Probabilistic Bayesian Neural Networks

Author: Khalid Salama
Date created: 2021/01/15
Last modified: 2021/01/15
Description: Building probabilistic Bayesian neural network models with TensorFlow Probability.

View in Colab • GitHub source

Introduction

Taking a probabilistic approach to deep learning allows to account for *uncertainty*, so that models can assign less levels of confidence to incorrect predictions. Sources of uncertainty can be found in the data, due to measurement error or noise in the labels, or the model, due to insufficient data availability for the model to learn effectively.

This example demonstrates how to build basic probabilistic Bayesian neural networks to account for these two types of uncertainty. We use TensorFlow Probability library, which is compatible with Keras API.

This example requires TensorFlow 2.3 or higher. You can install Tensorflow Probability using the following command:

```
pip install tensorflow-probability
```

The dataset

We use the Wine Quality dataset, which is available in the TensorFlow Datasets. We use the red wine subset, which contains 4,898 examples. The dataset has 11numerical physicochemical features of the wine, and the task is to predict the wine quality, which is a score between 0 and 10. In this example, we treat this as a regression task.

You can install TensorFlow Datasets using the following command:

```
pip install tensorflow-datasets
```

Setup

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_datasets as tfds
import tensorflow_probability as tfp
```

Create training and evaluation datasets

Here, we load the `wine_quality` dataset using `tfds.load()`, and we convert the target feature to float. Then, we shuffle the dataset and split it into training and test sets. We take the first `train_size` examples as the train split, and the rest as the test split.

```
def get_train_and_test_splits(train_size, batch_size=1):
    # We prefetch with a buffer the same size as the dataset because th dataset
    # is very small and fits into memory.
    dataset = (
        tfds.load(name="wine_quality", as_supervised=True, split="train")
        .map(lambda x, y: (x, tf.cast(y, tf.float32)))
        .prefetch(buffer_size=dataset_size)
        .cache()
    )
    # We shuffle with a buffer the same size as the dataset.
    train_dataset = (
        dataset.take(train_size).shuffle(buffer_size=train_size).batch(batch_size)
    )
    test_dataset = dataset.skip(train_size).batch(batch_size)

    return train_dataset, test_dataset
```

Compile, train, and evaluate the model

```
hidden_units = [8, 8]
learning_rate = 0.001


def run_experiment(model, loss, train_dataset, test_dataset):

    model.compile(
        optimizer=keras.optimizers.RMSprop(learning_rate=learning_rate),
        loss=loss,
        metrics=[keras.metrics.RootMeanSquaredError()],
    )

    print("Start training the model...")
    model.fit(train_dataset, epochs=num_epochs, validation_data=test_dataset)
    print("Model training finished.")
    _, rmse = model.evaluate(train_dataset, verbose=0)
    print(f"Train RMSE: {round(rmse, 3)}")

    print("Evaluating model performance...")
    _, rmse = model.evaluate(test_dataset, verbose=0)
    print(f"Test RMSE: {round(rmse, 3)}")
```

Create model inputs

```
FEATURE_NAMES = [
    "fixed acidity",
    "volatile acidity",
    "citric acid",
    "residual sugar",
    "chlorides",
    "free sulfur dioxide",
    "total sulfur dioxide",
    "density",
    "pH",
    "sulphates",
    "alcohol",
]


def create_model_inputs():
    inputs = {}
    for feature_name in FEATURE_NAMES:
        inputs[feature_name] = layers.Input(
            name=feature_name, shape=(1,), dtype=tf.float32
        )
    return inputs
```

Experiment 1: standard neural network

We create a standard deterministic neural network model as a baseline.

```
def create_baseline_model():
    inputs = create_model_inputs()
    input_values = [value for _, value in sorted(inputs.items())]
    features = keras.layers.concatenate(input_values)
    features = layers.BatchNormalization()(features)

    # Create hidden layers with deterministic weights using the Dense layer.
    for units in hidden_units:
        features = layers.Dense(units, activation="sigmoid")(features)
    # The output is deterministic: a single point estimate.
    outputs = layers.Dense(units=1)(features)

    model = keras.Model(inputs=inputs, outputs=outputs)
    return model
```

Let's split the wine dataset into training and test sets, with 85% and 15% of the examples, respectively.

```
dataset_size = 4898
batch_size = 256
train_size = int(dataset_size * 0.85)
train_dataset, test_dataset = get_train_and_test_splits(train_size, batch_size)
```

Now let's train the baseline model. We use the `MeanSquaredError` as the loss function.

```
num_epochs = 100
mse_loss = keras.losses.MeanSquaredError()
baseline_model = create_baseline_model()
run_experiment(baseline_model, mse_loss, train_dataset, test_dataset)
```

```
Start training the model...
Epoch 1/100
17/17 [==============================] - 1s 53ms/step - loss: 37.5710 -
root_mean_squared_error: 6.1294 - val_loss: 35.6750 - val_root_mean_squared_error: 5.9729
Epoch 2/100
17/17 [==============================] - 0s 7ms/step - loss: 35.5154 -
root_mean_squared_error: 5.9594 - val_loss: 34.2430 - val_root_mean_squared_error: 5.8518
Epoch 3/100
17/17 [==============================] - 0s 7ms/step - loss: 33.9975 -
root_mean_squared_error: 5.8307 - val_loss: 32.8003 - val_root_mean_squared_error: 5.7272
Epoch 4/100
17/17 [==============================] - 0s 12ms/step - loss: 32.5928 -
root_mean_squared_error: 5.7090 - val_loss: 31.3385 - val_root_mean_squared_error: 5.5981
Epoch 5/100
17/17 [==============================] - 0s 7ms/step - loss: 30.8914 -
root_mean_squared_error: 5.5580 - val_loss: 29.8659 - val_root_mean_squared_error: 5.4650

...

Epoch 95/100
17/17 [==============================] - 0s 6ms/step - loss: 0.6927 -
root_mean_squared_error: 0.8322 - val_loss: 0.6901 - val_root_mean_squared_error: 0.8307
Epoch 96/100
17/17 [==============================] - 0s 6ms/step - loss: 0.6929 -
root_mean_squared_error: 0.8323 - val_loss: 0.6866 - val_root_mean_squared_error: 0.8286
Epoch 97/100
17/17 [==============================] - 0s 6ms/step - loss: 0.6582 -
root_mean_squared_error: 0.8112 - val_loss: 0.6797 - val_root_mean_squared_error: 0.8244
Epoch 98/100
17/17 [==============================] - 0s 6ms/step - loss: 0.6733 -
root_mean_squared_error: 0.8205 - val_loss: 0.6740 - val_root_mean_squared_error: 0.8210
Epoch 99/100
17/17 [==============================] - 0s 7ms/step - loss: 0.6623 -
root_mean_squared_error: 0.8138 - val_loss: 0.6713 - val_root_mean_squared_error: 0.8193
Epoch 100/100
17/17 [==============================] - 0s 6ms/step - loss: 0.6522 -
root_mean_squared_error: 0.8075 - val_loss: 0.6666 - val_root_mean_squared_error: 0.8165
Model training finished.
Train RMSE: 0.809
Evaluating model performance...
Test RMSE: 0.816
```

We take a sample from the test set use the model to obtain predictions for them. Note that since the baseline model is deterministic, we get a single a *point estimate* prediction for each test example, with no information about the uncertainty of the model nor the prediction.

```
sample = 10
examples, targets = list(test_dataset.unbatch().shuffle(batch_size * 10).batch(sample))[
    0
]

predicted = baseline_model(examples).numpy()
for idx in range(sample):
    print(f"Predicted: {round(float(predicted[idx][0]), 1)} - Actual: {targets[idx]}")
```

```
Predicted: 6.0 - Actual: 6.0
Predicted: 6.2 - Actual: 6.0
Predicted: 5.8 - Actual: 7.0
Predicted: 6.0 - Actual: 5.0
Predicted: 5.7 - Actual: 5.0
Predicted: 6.2 - Actual: 7.0
Predicted: 5.6 - Actual: 5.0
Predicted: 6.2 - Actual: 6.0
Predicted: 6.2 - Actual: 6.0
Predicted: 6.2 - Actual: 7.0
```

Experiment 2: Bayesian neural network (BNN)

The object of the Bayesian approach for modeling neural networks is to capture the *epistemic uncertainty*, which is uncertainty about the model fitness, due to limited training data.

The idea is that, instead of learning specific weight (and bias) *values* in the neural network, the Bayesian approach learns weight *distributions* - from which we can sample to produce an output for a given input - to encode weight uncertainty.

Thus, we need to define prior and the posterior distributions of these weights, and the training process is to learn the parameters of these distributions.

```
# Define the prior weight distribution as Normal of mean=0 and stddev=1.
# Note that, in this example, the we prior distribution is not trainable,
# as we fix its parameters.
def prior(kernel_size, bias_size, dtype=None):
    n = kernel_size + bias_size
    prior_model = keras.Sequential(
        [
            tfp.layers.DistributionLambda(
                lambda t: tfp.distributions.MultivariateNormalDiag(
                    loc=tf.zeros(n), scale_diag=tf.ones(n)
                )
            )
        ]
    )
    return prior_model


# Define variational posterior weight distribution as multivariate Gaussian.
# Note that the learnable parameters for this distribution are the means,
# variances, and covariances.
def posterior(kernel_size, bias_size, dtype=None):
    n = kernel_size + bias_size
    posterior_model = keras.Sequential(
        [
            tfp.layers.VariableLayer(
                tfp.layers.MultivariateNormalTriL.params_size(n), dtype=dtype
            ),
            tfp.layers.MultivariateNormalTriL(n),
        ]
    )
    return posterior_model
```

We use the `tfp.layers.DenseVariational` layer instead of the standard `keras.layers.Dense` layer in the neural network model.

```python
def create_bnn_model(train_size):
    inputs = create_model_inputs()
    features = keras.layers.concatenate(list(inputs.values()))
    features = layers.BatchNormalization()(features)

    # Create hidden layers with weight uncertainty using the DenseVariational layer.
    for units in hidden_units:
        features = tfp.layers.DenseVariational(
            units=units,
            make_prior_fn=prior,
            make_posterior_fn=posterior,
            kl_weight=1 / train_size,
            activation="sigmoid",
        )(features)

    # The output is deterministic: a single point estimate.
    outputs = layers.Dense(units=1)(features)
    model = keras.Model(inputs=inputs, outputs=outputs)
    return model
```

The epistemic uncertainty can be reduced as we increase the size of the training data. That is, the more data the BNN model sees, the more it is certain about its estimates for the weights (distribution parameters). Let's test this behaviour by training the BNN model on a small subset of the training set, and then on the full training set, to compare the output variances.

Train BNN with a small training subset.

```python
num_epochs = 500
train_sample_size = int(train_size * 0.3)
small_train_dataset = train_dataset.unbatch().take(train_sample_size).batch(batch_size)

bnn_model_small = create_bnn_model(train_sample_size)
run_experiment(bnn_model_small, mse_loss, small_train_dataset, test_dataset)
```

```
Start training the model...
Epoch 1/500
5/5 [==============================] - 2s 123ms/step - loss: 34.5497 -
root_mean_squared_error: 5.8764 - val_loss: 37.1164 - val_root_mean_squared_error: 6.0910
Epoch 2/500
5/5 [==============================] - 0s 28ms/step - loss: 36.0738 -
root_mean_squared_error: 6.0007 - val_loss: 31.7373 - val_root_mean_squared_error: 5.6322
Epoch 3/500
5/5 [==============================] - 0s 29ms/step - loss: 33.3177 -
root_mean_squared_error: 5.7700 - val_loss: 36.2135 - val_root_mean_squared_error: 6.0164
Epoch 4/500
5/5 [==============================] - 0s 30ms/step - loss: 35.1247 -
root_mean_squared_error: 5.9232 - val_loss: 35.6158 - val_root_mean_squared_error: 5.9663
Epoch 5/500
5/5 [==============================] - 0s 23ms/step - loss: 34.7653 -
root_mean_squared_error: 5.8936 - val_loss: 34.3038 - val_root_mean_squared_error: 5.8556

...

Epoch 495/500
5/5 [==============================] - 0s 24ms/step - loss: 0.6978 - root_mean_squared_error:
0.8162 - val_loss: 0.6258 - val_root_mean_squared_error: 0.7723
Epoch 496/500
5/5 [==============================] - 0s 22ms/step - loss: 0.6448 - root_mean_squared_error:
0.7858 - val_loss: 0.6372 - val_root_mean_squared_error: 0.7808
Epoch 497/500
5/5 [==============================] - 0s 23ms/step - loss: 0.6871 - root_mean_squared_error:
0.8121 - val_loss: 0.6437 - val_root_mean_squared_error: 0.7825
Epoch 498/500
5/5 [==============================] - 0s 23ms/step - loss: 0.6213 - root_mean_squared_error:
0.7690 - val_loss: 0.6581 - val_root_mean_squared_error: 0.7922
Epoch 499/500
5/5 [==============================] - 0s 22ms/step - loss: 0.6604 - root_mean_squared_error:
0.7913 - val_loss: 0.6522 - val_root_mean_squared_error: 0.7908
Epoch 500/500
5/5 [==============================] - 0s 22ms/step - loss: 0.6190 - root_mean_squared_error:
0.7678 - val_loss: 0.6734 - val_root_mean_squared_error: 0.8037
Model training finished.
Train RMSE: 0.805
Evaluating model performance...
Test RMSE: 0.801
```

Since we have trained a BNN model, the model produces a different output each time we call it with the same input, since each time a new set of weights are sampled from the distributions to construct the network and produce an output. The less certain the mode weights are, the more variability (wider range) we will see in the outputs of the same inputs.

```
def compute_predictions(model, iterations=100):
    predicted = []
    for _ in range(iterations):
        predicted.append(model(examples).numpy())
    predicted = np.concatenate(predicted, axis=1)

    prediction_mean = np.mean(predicted, axis=1).tolist()
    prediction_min = np.min(predicted, axis=1).tolist()
    prediction_max = np.max(predicted, axis=1).tolist()
    prediction_range = (np.max(predicted, axis=1) - np.min(predicted, axis=1)).tolist()

    for idx in range(sample):
        print(
            f"Predictions mean: {round(prediction_mean[idx], 2)}, "
            f"min: {round(prediction_min[idx], 2)}, "
            f"max: {round(prediction_max[idx], 2)}, "
            f"range: {round(prediction_range[idx], 2)} - "
            f"Actual: {targets[idx]}"
        )


compute_predictions(bnn_model_small)
```

```
Predictions mean: 5.63, min: 4.92, max: 6.15, range: 1.23 - Actual: 6.0
Predictions mean: 6.35, min: 6.01, max: 6.54, range: 0.53 - Actual: 6.0
Predictions mean: 5.65, min: 4.84, max: 6.25, range: 1.41 - Actual: 7.0
Predictions mean: 5.74, min: 5.21, max: 6.25, range: 1.04 - Actual: 5.0
Predictions mean: 5.99, min: 5.26, max: 6.29, range: 1.03 - Actual: 5.0
Predictions mean: 6.26, min: 6.01, max: 6.47, range: 0.46 - Actual: 7.0
Predictions mean: 5.28, min: 4.73, max: 5.86, range: 1.12 - Actual: 5.0
Predictions mean: 6.34, min: 6.06, max: 6.53, range: 0.47 - Actual: 6.0
Predictions mean: 6.23, min: 5.91, max: 6.44, range: 0.53 - Actual: 6.0
Predictions mean: 6.33, min: 6.05, max: 6.54, range: 0.48 - Actual: 7.0
```

Train BNN with the whole training set.

```
num_epochs = 500
bnn_model_full = create_bnn_model(train_size)
run_experiment(bnn_model_full, mse_loss, train_dataset, test_dataset)

compute_predictions(bnn_model_full)
```

```
Start training the model...
Epoch 1/500
17/17 [==============================] - 2s 32ms/step - loss: 25.4811 -
root_mean_squared_error: 5.0465 - val_loss: 23.8428 - val_root_mean_squared_error: 4.8824
Epoch 2/500
17/17 [==============================] - 0s 7ms/step - loss: 23.0849 -
root_mean_squared_error: 4.8040 - val_loss: 24.1269 - val_root_mean_squared_error: 4.9115
Epoch 3/500
17/17 [==============================] - 0s 7ms/step - loss: 22.5191 -
root_mean_squared_error: 4.7449 - val_loss: 23.3312 - val_root_mean_squared_error: 4.8297
Epoch 4/500
17/17 [==============================] - 0s 7ms/step - loss: 22.9571 -
root_mean_squared_error: 4.7896 - val_loss: 24.4072 - val_root_mean_squared_error: 4.9399
Epoch 5/500
17/17 [==============================] - 0s 6ms/step - loss: 21.4049 -
root_mean_squared_error: 4.6245 - val_loss: 21.1895 - val_root_mean_squared_error: 4.6027

...

Epoch 495/500
17/17 [==============================] - 0s 7ms/step - loss: 0.5799 -
root_mean_squared_error: 0.7511 - val_loss: 0.5902 - val_root_mean_squared_error: 0.7572
Epoch 496/500
17/17 [==============================] - 0s 6ms/step - loss: 0.5926 -
root_mean_squared_error: 0.7603 - val_loss: 0.5961 - val_root_mean_squared_error: 0.7616
Epoch 497/500
17/17 [==============================] - 0s 7ms/step - loss: 0.5928 -
root_mean_squared_error: 0.7595 - val_loss: 0.5916 - val_root_mean_squared_error: 0.7595
Epoch 498/500
17/17 [==============================] - 0s 7ms/step - loss: 0.6115 -
root_mean_squared_error: 0.7715 - val_loss: 0.5869 - val_root_mean_squared_error: 0.7558
Epoch 499/500
17/17 [==============================] - 0s 7ms/step - loss: 0.6044 -
root_mean_squared_error: 0.7673 - val_loss: 0.6007 - val_root_mean_squared_error: 0.7645
Epoch 500/500
17/17 [==============================] - 0s 7ms/step - loss: 0.5853 -
root_mean_squared_error: 0.7550 - val_loss: 0.5999 - val_root_mean_squared_error: 0.7651
Model training finished.
Train RMSE: 0.762
Evaluating model performance...
Test RMSE: 0.759
Predictions mean: 5.41, min: 5.06, max: 5.9, range: 0.84 - Actual: 6.0
Predictions mean: 6.5, min: 6.16, max: 6.61, range: 0.44 - Actual: 6.0
Predictions mean: 5.59, min: 4.96, max: 6.0, range: 1.04 - Actual: 7.0
Predictions mean: 5.67, min: 5.25, max: 6.01, range: 0.76 - Actual: 5.0
Predictions mean: 6.02, min: 5.68, max: 6.39, range: 0.71 - Actual: 5.0
Predictions mean: 6.35, min: 6.11, max: 6.52, range: 0.41 - Actual: 7.0
Predictions mean: 5.21, min: 4.85, max: 5.68, range: 0.83 - Actual: 5.0
Predictions mean: 6.53, min: 6.35, max: 6.64, range: 0.28 - Actual: 6.0
Predictions mean: 6.3, min: 6.05, max: 6.47, range: 0.42 - Actual: 6.0
Predictions mean: 6.44, min: 6.19, max: 6.59, range: 0.4 - Actual: 7.0
```

Notice that the model trained with the full training dataset shows smaller range (uncertainty) in the prediction values for the same inputs, compared to the model trained with a subset of the training dataset.

Experiment 3: probabilistic Bayesian neural network

So far, the output of the standard and the Bayesian NN models that we built is deterministic, that is, produces a point estimate as a prediction for a given example. We can create a probabilistic NN by letting the model output a distribution. In this case, the model captures the *aleatoric uncertainty* as well, which is due to irreducible noise in the data, or to the stochastic nature of the process generating the data.

In this example, we model the output as a `IndependentNormal` distribution, with learnable mean and variance parameters. If the task was classification, we would have used `IndependentBernoulli` with binary classes, and `OneHotCategorical` with multiple classes, to model distribution of the model output.

```
def create_probablistic_bnn_model(train_size):
    inputs = create_model_inputs()
    features = keras.layers.concatenate(list(inputs.values()))
    features = layers.BatchNormalization()(features)

    # Create hidden layers with weight uncertainty using the DenseVariational layer.
    for units in hidden_units:
        features = tfp.layers.DenseVariational(
            units=units,
            make_prior_fn=prior,
            make_posterior_fn=posterior,
            kl_weight=1 / train_size,
            activation="sigmoid",
        )(features)

    # Create a probabilisticå output (Normal distribution), and use the `Dense` layer
    # to produce the parameters of the distribution.
    # We set units=2 to learn both the mean and the variance of the Normal distribution.
    distribution_params = layers.Dense(units=2)(features)
    outputs = tfp.layers.IndependentNormal(1)(distribution_params)

    model = keras.Model(inputs=inputs, outputs=outputs)
    return model
```

Since the output of the model is a distribution, rather than a point estimate, we use the negative loglikelihood as our loss function to compute how likely to see the true data (targets) from the estimated distribution produced by the model.

```
def negative_loglikelihood(targets, estimated_distribution):
    return -estimated_distribution.log_prob(targets)


num_epochs = 1000
prob_bnn_model = create_probablistic_bnn_model(train_size)
run_experiment(prob_bnn_model, negative_loglikelihood, train_dataset, test_dataset)
```

```
Start training the model...
Epoch 1/1000
17/17 [==============================] - 2s 36ms/step - loss: 11.2378 -
root_mean_squared_error: 6.6758 - val_loss: 8.5554 - val_root_mean_squared_error: 6.6240
Epoch 2/1000
17/17 [==============================] - 0s 7ms/step - loss: 11.8285 -
root_mean_squared_error: 6.5718 - val_loss: 8.2138 - val_root_mean_squared_error: 6.5256
Epoch 3/1000
17/17 [==============================] - 0s 7ms/step - loss: 8.8566 -
root_mean_squared_error: 6.5369 - val_loss: 5.8749 - val_root_mean_squared_error: 6.3394
Epoch 4/1000
17/17 [==============================] - 0s 7ms/step - loss: 7.8191 -
root_mean_squared_error: 6.3981 - val_loss: 7.6224 - val_root_mean_squared_error: 6.4473
Epoch 5/1000
17/17 [==============================] - 0s 7ms/step - loss: 6.2598 -
root_mean_squared_error: 6.4613 - val_loss: 5.9415 - val_root_mean_squared_error: 6.3466

...

Epoch 995/1000
17/17 [==============================] - 0s 7ms/step - loss: 1.1323 -
root_mean_squared_error: 1.0431 - val_loss: 1.1553 - val_root_mean_squared_error: 1.1060
Epoch 996/1000
17/17 [==============================] - 0s 7ms/step - loss: 1.1613 -
root_mean_squared_error: 1.0686 - val_loss: 1.1554 - val_root_mean_squared_error: 1.0370
Epoch 997/1000
17/17 [==============================] - 0s 7ms/step - loss: 1.1351 -
root_mean_squared_error: 1.0628 - val_loss: 1.1472 - val_root_mean_squared_error: 1.0813
Epoch 998/1000
17/17 [==============================] - 0s 7ms/step - loss: 1.1324 -
root_mean_squared_error: 1.0858 - val_loss: 1.1527 - val_root_mean_squared_error: 1.0578
Epoch 999/1000
17/17 [==============================] - 0s 7ms/step - loss: 1.1591 -
root_mean_squared_error: 1.0801 - val_loss: 1.1483 - val_root_mean_squared_error: 1.0442
Epoch 1000/1000
17/17 [==============================] - 0s 7ms/step - loss: 1.1402 -
root_mean_squared_error: 1.0554 - val_loss: 1.1495 - val_root_mean_squared_error: 1.0389
Model training finished.
Train RMSE: 1.068
Evaluating model performance...
Test RMSE: 1.068
```

Now let's produce an output from the model given the test examples. The output is now a distribution, and we can use its mean and variance to compute the confidence intervals (CI) of the prediction.

```python
prediction_distribution = prob_bnn_model(examples)
prediction_mean = prediction_distribution.mean().numpy().tolist()
prediction_stdv = prediction_distribution.stddev().numpy()

# The 95% CI is computed as mean ± (1.96 * stdv)
upper = (prediction_mean + (1.96 * prediction_stdv)).tolist()
lower = (prediction_mean - (1.96 * prediction_stdv)).tolist()
prediction_stdv = prediction_stdv.tolist()

for idx in range(sample):
    print(
        f"Prediction mean: {round(prediction_mean[idx][0], 2)}, "
        f"stddev: {round(prediction_stdv[idx][0], 2)}, "
        f"95% CI: [{round(upper[idx][0], 2)} - {round(lower[idx][0], 2)}]"
        f" - Actual: {targets[idx]}"
    )
```

```
Prediction mean: 5.29, stddev: 0.66, 95% CI: [6.58 - 4.0] - Actual: 6.0
Prediction mean: 6.49, stddev: 0.81, 95% CI: [8.08 - 4.89] - Actual: 6.0
Prediction mean: 5.85, stddev: 0.7, 95% CI: [7.22 - 4.48] - Actual: 7.0
Prediction mean: 5.59, stddev: 0.69, 95% CI: [6.95 - 4.24] - Actual: 5.0
Prediction mean: 6.37, stddev: 0.87, 95% CI: [8.07 - 4.67] - Actual: 5.0
Prediction mean: 6.34, stddev: 0.78, 95% CI: [7.87 - 4.81] - Actual: 7.0
Prediction mean: 5.14, stddev: 0.65, 95% CI: [6.4 - 3.87] - Actual: 5.0
Prediction mean: 6.49, stddev: 0.81, 95% CI: [8.09 - 4.89] - Actual: 6.0
Prediction mean: 6.25, stddev: 0.77, 95% CI: [7.76 - 4.74] - Actual: 6.0
Prediction mean: 6.39, stddev: 0.78, 95% CI: [7.92 - 4.85] - Actual: 7.0
```