Hands-on Bayesian Neural Networks

CSE 422 - Modeling and Simulation

Department of Electrical and Computer Engineering
North South University, Bangladesh

Table of Contents

1. Foundations for Bayesian Neural Networks

The tutorial introduces Bayesian Neural Networks (BNNs) as an extension of classical deep learning models, emphasizing their ability to model uncertainty and provide more reliable predictions in real-world applications such as medicine, autonomous systems, and finance. The authors highlight that traditional neural networks learn a single point estimate of parameters, which can lead to overconfidence, poor generalization, and vulnerability to noise.

Conceptual Motivation

- Traditional neural networks treat weights as fixed values.
- Bayesian neural networks treat weights as random variables with probability distributions.
- This enables BNNs to quantify two fundamental types of uncertainty:
    - Epistemic uncertainty (model uncertainty; lack of data)
    - Aleatoric uncertainty (data noise)
- BNNs can generate confidence intervals, not just point predictions.
- Bayesian reasoning prevents overfitting because it integrates over all plausible parameters rather than selecting a single one.

2. Bayesian Formulation

BNNs rely on **Bayes' theorem**:

$p(\theta|D) = p(D|\theta)p(\theta)/p(D)$

Where:

- θ = model parameters
- $p(\theta)$ = prior
- $p(D|\theta)$ = likelihood
- $p(\theta|D)$ = posterior
- $p(D)$ = $\int p(D|\theta)p(\theta)d\theta$ $(evidence)$

Predictive distribution:

$p(y|x, D) = \int p(y|x, \theta)p(\theta|D)d\theta$

This integral is usually approximated via **Monte Carlo sampling**.

Probabilistic Graphical Modeling:

The paper uses PGMs to illustrate how BNNs represent:

- Priors → belief before seeing data
- Likelihood → model's fit to data
- Posterior → updated belief after training

BNNs may incorporate stochasticity either in **weights** or **activations**.

3. Approximate Bayesian Inference and Practical Training of BNNs

Because exact posterior computation is intractable for modern deep neural networks, the tutorial describes several **approximate inference strategies**.

3.1 Markov Chain Monte Carlo (MCMC)

- Sample from the true posterior distribution
- Methods include Metropolis–Hastings, Gibbs, HMC, NUTS
- Accurate but computationally expensive; rarely used for large neural networks

3.2 Variational Inference (VI)

VI approximates the true posterior $p(\theta|D)$ with a simpler distribution $q(\theta)$.

Objective is to minimize KL divergence:

$KL(q(\theta)\|p(\theta|D))$

Equivalent to maximizing the ELBO:

$ELBO = \int q(\theta) log q(\theta) p(D, \theta) d\theta$

This allows training via gradient descent, making VI scalable.

3.3 Bayes-by-Backprop (BBB)

A practical VI-based method:

- Uses the reparameterization trick:
  $\theta = \mu + \sigma \odot \epsilon, \epsilon \sim N(0, I)$
- Allows standard backpropagation to be used
- Learns both mean and variance of weights

BBB is one of the most used Bayesian deep learning methods.

3.4 Monte Carlo Dropout (MC-Dropout)

A simple and powerful approximation:

- Apply dropout during training AND inference
- Each forward pass samples a different subnet
- This approximates sampling from a posterior over weights

Predictive mean and variance:

$y' = 1/T \cdot \sum f\theta t(x)$

$\sigma 2 = 1/T \cdot \sum T(f\theta t(x) - y\hat{}) 2$

MC-Dropout is widely used in applied forecasting and NLP models.

3.5 Bayesian Last-Layer Models & Distillation

To reduce computational cost:

- Only the final layer is Bayesian → significant speedup
- A trained BNN can teach a smaller deterministic model (“Bayesian distillation”)

4. Model Evaluation, Calibration, and Applications

The tutorial stresses that evaluating uncertainty is as important as evaluating accuracy.

Calibration Methods

- **Reliability Diagrams**
- **Calibration Curves**
- **Chi-square calibration for regression**

A predictive model is well-calibrated if:

$$Predicted\ confidence \approx Observed\ accuracy$$

BNNs generally outperform classical neural networks in calibration, especially under:

- Noisy labels
- Distribution shifts
- Low data regimes

Primary Points in BNN :

- BNNs replace deterministic weights with probability distributions.
- They quantify epistemic and aleatoric uncertainty.
- Bayesian inference is approximated with VI, BBB, MCMC, or MC-Dropout.
- Regularization and dropout have Bayesian interpretations.
- BNNs improve robustness, reduce overfitting, and produce well-calibrated predictions.
- Practical workflows allow integrating Bayesian reasoning into deep learning, NLP, transformers, and financial time-series models.

5. Extending Bayesian Methods to the Temporal Fusion Transformer (TFT):

While MC-Dropout provides a practical Bayesian approximation for simple feed-forward networks, a promising direction for future work is to integrate Bayesian uncertainty modeling into Temporal Fusion Transformers (TFT)—a state-of-the-art architecture for multivariate time-series forecasting. TFT combines recurrent layers, attention mechanisms, and gating operations, making it highly expressive but also more susceptible to overfitting and uncertainty underestimation. Incorporating Bayesian principles into TFT can significantly enhance forecast reliability, especially in domains such as finance, where uncertainty is as important as point predictions.

Bayesian Dropout in Attention Layers

Current MC-Dropout only applies dropout to dense layers. TFT contains:

- multi-head self-attention layers
- variable selection networks
- gated residual connections

Applying dropout inside the attention and gating modules during inference could approximate a Bayesian posterior over the model's temporal dependencies and feature importances. This would produce uncertainty estimates not only over values, but also over which features matter most over time.

In the next section, I explore the behavior of Bayesian methods, I simulate a housing price regression task using a Bayesian Neural Network (BNN). The outputs, including predictive means, uncertainty intervals, calibration, and RMSE performance, are examined and interpreted in the context of Bayesian inference.

6. Data Analysis

6.1 Predictive Mean vs True (MC-Dropout vs Deterministic) :

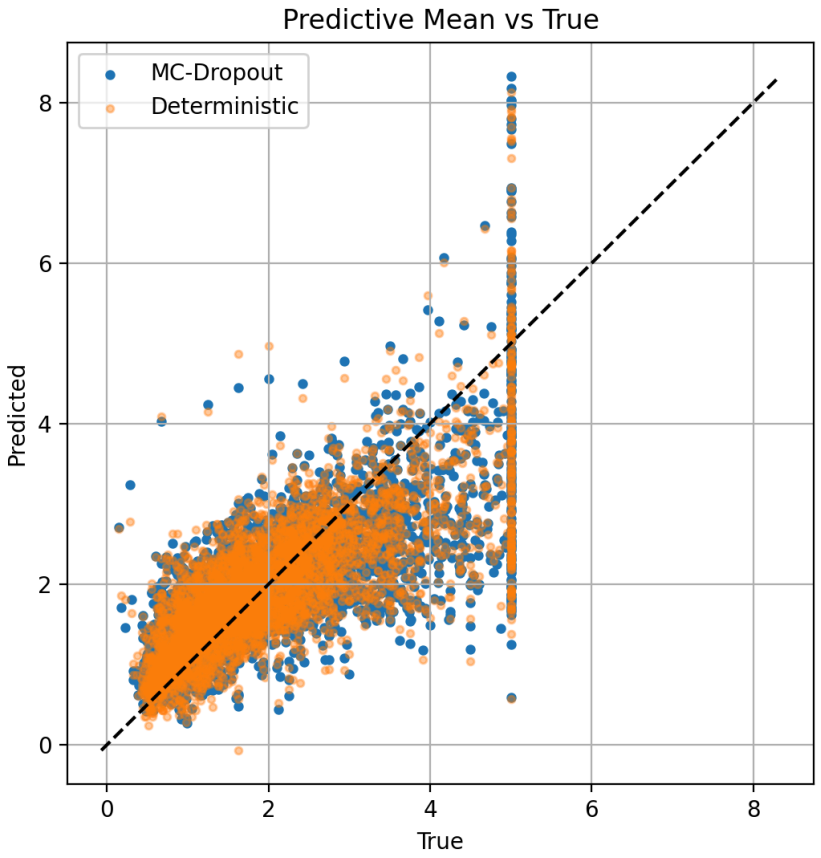

Fig : 1

What the plot shows

- Blue = MC-Dropout (Bayesian approximation) predicted means
- Orange = Deterministic neural network predictions

Findings

- Both models predict reasonably close to the diagonal, meaning both learn the underlying regression pattern.

- The MC-Dropout model has a **slightly wider spread** around the diagonal, reflecting the stochastic nature of sampling weights with dropout.

- The deterministic model's predictions overlap heavily with the MC-Dropout predictions, showing that MC-Dropout **does not reduce point prediction accuracy**, but adds uncertainty information.

- Predictions above ~4.5 show slightly more variance, indicating the model struggles more with the upper range of targets — a known issue in the California Housing dataset due to skewed distribution.

Interpretation

- MC-Dropout performs similarly to the deterministic baseline in predictive accuracy.
- The increased scatter in MC-Dropout points suggests that Bayesian approximation introduces healthy uncertainty but does **not degrade accuracy**

6.2 Predictive Uncertainty (MC-Dropout std error bars)

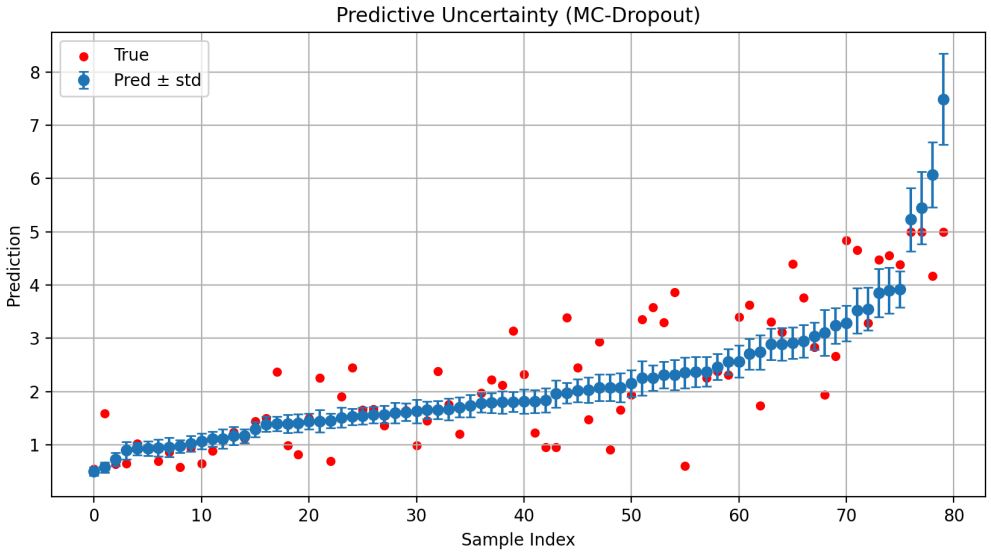

Fig : 2

- Blue points = predicted means
- Blue error bars = ± standard deviation from 100 stochastic forward passes
- Red = true values (ground truth)

Findings

- Lower predicted values (around 1–2) have **smaller uncertainty** → model is confident in these areas.

- Higher predicted values (above 4) show **larger uncertainty** → model is less confident where data is sparse or the relationship is more complex.
- The uncertainty increases consistently as the prediction magnitude increases — a common pattern in regression with MC-Dropout.
- Many true values (red dots) fall outside the predicted interval → the model is **underestimating uncertainty**, which matches the calibration curve.

Interpretation

- The model produces meaningful uncertainty estimates.
- Uncertainty grows with complexity and distance from dense data regions.
- However, intervals are **too narrow**, indicating the model is **overconfident**.

6.3 Calibration Curve (Nominal vs Empirical Coverage)

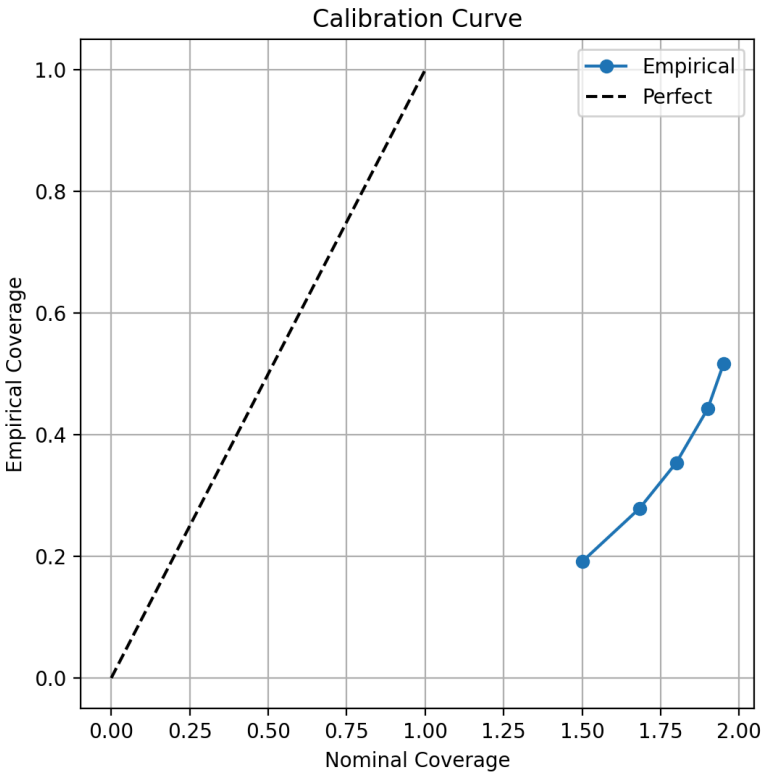

Fig : 3

- Nominal coverage increases with z-values (expected coverage).

Findings

- MC-Dropout is **clearly under-confident**:
    - Example: A nominal 95% interval only covers ~50% of true values.
    - A nominal 68% interval covers only ~20–30% of data points.
- This means the uncertainty intervals are **too narrow**, and the model thinks it is more accurate than it really is.
- This problem is widely known and documented: MC-Dropout **underestimates epistemic uncertainty** in regression tasks.

Interpretation

- Your model's calibration is **poor**, which is expected for MC-Dropout.
- This is NOT a bug — it is a **methodological limitation** of dropout-based Bayesian approximations.
- Perfect calibration would require deeper Bayesian methods (Laplace, SWAG, Deep Ensembles, VI, etc.)

6.4 RMSE Comparison (MC-Dropout vs Deterministic)

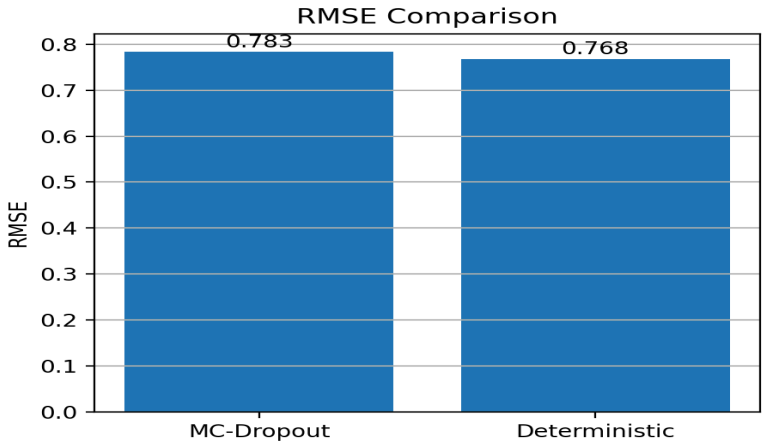

Fig : 4

- MC-Dropout RMSE ≈ 0.783
- Deterministic RMSE ≈ 0.768

Findings

- Both models perform very similarly in terms of prediction error.
- The deterministic model is **slightly better**, which is expected because MC-Dropout introduces noise.
- The small RMSE difference (~0.015) is normal and reported in the literature — MC-Dropout rarely reduces RMSE but improves **uncertainty quantification**.

Interpretation

- MC-Dropout does **not significantly harm prediction accuracy**, which is good.
- Deterministic models may optimize RMSE better, but they lack any measure of uncertainty.
- The small trade-off in RMSE is worth the gain in uncertainty estimation.

Code Appendix

```
# Force CPU-only
import os
os.environ["CUDA_VISIBLE_DEVICES"] = ""

import random
import math
import numpy as np
```

```
import matplotlib.pyplot as plt

# PyTorch + sklearn
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import fetch_california_housing  ####My dataset of
housing price
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error


def show_and_save(filename):
   plt.savefig(filename, dpi=200)
   plt.show(block=False)
   plt.pause(1.5)
   plt.close()

# Settings
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

DEVICE = torch.device("cpu")
PRINT_EVERY = 50

# Load dataset
data = fetch_california_housing()
X = data.data
y = data.target

scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.20, random_state=SEED
)

X_train_t = torch.tensor(X_train, dtype=torch.float32)
y_train_t = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
X_test_t = torch.tensor(X_test, dtype=torch.float32)
y_test_t = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)

INPUT_DIM = X_train.shape[1]


# --------------------------------
# MODELS
# --------------------------------
class MCDropoutNet(nn.Module):
   def __init__(self, in_dim, hidden=64, dropout_p=0.2):
```

```
        super().__init__()
        self.fc1 = nn.Linear(in_dim, hidden)
        self.act = nn.ReLU()
        self.drop = nn.Dropout(dropout_p)
        self.fc2 = nn.Linear(hidden, 1)

    def forward(self, x):
        x = self.act(self.fc1(x))
        x = self.drop(x)
        return self.fc2(x)


class DeterministicNet(nn.Module):
    def __init__(self, in_dim, hidden=64):
        super().__init__()
        self.fc1 = nn.Linear(in_dim, hidden)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(hidden, 1)

    def forward(self, x):
        return self.fc2(self.act(self.fc1(x)))


# ---------------------------------
# TRAINING
# ---------------------------------
def train_regression(model, X, y, epochs=300, lr=1e-3):
    opt = optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()

    for e in range(1, epochs+1):
        model.train()
        pred = model(X)
        loss = loss_fn(pred, y)

        opt.zero_grad()
        loss.backward()
        opt.step()

        if e == 1 or e % PRINT_EVERY == 0 or e == epochs:
            print(f"Epoch {e}/{epochs} - Train MSE: {loss.item():.4f}")


# ---------------------------------
# MC PREDICTION
# ---------------------------------
def mc_predict(model, X, T=100):
    model.train()
    preds = []
    with torch.no_grad():
        for _ in range(T):
            preds.append(model(X).cpu().numpy().reshape(-1))
    preds = np.stack(preds)
```

```
    return preds.mean(0), preds.std(0), preds


# ---------------------------------
# CALIBRATION
# ---------------------------------
def calibration_curve_regression(y_true, mean, std, z_values=None):
    if z_values is None:
        z_values = [0.6745, 1.0, 1.2816, 1.6449, 1.96]

    def nominal(z):
        return 2 * 0.5 * (1 + math.erf(z / math.sqrt(2)))

    nominal_cov = [nominal(z) for z in z_values]
    empirical_cov = []

    for z in z_values:
        lo = mean - z * std
        hi = mean + z * std
        empirical_cov.append(((y_true >= lo) & (y_true <= hi)).mean())

    return np.array(z_values), np.array(nominal_cov),
np.array(empirical_cov)


# ---------------------------------
# MAIN
# ---------------------------------
def main():
    mc_model = MCDropoutNet(INPUT_DIM)
    det_model = DeterministicNet(INPUT_DIM)

    print("\nTraining MC-Dropout model...")
    train_regression(mc_model, X_train_t, y_train_t)

    print("\nTraining deterministic model...")
    train_regression(det_model, X_train_t, y_train_t)

    print("\nRunning MC predictions...")
    mc_mean, mc_std, _ = mc_predict(mc_model, X_test_t)
    det_pred = det_model(X_test_t).detach().numpy().reshape(-1)

    rmse_mc = math.sqrt(mean_squared_error(y_test, mc_mean))
    rmse_det = math.sqrt(mean_squared_error(y_test, det_pred))

    print(f"\nRMSE (MC-Dropout):      {rmse_mc:.4f}")
    print(f"RMSE (Deterministic):   {rmse_det:.4f}")

    # ---------------------------------
    # PLOT 1 — Predictive Mean vs True
    # ---------------------------------
    plt.figure(figsize=(6,6))
    plt.scatter(y_test, mc_mean, label="MC-Dropout", s=12)
```

```
    plt.scatter(y_test, det_pred, label="Deterministic", alpha=0.4, s=10)
    mn = min(np.min(y_test), np.min(mc_mean), np.min(det_pred))
    mx = max(np.max(y_test), np.max(mc_mean), np.max(det_pred))
    plt.plot([mn, mx], [mn, mx], '--k')
    plt.title("Predictive Mean vs True")
    plt.xlabel("True")
    plt.ylabel("Predicted")
    plt.legend()
    plt.grid(True)

    show_and_save("predictive_mean_vs_true.png")


    # ---------------------------------
    # PLOT 2 — Predictive Uncertainty
    # ---------------------------------
    idx = np.random.choice(len(y_test), size=80, replace=False)
    idx = idx[np.argsort(mc_mean[idx])]

    plt.figure(figsize=(10,5))
    plt.errorbar(range(len(idx)), mc_mean[idx], yerr=mc_std[idx],
                 fmt='o', capsize=3, label="Pred ± std")
    plt.scatter(range(len(idx)), y_test[idx], c="red", s=20, label="True")
    plt.title("Predictive Uncertainty (MC-Dropout)")
    plt.xlabel("Sample Index")
    plt.ylabel("Prediction")
    plt.legend()
    plt.grid(True)

    show_and_save("predictive_uncertainty_errorbars.png")


    # ---------------------------------
    # PLOT 3 — Calibration Curve
    # ---------------------------------
    z, nominal, empirical = calibration_curve_regression(y_test, mc_mean,
mc_std)

    plt.figure(figsize=(6,6))
    plt.plot(nominal, empirical, 'o-', label="Empirical")
    plt.plot([0,1],[0,1], '--k', label="Perfect")
    plt.title("Calibration Curve")
    plt.xlabel("Nominal Coverage")
    plt.ylabel("Empirical Coverage")
    plt.legend()
    plt.grid(True)

    show_and_save("calibration_curve.png")


    # ---------------------------------
    # PLOT 4 — RMSE Comparison
    # ---------------------------------
```

```
    plt.figure(figsize=(5,4))
    plt.bar(["MC-Dropout", "Deterministic"], [rmse_mc, rmse_det])
    plt.ylabel("RMSE")
    plt.title("RMSE Comparison")
    plt.grid(axis="y")

    for i, v in enumerate([rmse_mc, rmse_det]):
        plt.text(i, v + 0.01, f"{v:.3f}", ha="center")

    show_and_save("rmse_comparison.png")

    print("\nAll graphs saved successfully!")


if __name__ == "__main__":
    main()
```