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

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.

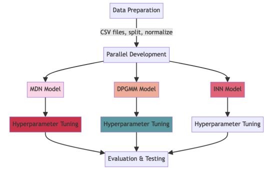


Figure: Schematic representation of our methodological approach, showing the stages from data preparation to model evaluation and testing.

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

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

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

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

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?

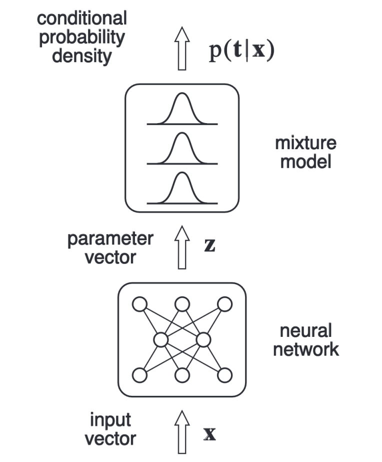
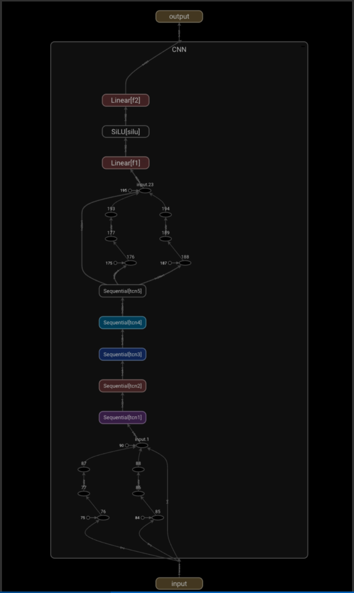


Figure: Model overview

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 $p(\mathbf{t} \mid \mathbf{x})$.
2. A good approximation is $\max(\frac{\alpha_i(\mathbf{x}^c)}{\sigma i(\mathbf{x}^c)})$.
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.

```python
class MDN3(nn.Module):
    def __init__(self, in_features=10, out_features=5, num_gaussians=9):
        super(MDN3, self).__init__()

        self.in_features, self.out_features, self.num_gaussians = in_features, out_features, num_gaussians

        # Different architecture for shared layers
        self.shared_layers2 = nn.Sequential(
            nn.Conv1d(in_features, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(256),
            nn.Dropout(0.2),
            nn.ELU(),
            nn.Conv1d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(256),
            nn.Dropout(0.2),
            nn.ELU(),
        )

        self.shared_layers1 = nn.Sequential(
            nn.Linear(256, 256),
            nn.BatchNorm1d(256),
            nn.ELU(),
            nn.Dropout(0.2),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ELU(),
            nn.Dropout(0.2),
            nn.Linear(128, 128),
            nn.BatchNorm1d(128),
            nn.ELU(),
            nn.Dropout(0.2),
            nn.Linear(128, 64),
            nn.ELU()
        )

        # Different activation function for pi and a different structure for sigma and mu
        self.pi = nn.Sequential(
            nn.Linear(64, num_gaussians),
            nn.Softmax(dim=1)
        )
        self.sigma = nn.Linear(64, out_features * num_gaussians)
        self.mu = nn.Linear(64, out_features * num_gaussians)

    def forward(self, y):
        y = y.reshape(y.shape[0], y.shape[1], 1)
        y = self.shared_layers2(y)
        y = y.reshape(y.shape[0], y.shape[1])
        y = self.shared_layers1(y)
        pi = self.pi(y)
        sigma = torch.exp(self.sigma(y))
        sigma = sigma.view(-1, self.num_gaussians, self.out_features)
        mu = self.mu(y)
        mu = mu.view(-1, self.num_gaussians, self.out_features)
        return pi, sigma, mu

    def init_weights(self, m):
```

Method - MDN - Traning models

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

```python
def train_mdn(t_loader, v_loader, n_epochs, early_stop, model):
    if isinstance(model, MDN4) or isinstance(model, MDN6):
        optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4, eps=1e-8, betas=(0.9, 0.999))
        scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)
    else:
        optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    early_stop_counter = 0
    best_model_wts = copy.deepcopy(model.state_dict())
    best_loss, best_train = 1000.0, 0
    history = dict(train=[], val=[])
    model.apply(model.init_weights)
    for epoch in range(1, n_epochs + 1):
        # Training
        model = model.train()
        train_losses = []
        for batch_idx, (y, x) in enumerate(t_loader):
            optimizer.zero_grad()
            pi_variable, sigma_variable, mu_variable = model(y)
            loss = mdn_loss_fn(pi_variable, sigma_variable, mu_variable, x)
            loss.backward()
            optimizer.step()
            train_losses.append(loss.item())
        if isinstance(model, MDN4) or isinstance(model, MDN6):
            old_lr = scheduler.get_last_lr()[0]  # Get the last learning rate
            scheduler.step()
            new_lr = scheduler.get_lr()[0]  # Get the current learning rate
            if new_lr != old_lr:
                print(f'Epoch {epoch}: Learning rate changed from {old_lr} to {new_lr}')

        if isinstance(model, MDN3) or isinstance(model, MDN6):
            # gradient clipping triggered when gradient explodes
            for p in model.parameters():
                if p.grad is not None:
                    if torch.any(torch.isnan(p.grad)):
                        print(f'Epoch {epoch}: Gradient exploded')
                        nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

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.

```python
def grid_search_bayesian_parallel(models, num_trials=30, num_epochs=250, early_stop=5, batch_size=5):
    all_trial_mappings = {}
    best_model_params = {}
    for model in models:
        model_name = model.__name__
        print(f'Now training {model_name}')
        ax_client = AxClient()
        ax_client.create_experiment(
            name=f"mdn_experiment_{model_name}",
            parameters=[
                {"name": "num_gaussians", "type": "range", "bounds": [6, 14], "value_type": "int"},
                {"name": "dropout", "type": "range", "bounds": [0.1, 0.8], "value_type": "float"},
            ],
            objective_name="validation_loss",
            minimize=True
        )
        trial_model_mapping = {}
        for batch in tqdm(range(0, num_trials, batch_size), desc=f"Optimizing {model_name}"):
            trials_data = []
            for _ in range(batch_size):
                parameters, trial_index = ax_client.get_next_trial()
                trials_data.append((parameters, trial_index))
                unique_trial_index = f"{model_name}_{batch + _}"
                trial_model_mapping[unique_trial_index] = model_name
            # Training models in parallel for the current batch
            validation_losses = train_model_in_parallel(model, [data[0] for data in trials_data],
                                                        mdn_train_loader, mdn_val_loader, num_epochs,
                                                        early_stop)
            # Completing each trial in the batch
            for (validation_loss, (_, trial_index)) in zip(validation_losses, trials_data):
                ax_client.complete_trial(trial_index=trial_index, raw_data=(validation_loss, 0.0))
        best_parameters, values = ax_client.get_best_parameters()
        best_model_params[model_name] = best_parameters
        all_trial_mappings[model_name] = trial_model_mapping

    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

```python
# Load models
    models = [torch.load(f'MDN{i}.pth') for i in range(1, 7)]
    # Load the forward model
    mod = torch.load('models/forward_models/cnn.pth')
    # Store final predictions from each model
    all_final_predictions = []
    # Generate and process predictions for each model
    for model in models:
        d5, d101 = [], []
        for batch_idx, (y, x) in enumerate(mdn_test_loader):
            for i in y:
                pi_variable, sigma_variable, mu_variable = model(i.unsqueeze(0))
                j = list(pi_variable.detach().numpy()[0]).index(max(list(pi_variable.detach().numpy()[0])))
                d5.append(list(mu_variable.detach().numpy()[0][j]))
                pre = mod(torch.from_numpy(mu_variable.detach().numpy()[0][j]).unsqueeze(0)).squeeze().tolist()
                d101.append(pre)
        all_final_predictions.append(d101)
    # Convert to numpy array for averaging
    all_final_predictions_np = np.array(all_final_predictions)
    # Average the final predictions
    d101 = np.mean(all_final_predictions_np, axis=0)
```

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

```python
def ensemble_predict(models, input_data):
    batch_pi, batch_sigma, batch_mu = [], [], []
    for model in models:
        pi, sigma, mu = model(input_data)
        batch_pi.append(pi)
        batch_sigma.append(sigma)
        batch_mu.append(mu)
    # Average across models
    avg_pi = torch.mean(torch.stack(batch_pi), dim=0)
    avg_sigma = torch.mean(torch.stack(batch_sigma), dim=0)
    avg_mu = torch.mean(torch.stack(batch_mu), dim=0)
    return avg_pi, avg_sigma, avg_mu
p, s, m, o = [], [], [], []
for batch_idx, (y, x) in enumerate(mdn_test_loader):
    for i in y:
        pi_variable, sigma_variable, mu_variable = ensemble_predict(models, i.unsqueeze(0))
        p.append(pi_variable.detach().numpy()[0])
        s.append(sigma_variable.detach().numpy()[0])
        m.append(mu_variable.detach().numpy()[0])
d5, d101 = [], []
mod = torch.load('models/forward_models/cnn.pth')
for i in range(len(m)):
    j = list(p[i]).index(max(list(p[i])))
    d5.append(list(m[i][j]))
    pre = mod(torch.from_numpy(m[i][j]).unsqueeze(0)).squeeze().tolist()
    d101.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

```python
def gaussian_distribution(x, mu, sigma):
    # Adding a small constant to sigma for numerical stability
    sigma += 1e-5
    target = x.unsqueeze(1).expand_as(mu)
    # Use log-probability to avoid underflow
    log_ret = -0.5 * ((target - mu) / sigma)**2 - torch.log(sigma) - ...
    0.5 * torch.log(torch.tensor(2.0 * np.pi))
    # Sum log probabilities over dimensions
    log_ret = log_ret.sum(dim=2)
    return log_ret
def mdn_loss_fn(pi, sigma, mu, y):
    # Ensuring sigma is within a reasonable range
    min_sigma = 1e-3
    sigma = torch.clamp(sigma, min=min_sigma)
    # Calculating the log Gaussian distribution
    log_gaussians = gaussian_distribution(y, mu, sigma)
    # Log-sum-exp trick for stability
    weighted_log_probs = torch.log(pi + 1e-10) + log_gaussians
    # Adding a small constant to pi for numerical stability
    log_sum = torch.logsumexp(weighted_log_probs, dim=1)
    # Negative log likelihood
    loss = -torch.mean(log_sum)
    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:

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

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:

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

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.

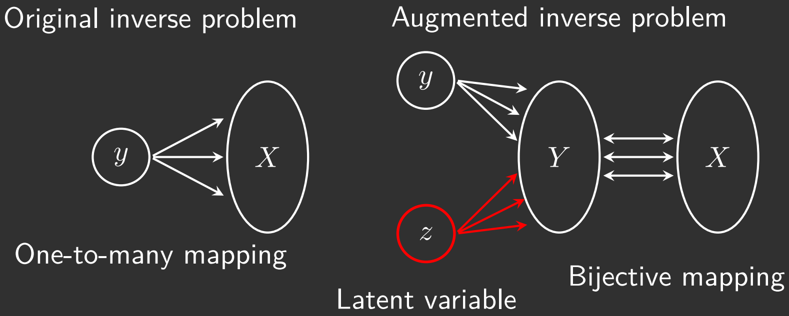
Table: Training and Evaluation Steps for DPGMM

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



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

$$y = f(x) = x \odot \exp(s(x)) + t(x) \tag{6}$$
$$x = f^{-1}(y) = (y - t(x)) \odot \exp(-s(x)) \tag{7}$$

The design of $s(x)$ and $t(x)$ is critical. They must be constructed in a way that their outputs do not depend on all components of x, thus allowing for the computation of the inverse. Achieved through channel-wise splitting:

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

s_j and t_j scaling and transformation functions modeled by neural networks. Evaluated in the forward direction, even if the block is inverted.

Method - INN - Training

```python
for x, y in train_loader:
    optimizer.zero_grad()

    # Forward
    output, _ = model(x_padded)
    y_pred, z = output[:, :ndim_y], output[:, ndim_y:]
    loss = loss_fit(y_pred, y)
    loss += loss_latent(z)

    # Backward
    x_pred, _ = model(y, rev=True)
    loss += loss_rev_fit(x_pred, x)

    loss.backward()
    optimizer.step()
```

1. Forward MSE
2. Latent MMD
3. Backward MSE

Method - INN - Challenges

Results & Discussion

Results - MDN - Model outputs

Model output

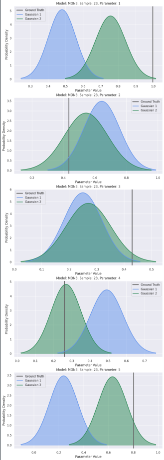


Figure: Model 3

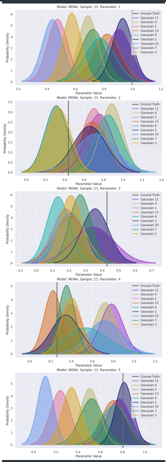


Figure: Model 4

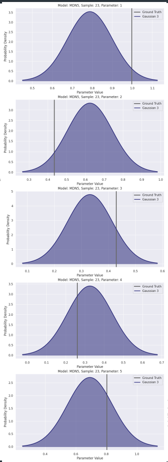


Figure: Model 5

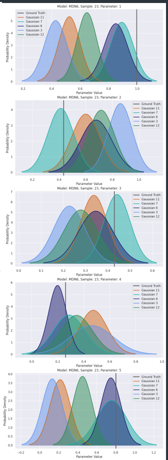


Figure: Model 6

Results - MDN - Performence 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

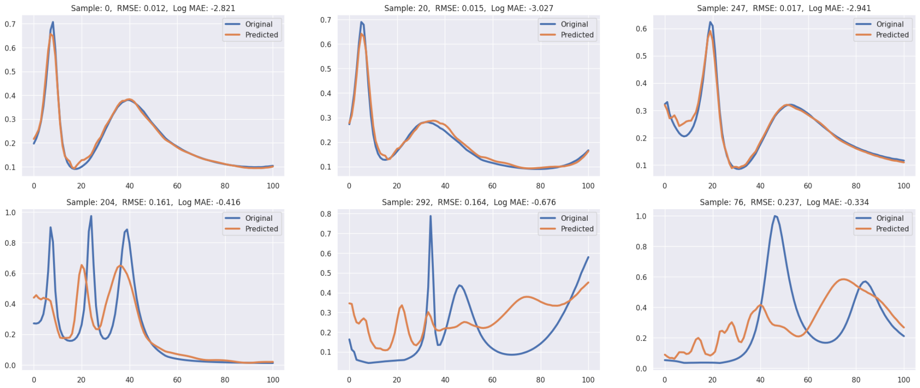


Figure: Caption

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.

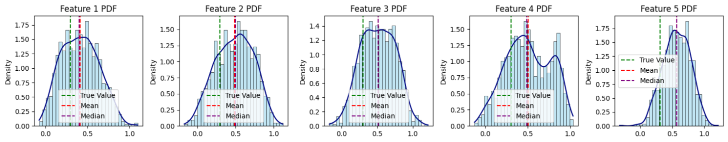


Figure: Probability Density Function (PDF) Plot of DPGMM Predictions

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

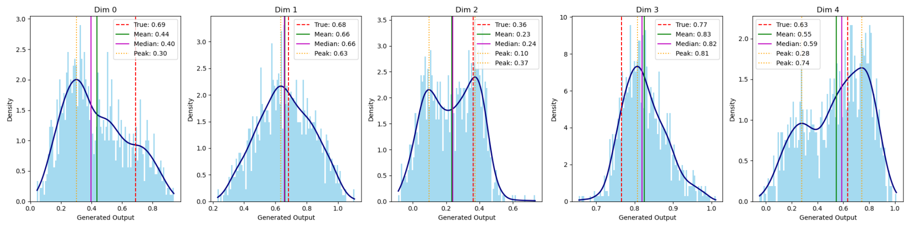


Figure: Density distribution of generated outputs for sample 0 across five dimensions.

Discussion - INN

Future Work - INN

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

Conclusions

Conclusions

The End