

Inverse Problems in Photonics

Machine Learning Accelerated Solutions of Inverse Problems in Photonics

Gustav Fredrikson
Johan Rensfeldt
Fredrik Gillgren

Francesco Ferranti & Prashant Singh
Uppsala University

January 8, 2024

Table of Contents

- 1 Introduction
- 2 Previous Research
- 3 Theory & Method
- 4 Results & Discussion
- 5 Conclusions

Introduction

Introduction

Research Focus

- Addressing inverse problems in photonics with advanced machine learning techniques.
- Exploring Mixture Density Network (MDN), Invertible Neural Network (INN), and Dirichlet Process Gaussian Mixture Model (DPGMM).
- Tackling the one-to-many mapping problem in nanophotonic structures.

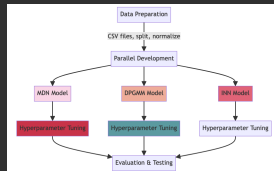


Figure: Schematic representation of our methodological approach, showing the stages from data preparation to model evaluation and testing.

Introduction - Dataset Overview

Data Composition

- The dataset is sourced from CSV files.
- Input features X and target responses Y .
- Dimensions of X : 3847×5 (samples \times features).
- Dimensions of Y : 3847×101 (samples \times features).

Data Processing and Normalization

- Dataset split into training, validation, and testing subsets.
- Normalization: Both X and Y values are normalized between 0 and 1.

Previous Research

Our contribution



Theory & Method

Theory - MDN - Overview

- **Mixture density network (MDN):** A Mixture Density Network is a type of neural network that outputs the parameters of a mixture model, usually Gaussian, to model complex and multimodal data distributions.

Mathematical Formulation

$$p(\mathbf{t} \mid \mathbf{x}) = \sum_{i=1}^m \alpha_i(\mathbf{x}) \phi_i(\mathbf{t} \mid \mathbf{x}) \quad (1)$$

$$\phi_i(\mathbf{t} \mid \mathbf{x}) = \frac{1}{(2\pi)^{\frac{c}{2}} \sigma_i(\mathbf{x})^c} \exp \left\{ -\frac{\|\mathbf{t} - \mu_i(\mathbf{x})\|^2}{2\sigma_i(\mathbf{x})^2} \right\} \quad (2)$$

$$\sum_{i=1}^m \phi_i(\mathbf{x}) = 1, \alpha_i = \frac{\exp(z_i^\alpha)}{\sum_{i=1}^M \exp(z_i^\alpha)}, \sigma_i = \exp(z_i^\sigma), \mu_{ik} = z_{ik}^\mu \quad (3)$$

Theory - MDN - Model structure

- 1 Neural network model learns parameters: means, variances, and mixture coefficients for each component.
- 2 Mixture model uses the parameters output by the neural network to define a set of Gaussian distributions.
- 3 Making predictions?

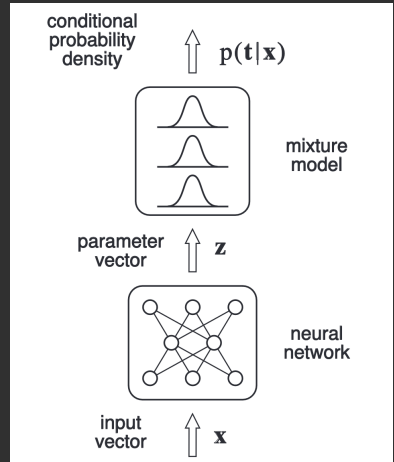
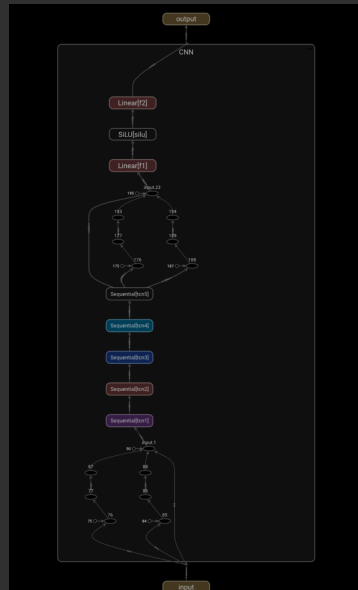


Figure: Model overview

Theory - MDN - Making predictions

- How do you choose what the most probable value of x given t ?
 - 1 The most likely value of x given input vector t is the maximum of the conditional density $p(t | x)$.
 - 2 A good approximation is $\max(\frac{\alpha_i(x^c)}{\sigma_i(x^c)})$.
 - 3 Or training another neural network in choosing the best approximate value.



Method - MDN - Overview

- 1 Create and train several different MDNs (MDN1-6).
- 2 Optimize models
- 3 Evaluate Model accuracy on all of the models individually.
- 4 Combine the models.

```

1 class MDN(nn.Module):
2     def __init__(self, in_features=10, out_features=5, num_gaussians=9):
3         super(MDN, self).__init__()
4
5         self.in_features, self.out_features, self.num_gaussians = in_features, out_features, num_gaussians
6
7         # Different architecture for shared layers
8         self.shared_layers2 = nn.Sequential(
9             nn.Conv1d(in_features, 256, kernel_size=3, stride=1, padding=1),
10            nn.BatchNorm1d(256),
11            nn.Dropout(0.2),
12            nn.ReLU(),
13            nn.Conv1d(256, 256, kernel_size=3, stride=1, padding=1),
14            nn.BatchNorm1d(256),
15            nn.Dropout(0.2),
16            nn.ReLU(),
17        )
18
19        self.shared_layers1 = nn.Sequential(
20            nn.Linear(256, 256),
21            nn.BatchNorm1d(256),
22            nn.ReLU(),
23            nn.Dropout(0.2),
24            nn.Linear(256, 128),
25            nn.BatchNorm1d(128),
26            nn.ReLU(),
27            nn.Dropout(0.2),
28            nn.Linear(128, 128),
29            nn.BatchNorm1d(128),
30            nn.ReLU(),
31            nn.Dropout(0.2),
32            nn.Linear(128, 64),
33            nn.ReLU()
34        )
35
36        # Different activation function for pi and a different structure for sigma and mu
37        self.pi = nn.Sequential(
38            nn.Linear(64, num_gaussians),
39            nn.Softmax(dim=1)
40        )
41        self.sigma = nn.Linear(64, out_features * num_gaussians)
42        self.mu = nn.Linear(64, out_features * num_gaussians)
43
44        def forward(self, y):
45            y = y.reshape(y.shape[0], y.shape[1], 1)
46            y = self.shared_layers2(y)
47            y = y.reshape(y.shape[0], y.shape[1])
48            y = self.shared_layers1(y)
49            pi = self.pi(y)
50            sigma = torch.exp(self.sigma(y))
51            sigma = sigma.view(-1, self.num_gaussians, self.out_features)
52            mu = self.mu(y)
53            mu = mu.view(-1, self.num_gaussians, self.out_features)
54            return pi, sigma, mu
55
56        def init_weights(self, m):

```

Method - MDN - Training models

- Using ADAMS optimizer.
- Negative log likelihood loss function.
- Using gradient clipping.
- Using StepLR scheduler.
- 500 epochs

```

1 def train_mdn(t_loader, v_loader, n_epochs, early_stop, model):
2     if isinstance(model, MDN4) or isinstance(model, MDN6):
3         optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4, eps=1e-8, betas=(0.9, 0.999))
4         scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)
5     else:
6         optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
7     early_stop_counter = 0
8     best_model_wts = copy.deepcopy(model.state_dict())
9     best_loss, best_train = 1000.0, 0
10    history = dict(train=[], val=[])
11    model.apply(model.init_weights)
12    for epoch in range(1, n_epochs + 1):
13        # Training
14        model = model.train()
15        train_losses = []
16        for batch_idx, (y, x) in enumerate(t_loader):
17            optimizer.zero_grad()
18            pi_variable, sigma_variable, mu_variable = model(y)
19            loss = mdn_loss_fn(pi_variable, sigma_variable, mu_variable, x)
20            loss.backward()
21            optimizer.step()
22            train_losses.append(loss.item())
23        if isinstance(model, MDN4) or isinstance(model, MDN6):
24            old_lr = scheduler.get_last_lr()[0] # Get the last learning rate
25            scheduler.step()
26            new_lr = scheduler.get_lr()[0] # Get the current learning rate
27            if new_lr != old_lr:
28                print(f"Epoch {epoch}: Learning rate changed from {old_lr} to {new_lr}")
29        if isinstance(model, MDN4) or isinstance(model, MDN6):
30            # gradient clipping triggered when gradient explodes
31            for p in model.parameters():
32                if p.grad is not None:
33                    if torch.any(torch.isnan(p.grad)):
34                        print(f"Epoch {epoch}: Gradient exploded")
35                        nn.utils.clip_grad_norm_(model.parameters(), 1.0)
36

```

Figure: Model overview

Method - MDN - Optimizing models

- Focusing on dropout rate and the number of gaussians.
- Baesian gridsearch.
- Ax and BoTorch.
- 40 trails and 300 epochs.

```

1 def grid_search_bayesian_parallel(models, num_trials=30, num_epochs=250, early_stop=5, batch_size=5):
2     all_trial_mappings = {}
3     best_model_params = {}
4     for model in models:
5         model_name = model.__name__
6         print(f'Now training {model_name}')
7         ax_client = AxClient()
8         ax_client.create_experiment(
9             name=f"mdn_experiment_{model_name}",
10            parameters=[
11                ("num_gaussians", "type": "range", "bounds": [6, 14], "value_type": "int"),
12                ("dropout", "type": "range", "bounds": [0.1, 0.8], "value_type": "float"),
13            ],
14            objective_name="validation_loss",
15            minimize=True
16        )
17        trial_model_mapping = {}
18        for batch in tqdm(range(0, num_trials, batch_size), desc=f"Optimizing {model_name}"):
19            trials_data = []
20            for _ in range(batch_size):
21                parameters, trial_index = ax_client.get_next_trial()
22                trials_data.append((parameters, trial_index))
23                unique_trial_index = f"{model_name}_{batch + 1}_{trial_index}"
24                trial_model_mapping[unique_trial_index] = model
25                # Training models in parallel for the current batch
26                validation_losses = train_model_in_parallel(model, [data[0] for data in trials_data],
27                                                            mdn_train_loader, mdn_val_loader, num_epochs,
28                                                            early_stop)
29                # Completing each trial in the batch
30                for (validation_loss, (_, trial_index)) in zip(validation_losses, trials_data):
31                    ax_client.complete_trial(trial_index=trial_index, raw_data=(validation_loss, 0.0))
32            best_parameters, values = ax_client.get_best_parameters()
33            best_model_params[model_name] = best_parameters
34            all_trial_mappings[model_name] = trial_model_mapping
35
36    return best_model_params, all_trial_mappings

```

Figure: Model overview

Method - MDN - Models

■ Summary of the models

Model	Num. Gaussians	Special Features
MDN1	12	Linear layers with SiLU, Tanh
MDN2	8	Conv1d, BatchNorm1d, ReLU
MDN3	12	Conv1d, ELU, Linear
MDN4	13	LSTM layers, ReLU, Linear
MDN5	10	Linear layers with LeakyReLU, PReLU
MDN6	12	Conv1d, MaxPool1d, Relu

Table: Overview of MDN Models

Method - MDN - Combining models

- Average mu, pi and sigmas or average model outputs.

```

1 # Load models
2 models = [torch.load(f'MDN{i}.pth') for i in range(1, 7)]
3 # Load the forward model
4 mod = torch.load('models/forward_models/cnn.pth')
5 # Store final predictions from each model
6 all_final_predictions = []
7 # Generate and process predictions for each model
8 for model in models:
9     dS, dI81 = [], []
10    for batch_idx, (y, x) in enumerate(mdn_test_loader):
11        for i in y:
12            pi_variable, sigma_variable, mu_variable = model[i].unsqueeze(0)
13            j = list(pi_variable.detach()).numpy()[0].index(max(list(pi_variable.detach()).numpy()[0]))
14            dS.append(list(mu_variable.detach()).numpy()[0][j])
15            pre = mod(torch.from_numpy(mu_variable.detach()).numpy()[0][j]).unsqueeze(0).squeeze().tolist()
16            dI81.append(pre)
17        all_final_predictions.append(dI81)
18    # Convert to numpy array for averaging
19    all_final_predictions_np = np.array(all_final_predictions)
20    # Average the final predictions
21    dI81 = np.mean(all_final_predictions_np, axis=0)

```

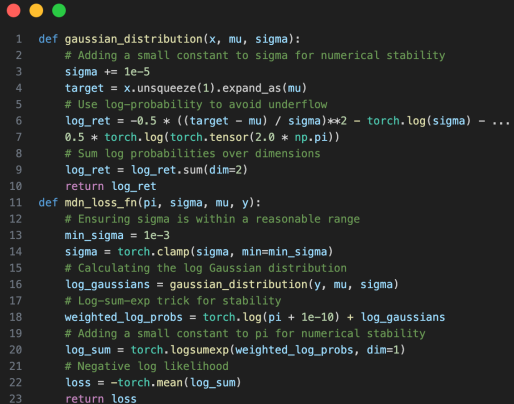
```

1 def ensemble_predict(models, input_data):
2     batch_pi, batch_sigma, batch_mu = [], [], []
3     for model in models:
4         pi, sigma, mu = model(input_data)
5         batch_pi.append(pi)
6         batch_sigma.append(sigma)
7         batch_mu.append(mu)
8     # Average across models
9     avg_pi = torch.mean(torch.stack(batch_pi), dim=0)
10    avg_sigma = torch.mean(torch.stack(batch_sigma), dim=0)
11    avg_mu = torch.mean(torch.stack(batch_mu), dim=0)
12    return avg_pi, avg_sigma, avg_mu
13 p, s, mu, o = [], [], [], []
14 for batch_idx, (y, x) in enumerate(mdn_test_loader):
15     for i in y:
16         pi_variable, sigma_variable, mu_variable = ensemble_predict(models, i.unsqueeze(0))
17         p.append(pi_variable.detach().numpy()[0])
18         s.append(sigma_variable.detach().numpy()[0])
19         m.append(mu_variable.detach().numpy()[0])
20 dS, dI81 = [], []
21 mod = torch.load('models/forward_models/cnn.pth')
22 for i in range(len(a)):
23     j = list(p[i]).index(max(list(p[i])))
24     dS.append(list(m[i][j]))
25     pre = mod(torch.from_numpy(m[i][j])).unsqueeze(0).squeeze().tolist()
26     dI81.append(pre)

```


Method - MDN - Numerical stability

- 1 Using the Log-Sum-Exp Trick.
- 2 Adding a small constant to sigma and pi.
- 3 Avoiding numerical underflow by computing the log of the Gaussian probability rather than the probability itself.
- 4 Clamping sigma to a smallest value of 1e-3



```

1  def gaussian_distribution(x, mu, sigma):
2      # Adding a small constant to sigma for numerical stability
3      sigma += 1e-5
4      target = x.unsqueeze(1).expand_as(mu)
5      # Use log-probability to avoid underflow
6      log_ret = -0.5 * ((target - mu) / sigma)**2 - torch.log(sigma) - ...
7      0.5 * torch.log(torch.tensor(2.0 * np.pi))
8      # Sum log probabilities over dimensions
9      log_ret = log_ret.sum(dim=2)
10     return log_ret
11
12  def mdn_loss_fn(pi, sigma, mu, y):
13      # Ensuring sigma is within a reasonable range
14      min_sigma = 1e-3
15      sigma = torch.clamp(sigma, min=min_sigma)
16      # Calculating the log Gaussian distribution
17      log_gaussians = gaussian_distribution(y, mu, sigma)
18      # Log-sum-exp trick for stability
19      weighted_log_probs = torch.log(pi + 1e-10) + log_gaussians
20      # Adding a small constant to pi for numerical stability
21      log_sum = torch.logsumexp(weighted_log_probs, dim=1)
22      # Negative log likelihood
23      loss = -torch.mean(log_sum)
24      return loss

```

Figure: Model overview

DPGMM - Theory - Dirichlet Process

- **Dirichlet Process (DP):** A Bayesian nonparametric approach for modeling infinite-dimensional probability spaces.
- **Key Elements:** Base Distribution G_0 and Concentration Parameter α .

Mathematical Formulation

DP uses a stick-breaking process for its discrete nature:

$$G = \sum_{k=1}^{\infty} \beta_k \delta_{\theta_k}, \quad \beta_k = \nu_k \prod_{l=1}^{k-1} (1 - \nu_l), \quad \nu_k \sim \text{Beta}(1, \alpha). \quad (4)$$

DPGMM - Theory - GMM & DPGMM Combination

- **Gaussian Mixture Model (GMM):** A probabilistic model using a combination of Gaussian distributions.
- **Components:** Each Gaussian is characterized by a mean vector (μ_k) and covariance matrix (Σ_k).
- **DPGMM:** Merges the Dirichlet Process and Gaussian Mixture Model, leveraging nonparametric priors for the mixing proportions.

Mathematical Representation of DPGMM

DPGMM combines infinite Gaussian components with DP-derived mixing coefficients:

$$p(x) = \sum_{k=1}^{\infty} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k), \quad (5)$$

where π_k are mixing coefficients from DP, and $\mathcal{N}(x|\mu_k, \Sigma_k)$ are Gaussian components.

DPGMM - Method - Class Structure and Functionality

■ Initialization:

- Initializes model with parameters: number of components, covariance type, weight concentration prior type, etc.
- Parameters define model behavior and capabilities.

■ Predict Method:

- Central to VBGMR class.
- Takes dataset and indices of input/output variables.
- Employs trained Gaussian mixture model for output estimation.

■ Model Training:

- 'fit' method trains VBGMR on a dataset.
- Optimizes mixture model parameters using variational Bayesian approach.

DPGMM- Method - Training and Evaluation Process

- Detailed process to train and evaluate the DPGMM.

Step	Description
1. Data Standardization	Scale invariance for input/output data.
2. Model Fitting	Training VBGMR on the dataset.
3. Analysis	Conducting forward and inverse analyses.
4. Performance Metrics	Evaluating with MSE and different losses.

Table: Training and Evaluation Steps for DPGMM

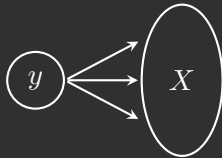
DPGMM - Method - Implementation Challenges

Implementation Challenges

- Balancing model complexity with performance for diverse data patterns.
- Achieving computational efficiency using Python and standard libraries.
- Potential enhancement with neural network-based training.

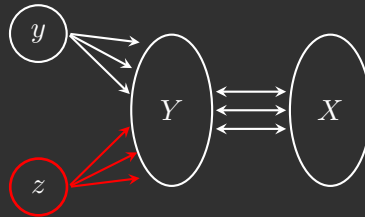
Theory - INN - Introduction

Original inverse problem



One-to-many mapping

Augmented inverse problem



Latent variable

Bijective mapping

The original problem is often ill-posed due to one-to-many mapping. An augmented inverse problem is formulated based on bijective mapping by introducing an additional latent random variable z .

Theory - INN - Transformations

Reversible Transformations

$$y = f(x) = x \odot \exp(s(x)) + t(x) \quad (6)$$

$$x = f^{-1}(y) = (y - t(x)) \odot \exp(-s(x)) \quad (7)$$

Critical Design Considerations

- The design of $s(x)$ and $t(x)$ critical for inverse computation.
- Channel-wise splitting enables independence in outputs relative to inputs.

Note: $s(x)$ and $t(x)$ are scaling and transformation functions modeled by neural networks.

Theory - INN - Channel-Wise Splitting

Channel-wise Splitting

Input divided into two parts. For instance:

$$\begin{aligned}v_1 &= u_1 \exp(s_1(u_2)) + t_1(u_2) \\v_2 &= u_2 \exp(s_2(v_1)) + t_1(v_1)\end{aligned}\tag{8}$$

Forward and Inverse Computation

These functions are evaluated in the forward direction, even when the block is inverted, as follows:

$$\begin{aligned}u_2 &= v_2 - t_2(v_1) \exp(-s_2(v_1)) \\u_1 &= v_1 - t_1(u_2) \exp(-s_1(u_2))\end{aligned}\tag{9}$$

Method - INN - Training

```
1  for x, y in train_loader:
2      optimizer.zero_grad()
3
4      # Forward
5      output, _ = model(x_padded)
6      y_pred, z = output[:, :ndim_y], output[:, ndim_y:]
7      loss = loss_fit(y_pred, y)
8      loss += loss_latent(z)
9
10     # Backward
11     x_pred, _ = model(y, rev=True)
12     loss += loss_rev_fit(x_pred, x)
13
14     loss.backward()
15     optimizer.step()
```

Key Metrics

- Forward MSE
- Latent MMD
- Backward MSE

Figure: Training Code Snippet

Method - INN - Challenges

Implementation Challenges

1. **Initial Implementation Approach:** Transitioning to FrEIA framework for scalability.
2. **Double Backpropagation in Training Loop:** Revised to a single, more efficient backpropagation step.
3. **Balancing Loss Components:** Calibrating composite loss function for effective learning.
4. **Hyperparameter Optimization:** Efficient model configuration within computational constraints.
5. **Extended Training Time and Feedback Loop:** Managing prolonged training times for iterative adjustments.

Results & Discussion

Results - MDN - Model outputs

Model output

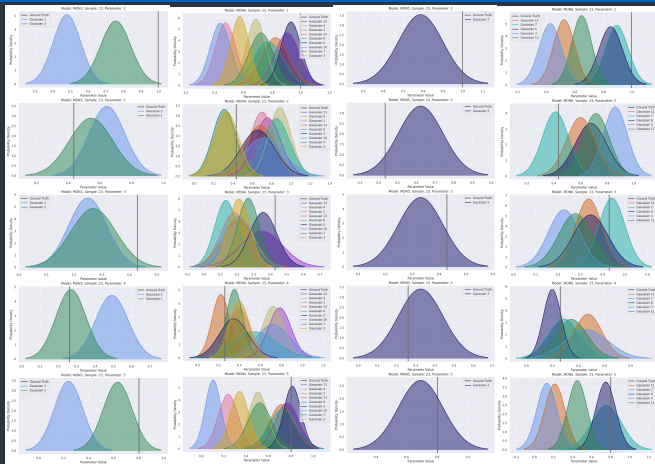


Figure: Model Figure: Model Figure: Model Figure: Model

3

4

5

6

Results - MDN - Performance metrics

- Evaluating performance in forward pass for all models as well as the average model.

Model	RMSE	Log MAE
Model 1	0.04744	-1.47919
Model 2	0.18287	-0.32174
Model 3	0.03565	-1.63800
Model 4	0.07076	-1.72049
Model 5	0.14245	-0.46148
Model 6	0.04430	-2.03734
Overall Average RMSE		0.07653
Overall Average Log MAE		-1.47330

Table: Performance Metrics of Models

Results - MDN - Visualizing predictions

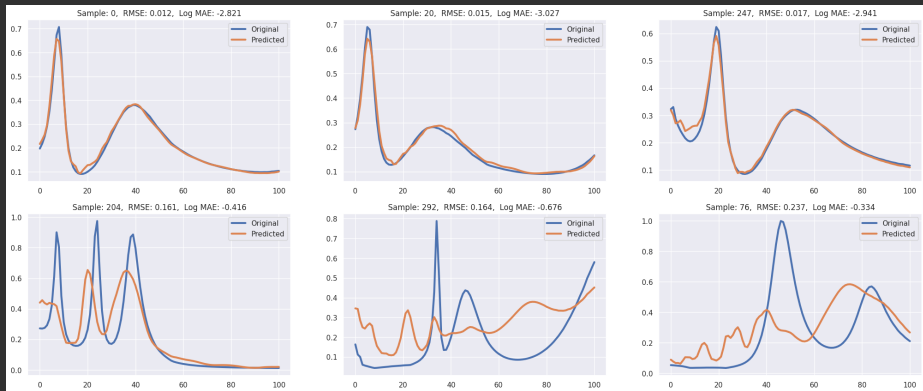


Figure: Caption

Result - DPGMM - Performance Metrics

- Evaluating the DPGMM's performance in forward and inverse analyses.

Metric	Metric Type	Value
MSE	Forward Analysis	0.013258
MSE	Inverse Analysis	0.058056
Training Loss	-	0.163675
Validation Loss	-	0.480573

Table: Performance Metrics for the DPGMM

Result - DPGMM - PDF Plot Visualizations

- Visualization of the density distribution of DPGMM's predictions.
- True Value, Mean, and Median Predictions are indicated by colored lines.

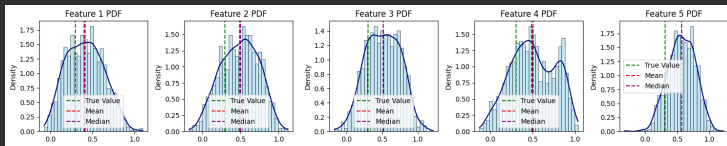


Figure: Probability Density Function (PDF) Plot of DPGMM Predictions

Results - INN - Quantitative

Metric Type	Metric	Value
Prediction	Forward MSE	3.29×10^{-4}
	Backward MSE	8.41×10^{-3}
Training	Train Loss	0.0462
	Validation Loss	0.0793
Reverse	Reverse Train Loss	0.0018
	Reverse Val Loss	0.0809

Table: Performance Metrics for Invertible Neural Network (INN). Loss on INN refers to the weighted composition loss.

Results - INN - Qualitative

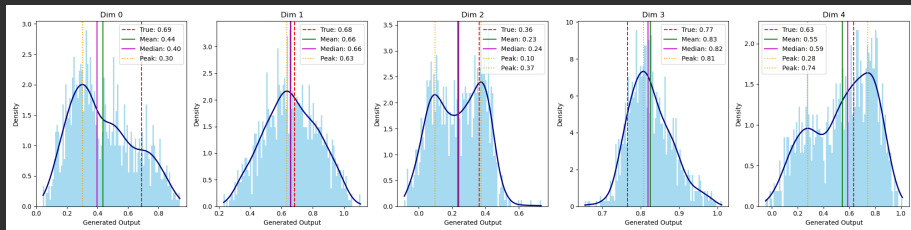


Figure: Density distribution of generated outputs for sample 0 across five dimensions. Showcasing multimodality in Dim 2 and 4.

Discussion - INN

High Accuracy in Forward Predictions

Demonstrates INN's effectiveness in input-output mapping. MSE of 3.29×10^{-4} indicates precise learning.

Reverse Mapping Challenges

Backward MSE of 8.41×10^{-3} suggests areas for improvement in reverse predictions.

Implications for Photonics

INN's proficiency in handling complex data, shown by multimodal distributions, makes it suitable for diverse applications in photonics and related fields.

Discussion - INN

High Accuracy in Forward Predictions

Demonstrates INN's effectiveness in input-output mapping. MSE of 3.29×10^{-4} indicates precise learning.

Reverse Mapping Challenges

Backward MSE of 8.41×10^{-3} suggests areas for improvement in reverse predictions.

Implications for Photonics

INN's proficiency in handling complex data, shown by multimodal distributions, makes it suitable for diverse applications in photonics and related fields.

Discussion - INN

High Accuracy in Forward Predictions

Demonstrates INN's effectiveness in input-output mapping. MSE of 3.29×10^{-4} indicates precise learning.

Reverse Mapping Challenges

Backward MSE of 8.41×10^{-3} suggests areas for improvement in reverse predictions.

Implications for Photonics

INN's proficiency in handling complex data, shown by multimodal distributions, makes it suitable for diverse applications in photonics and related fields.

Future Work - INN

Future Directions

1. Active Learning Integration
2. Physics-Informed Neural Networks
3. Transfer Learning Applications

Conclusions

Conclusions

Exploration of ML Models in Photonics

The study effectively explores the efficacy of MDN, INN, and DPGMM models in complex inverse problems, showing significant potential to enhance the photonics field.

Machine Learning in Photonic Design Optimization

The findings validate the role of machine learning in photonic design optimization, especially in improving accuracy and simplifying inverse problem-solving in critical applications like sensing and imaging.

Conclusions

Exploration of ML Models in Photonics

The study effectively explores the efficacy of MDN, INN, and DPGMM models in complex inverse problems, showing significant potential to enhance the photonics field.

Machine Learning in Photonic Design Optimization

The findings validate the role of machine learning in photonic design optimization, especially in improving accuracy and simplifying inverse problem-solving in critical applications like sensing and imaging.

The End