towards
data science

Follow 573K Followers

Bayesian Neural Networks with TensorFlow Probability

A step by step guide to uncertainty prediction with probabilistic modeling.

Georgi Tancev Jan 31, 2020 · 6 min read ★

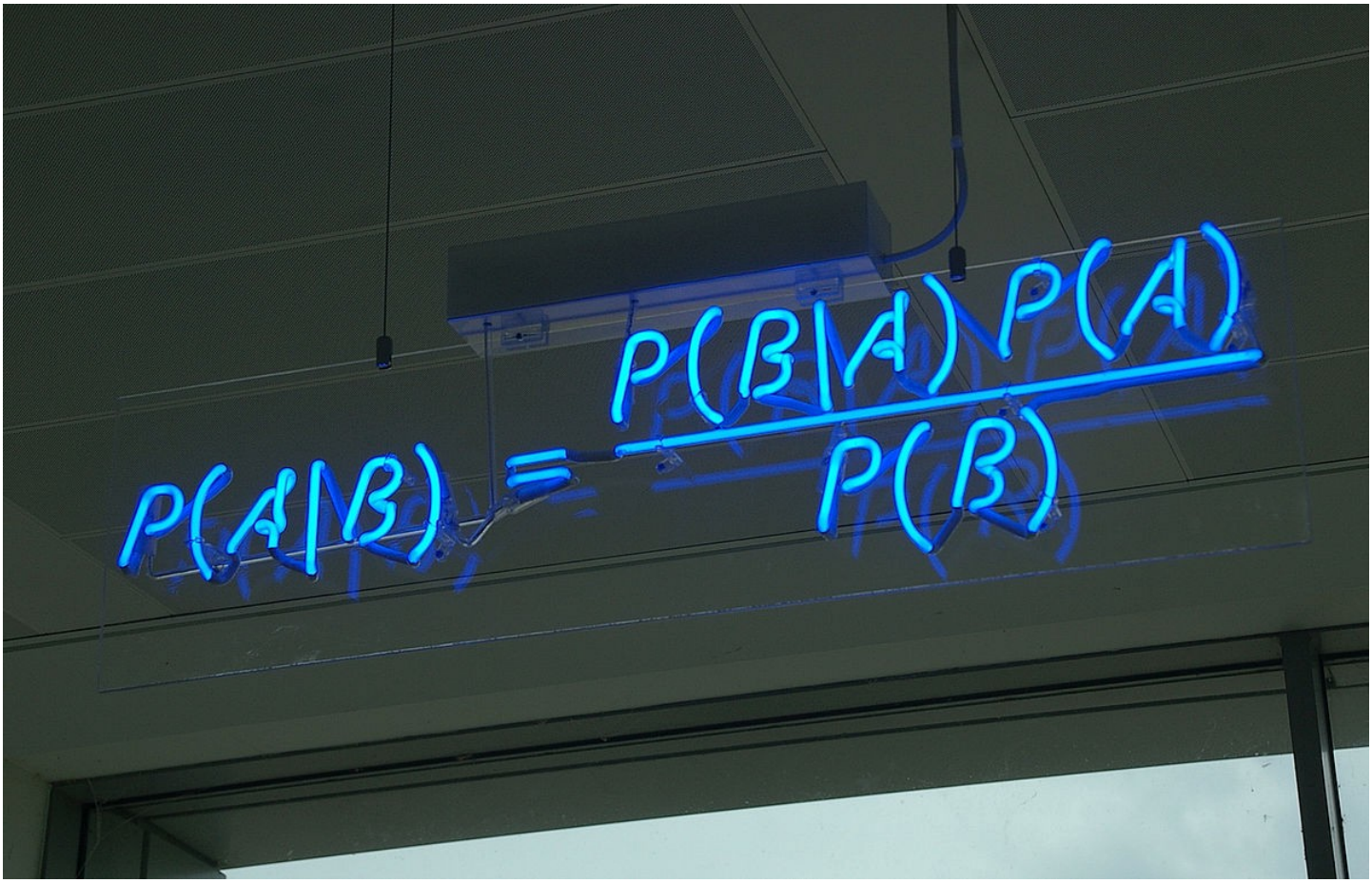


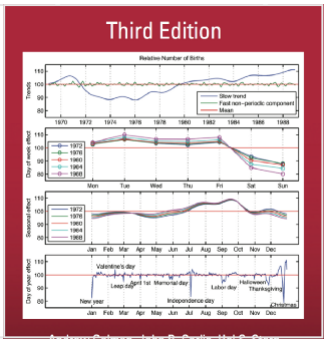
Image via Wikipedia.

that map input to output using a **point estimate** of parameter weights calculated by **maximum-likelihood** methods. However, there is a lot of statistical fluke going on in the background. For instance, a dataset itself is a finite random set of points of arbitrary size from a unknown distribution superimposed by additive noise, and for such a particular collection of points, different models (i.e. different parameter combinations) might be reasonable. Hence, there is some **uncertainty** about the parameters and predictions being made. **Bayesian statistics** provides a framework to deal with the so-called **aleoteric and epistemic** uncertainty, and with the release of **TensorFlow Probability**, **probabilistic modeling** has been made a lot easier, as I shall demonstrate with this post. Be aware that no theoretical background will be provided; for theory on this topic, I can really recommend the book “Bayesian Data Analysis” by Gelman et al., which is available as PDF-file for free.

Home page for the book, "Bayesian Data Analysis"

Here is the book in pdf form, available for download for non-commercial purposes.

www.stat.columbia.edu



Bayesian Neural Networks

A **Bayesian neural network** is characterized by its distribution over weights (parameters) and/or outputs. Depending on wether aleotoric, epistemic, or both uncertainties are considered, the code for a Bayesian neural network looks slighty different. To demonstrate the working principle, the Air Quality dataset from De Vito will serve as an example. It contains data from different chemical sensors for pollutants (as voltage) together with references as a year-long time series, which has been collected at a main street in an Italian city characterized by heavy car traffic, and the goal is to construct a mapping from sensor responses to reference concentrations (Figure 1), i.e. building a calibration function as a **regression** task.

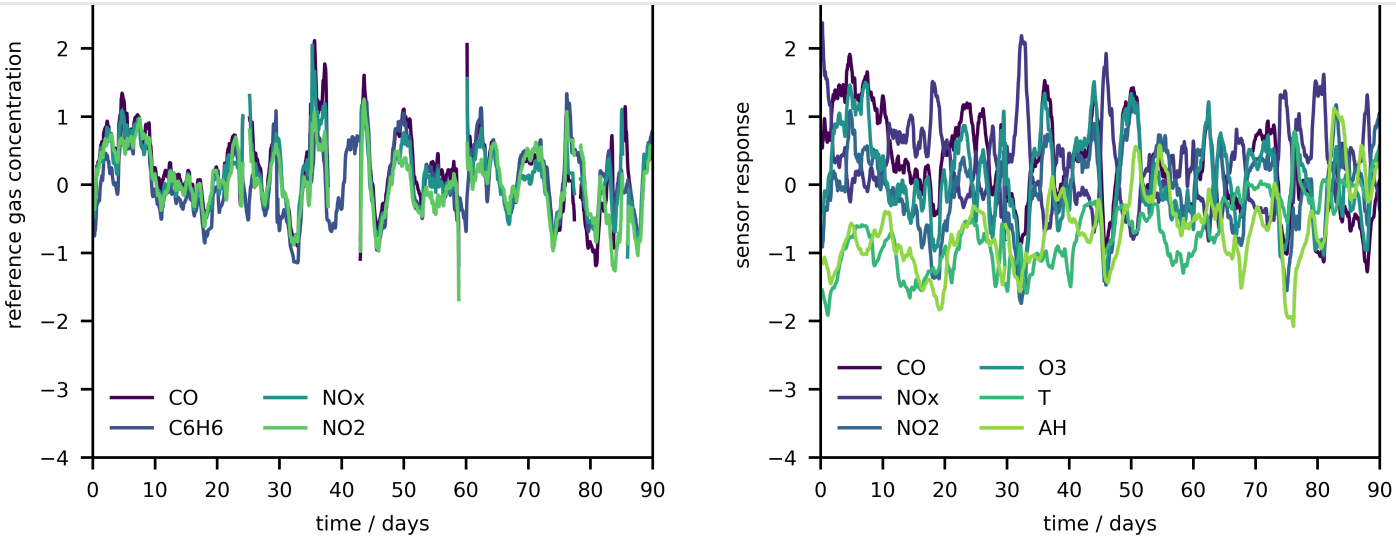


Figure 1: Reference and sensor data. Note the correlation within the reference data. (image by author)

Start-Up

If you have not installed TensorFlow Probability yet, you can do it with pip, but it might be a good idea to create a virtual environment before. (Since commands can change in later versions, you might want to install the ones I have used.)

```
# Install libraries.

pip install tensorflow==2.1.0
pip install tensorflow-probability==0.9.0
```

Open your favorite editor or JupyterLab. Import all necessarty libraries.

```
# Load libriaries and functions.

import pandas as pd
import numpy as np
import tensorflow as tf
tfk = tf.keras
tf.keras.backend.set_floatx("float64")
import tensorflow_probability as tfp
tfd = tfp.distributions
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import IsolationForest

# Define helper functions.
```

```
neg_log_likelihood = lambda x, iv_x: -iv_x.log_prob(x)
```

Next, grab the dataset (link can be found above) and load it as a pandas dataframe. As sensors tend to drift due to aging, it is better to discard the data past month six.

```
# Load data and keep only first six months due to drift.

data = pd.read_excel("data.xlsx")
data = data[data["Date"] <= "2004-09-10"]
```

Preprocessing

The data is quite messy and has to be preprocessed first. We will focus on the inputs and outputs which were measured for most of the time (one sensor died quite early). Data is scaled after removing rows with missing values. Afterwards, outliers are detected and removed using an **Isolation Forest**.

```
# Select columns and remove rows with missing values.

columns = ["PT08.S1(CO)", "PT08.S3(NOx)", "PT08.S4(NO2)",
"PT08.S5(O3)", "T", "AH", "CO(GT)", "C6H6(GT)", "NOx(GT)",
"NO2(GT)"]
data = data[columns].dropna(axis=0)

# Scale data to zero mean and unit variance.

X_t = scaler.fit_transform(data)

# Remove outliers.

is_inlier = detector.fit_predict(X_t)
X_t = X_t[(is_inlier > 0),:]

# Restore frame.

dataset = pd.DataFrame(X_t, columns=columns)

# Select labels for inputs and outputs.

inputs = ["PT08.S1(CO)", "PT08.S3(NOx)", "PT08.S4(NO2)",
"PT08.S5(O3)", "T", "AH"]
outputs = ["CO(GT)", "C6H6(GT)", "NOx(GT)", "NO2(GT)"]
```

of the data as training set. The sets are shuffled and repeating batches are constructed.

```
# Define some hyperparameters.

n_epochs = 50
n_samples = dataset.shape[0]
n_batches = 10
batch_size = np.floor(n_samples/n_batches)
buffer_size = n_samples

# Define training and test data sizes.

n_train = int(0.7*dataset.shape[0])

# Define dataset instance.

data = tf.data.Dataset.from_tensor_slices((dataset[inputs].values,
dataset[outputs].values))
data = data.shuffle(n_samples, reshuffle_each_iteration=True)

# Define train and test data instances.

data_train = data.take(n_train).batch(batch_size).repeat(n_epochs)
data_test = data.skip(n_train).batch(1)
```

Model Building

Aleotoric Uncertainty

To account for aleotoric uncertainty, which arises from the noise in the output, dense layers are combined with probabilistic layers. More specifically, the mean and covariance matrix of the output is modelled as a function of the input and parameter weights. The first hidden layer shall consist of ten nodes, the second one needs four nodes for the means plus ten nodes for the variances and covariances of the four-dimensional (there are four outputs) multivariate Gaussian posterior probability distribution in the final layer. This is achieved using the `params_size` method of the last layer (`MultivariateNormalTriL`), which is the declaration of the **posterior probability distribution** structure, in this case a multivariate normal distribution in which only one half of the covariance matrix is estimated (due to symmetry). The total number of parameters in the model is 224 — estimated by **variational methods**. The deterministic version of this neural network consists of an input layer, ten **latent**

```
# Define prior for regularization.

prior = tfd.Independent(tfd.Normal(loc=tf.zeros(len(outputs),
dtype=tf.float64), scale=1.0), reinterpreted_batch_ndims=1)

# Define model instance.

model = tfk.Sequential([
tfk.layers.InputLayer(input_shape=(len(inputs),), name="input"),
tfk.layers.Dense(10, activation="relu", name="dense_1"),
tfk.layers.Dense(tfp.layers.MultivariateNormalTriL.params_size(
len(outputs)), activation=None, name="distribution_weights"),
tfp.layers.MultivariateNormalTriL(len(outputs),
activity_regularizer=tfp.layers.KLDivergenceRegularizer(prior,
weight=1/n_batches), name="output")
], name="model")

# Compile model.

model.compile(optimizer="adam", loss=neg_log_likelihood)

# Run training session.

model.fit(data_train, epochs=n_epochs, validation_data=data_test,
verbose=False)

# Describe model.

model.summary()
```

The `activity_regularizer` argument acts as prior for the output layer (the weight has to be adjusted to the number of batches). The training session might take a while depending on the specifications of your machine. The algorithm needs about 50 epochs to converge (Figure 2).

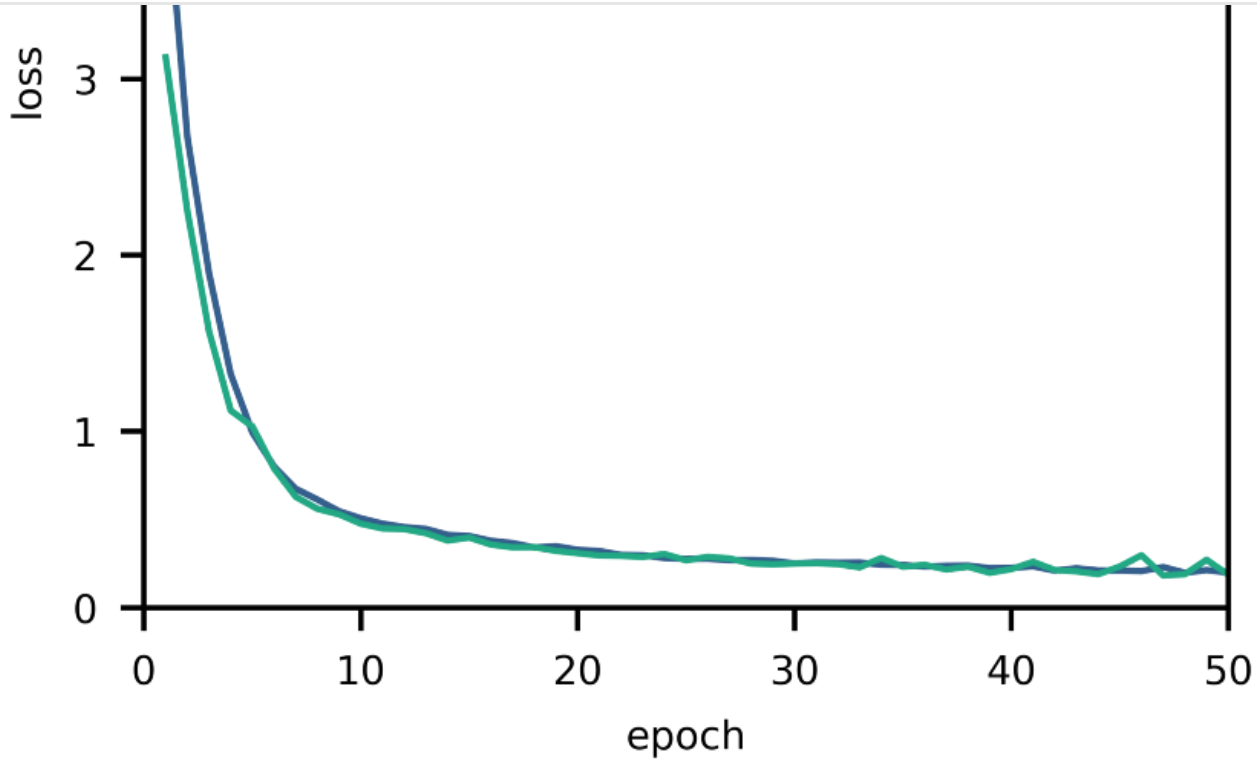


Figure 2: Learning curve with training and validation loss. (image by author)

Aleotoric and Epistemic Uncertainty

To account for aleotoric and epistemic uncertainty (uncertainty in parameter weights), the dense layers have to be exchanged with **Flipout** layers (`DenseFlipout`) or with **Variational** layers (`DenseVariational`). Such a model has more parameters, since every weight is parametrized by normal distribution with non-shared mean and standard deviation, hence doubling the amount of parameter weights. Weights will be resampled for different predictions, and in that case, the Bayesian neural network will act like an **ensemble**.

```
tfp.layers.DenseFlipout(10, activation="relu", name="dense_1")
```

The default prior distribution over weights is `tfd.Normal(loc=0., scale=1.)` and can be adjusted using the `kernel_prior_fn` argument.

Prediction

Since it is a probabilistic model, a **Monte Carlo experiment** is performed to provide a prediction. In particular, every prediction of a sample x results in a different output y,

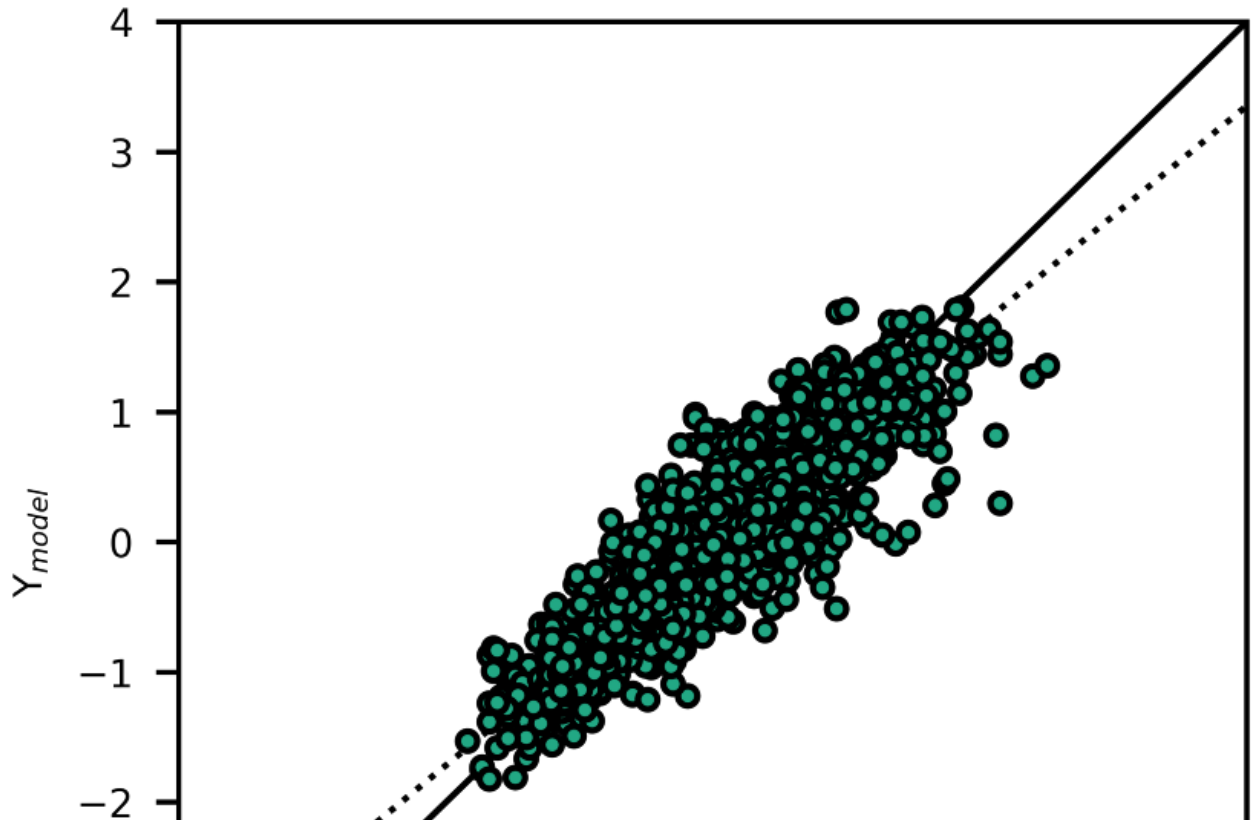
```
# Predict.

samples = 500
iterations = 10
test_iterator = tf.compat.v1.data.make_one_shot_iterator(data_test)
X_true, Y_true, Y_pred = np.empty(shape=(samples, len(inputs))),
np.empty(shape=(samples, len(outputs))), np.empty(shape=(samples,
len(outputs), iterations))
for i in range(samples):
    features, labels = test_iterator.get_next()
    X_true[i,:] = features
    Y_true[i,:] = labels.numpy()
    for k in range(iterations):
        Y_pred[i,:,k] = model.predict(features)

# Calculate mean and standard deviation.

Y_pred_m = np.mean(Y_pred, axis=-1)
Y_pred_s = np.std(Y_pred, axis=-1)
```

Figure 3 shows the measured data versus the expectation of the predictions for all outputs. The **coefficient of determination** is about 0.86, the **slope** is 0.84 — not too bad.



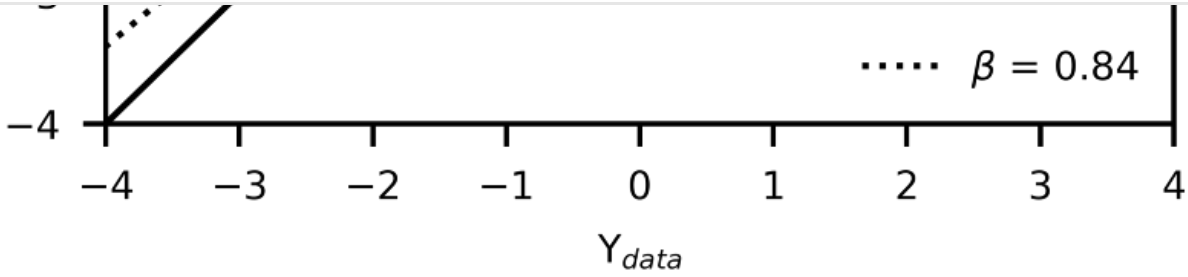


Figure 3: Agreement between measurements and predictions. (image by author)

Predicted uncertainty can be visualized by plotting error bars together with the expectations (Figure 4). In this case, the error bar is 1.96 times the standard deviation, i.e. accounting for 95% of the probability.

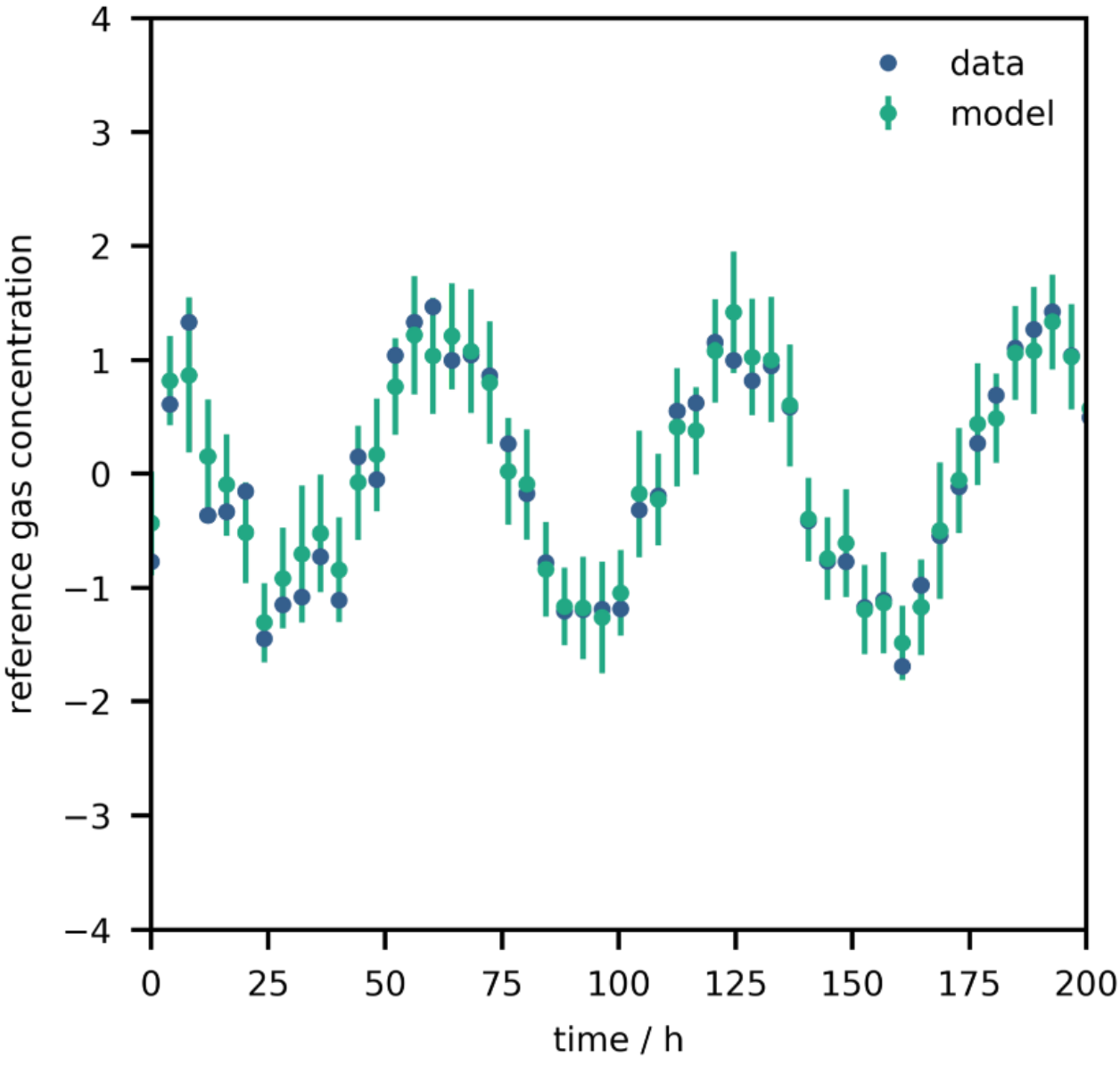


Figure 4: Comparison of data versus predictive (image by author)

In theory, a Bayesian approach is superior to a deterministic one due to the additional uncertainty information, but not always possible because of its high computational costs. Recent research revolves around developing novel methods to overcome these limitations.