around 42-43 epochs.

Third part of my work consisted of creating parts of code that would evaluate the results. We divided into three parts:

1. Humaun Kabir: Data handling.
2. Soumya Swaraj Mondal: Image Processing logic and
3. Shihab Shakur: Results visualization and graphical representation.

On my part for writing the Image processing logic and visualiization part I did these portions:

- **Core Processing (process_image)**:
  - Developed **process_image** for Zip/Batches
  - loading images with Image.open().convert('L')
  - cropping via image.crop,
  - applying cv2.threshold(..., cv2.THRESH_OTSU)
  - detecting contours with cv2.findContours, and computing metrics (e.g., line width, RMS roughness, overspray %) with pixel_to_micron conversion
  - returning a metrics dictionary with error handling.

- **Single Image Script**:
  - Builds end-to-end script with files.upload()
  - Reusing process_image
  - Adding aerosol jet diameter (via re.search for FR or user input)
  - Generating 9 Matplotlib subplots (e.g., plt.imshow, cv2.Canny, edge plots) for visualization.

Results:

- Metric Extraction
- Create images that show visual cues for metrics (e.g., overlay contours in green/red, edge points in cyan/magenta) for clear idea on the presence of the extracted metrices.

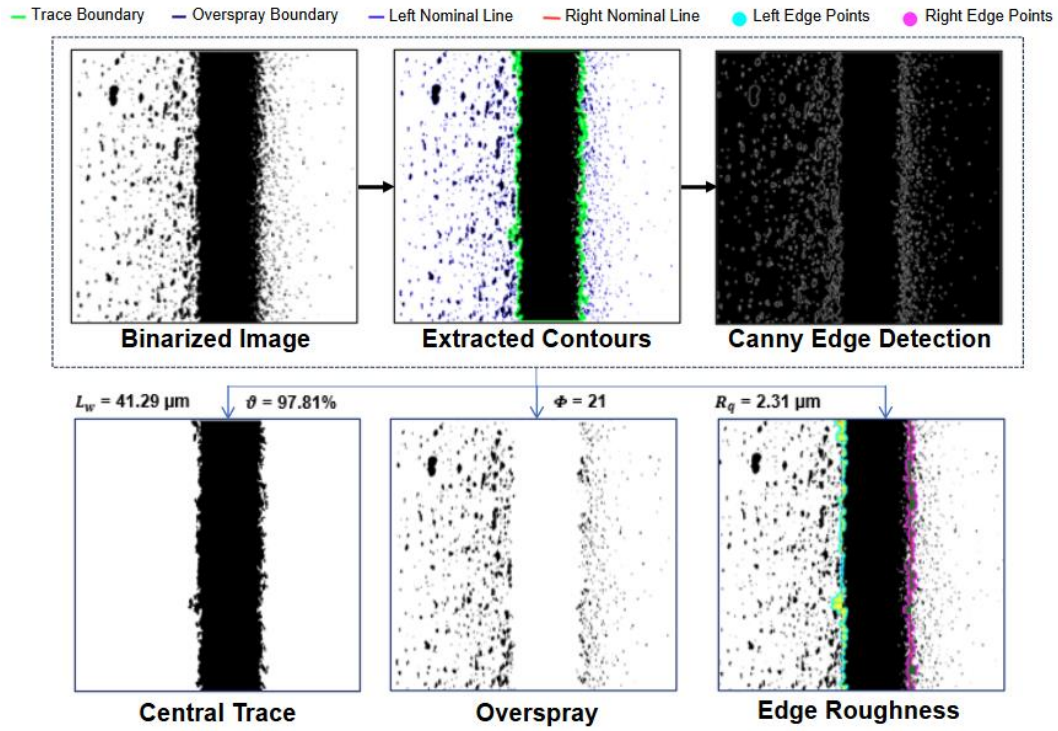

Fig Workflow for quantifying AJP-specific morphology attributes using OpenCV. Example trace printed with ATM = 30 V, CGFR = 6 sccm, FR = 13, and SS = 0.4 mm/s. Measured values: L_w = 41.29 µm, ϑ = 97.81%, R_q = 2.31 µm, and Φ of 21%.

Summary:

For this Project, I had a chance to work on making my understanding better on Aerosol Jet Printing (AJP) and how Deep Learning could help us to build a data driven approach to add reliability to this work system.

I have also worked on creating a Discriminator that is one of the core part of our CGAN. Through this I have got an idea how to make a feedback-based system that generates a probability based output (Real/Fake).

At the last portion, I have designed the image processing part that is used for evaluation metric. Here I have designed the code such that it can extract features from a picture and gives us the idea what are the deposit morphologies we are looking at.