Fortgeschrittene Programmierung in der Physik SE

Institute of Theoretical Physics
Computational Physics

Introduction to Baysian Neural Networks with examples in Physics and Engineering

Advisor:
Univ.-Prof. Dipl.-Phys. Dr. Wolfgang von der Linden

Tobias Leitgeb
Mat.Nr. 12006992

Summer term 2024

1 Introduction

In this report we will look at the basics of Baysian Neural Networks, how to implement them and at some specific exemplary problems. Additionally, there will be a short introduction to Baysian PINNs [**Yang˙2021**]. I want to give a quick overview of the mathematics needed for defining a Baysian Neural Network, training it and predicting new samples. For implementing the BNN, *NumPyro* [**bingham2019pyro**, **phan2019composable**] is used, which is a probabilistic library build in *JAX* [**jax2018github**]. This *JAX* backend provides very fast and efficient computing through access to JIT compilation, automatic differentiation and support of CUDA. Through some examples, the basic of *NumPyro* and *JAX* will be shown. At the end, there will be two examples for using a BNN for a regression task. The complete code used in this report can be found at my github.

2 Mathematical basics

2.1 Baysian Neural Networks

A Baysian Neural Network is a Neural Network with stochastic features, like probability distributions for the weights or for the activation functions [**BNNTut**]. For training general Neural Networks, one maximizes the likelihood $p(\mathcal{D}|\boldsymbol{\theta})$ to obtain a point estimate $\boldsymbol{\theta}^*$:

$$\boldsymbol{\theta}^* = \text{argmax}_{\boldsymbol{\theta}} p(\mathcal{D}|\boldsymbol{\theta}) \tag{1}$$

The training of the BNN on the other hand, is done by calculating the posterior distribution of the parameters $p(\boldsymbol{\theta}|\mathcal{D})$, given the training data $\mathcal{D}$. This has some advantages over general MLE or MAP estimates, since it is less likely to lead to overfitting [**BNNdistill**]. This posterior is given by Bayes rule:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{\int_{\theta'} p(\mathcal{D}|\boldsymbol{\theta}')p(\boldsymbol{\theta}')} \tag{2}$$

with the likelihood $p(\mathcal{D}|\boldsymbol{\theta})$, the prior for the parameters $p(\boldsymbol{\theta})$ and the integral for the evidence $\int_{\theta'} p(\mathcal{D}|\boldsymbol{\theta}')p(\boldsymbol{\theta}')$. This integral over all possible parametrizations of the Network is, most of the time, intractable due to its high dimensionality, which is especially true for deep Neural Networks. Therefore, we need to somehow approximate this posterior distribution. There are several methods for doing inference, e.g *Monte Carlo dropout, Markov Chain Monte Carlo, Laplace approximation, Variational Inference, Deep ensembles* etc. [**murphy**]. In this report, the focus will be on two inference methods, which are arguably the most robust and widely used algorithms for BNNs [**BNNTut**]:

- Markov Chain Monte Carlo
- Variational Inference

2.2 Markov Chain Monte Carlo

A very strong quality of *MCMC* methods is that they do not restrict or assume the form of the posterior, which we will later see, has to be done when using variational methods. *MCMC* is especially a good choice, when the Network is rather thin with a small number of neurons and layers. This is because *MCMC* does not scale that well when the dimension of the parameters greatly increases. However, efforts have been made for adapting *MCMC* for bigger BNNs [**murphy**]. Later on, this behaviour will be shown by specific examples. Another drawback is that the whole training set has to be used, and therefore kept in memory, for every training step. This further limits the capabilities for big datasets and deep Networks.
NumPyro allows a very simple and clear syntax for defining and doing inference with *MCMC*. In this example we use the so called NUTS algorithm, which is a variant of Hamilton Monte Carlo [**Brooks˙2011**, **hoffman2011nouturnsampleradaptivelyset**

Listing 1: Inference with NUTS

```python
from numpyro.infer import NUTS, MCMC
#define pseudo random number generator key
inf_key = jax.random.PRNGKey(0)
#define mcmc with the NUTS sampler, the network and the number of samples
mcmc = MCMC(NUTS(BNN_model), num_warmup=1000, num_samples=1000)
#running inference with the training data
mcmc.run(inf_key, X_train, Y_train, LAYERS)
mcmc.print_summary()
```

After running the inference, the summary gives an overview of the calculated samples for the parameters of the network:

Listing 2: Results of Inference with HMC

```
sample: 100%|                              | 2000/2000 [04:36<00:00,  7.23it/s, 1023 steps of size
    6.49e-04. acc. prob=0.93]

                              mean       std    median      5.0%     95.0%     n_eff     r_hat
                 W0[0,0]     -0.06      0.98      0.06     -1.71      1.31     12.27      1.10
                 W0[0,1]     -0.14      0.82     -0.11     -1.71      1.05     33.64      1.10
                 W0[0,2]     -0.03      0.84     -0.09     -1.29      1.37     25.33      1.21
                 W0[0,3]     -0.04      0.92     -0.03     -1.66      1.33     15.52      1.10
                 W0[0,4]     -0.24      0.72     -0.22     -1.44      0.90     27.59      1.06
                 W0[0,5]      0.12      0.95      0.10     -1.46      1.72     40.72      1.06
                 W0[0,6]      0.14      0.67      0.10     -0.96      1.27     61.09      1.00
                 W0[0,7]      0.22      0.95      0.13     -1.22      1.88     25.26      1.01
                 W0[0,8]     -0.17      0.89     -0.15     -1.70      1.17     29.00      1.03
                 W0[0,9]     -0.05      0.79     -0.03     -1.42      1.24     38.40      1.05
                W0[0,10]      0.06      0.86      0.04     -1.46      1.43     13.57      1.13
                W0[0,11]      0.20      0.88      0.23     -1.10      1.85     30.89      1.05
                W0[0,12]      0.16      0.87      0.23     -1.37      1.62     32.36      1.12
                W0[0,13]      0.03      0.95     -0.05     -1.56      1.59     24.91      1.00
...
                   b2[0]      0.26      0.94      0.29     -1.18      1.81     34.99      1.01
  observation precision     44.33      6.95     44.19     34.01     56.20     14.91      1.08

Number of divergences: 2
```

2.3 Variational Inference

In variational inference the idea is to approximate the posterior distribution with a parameterized approximated distribution $q_\phi(\boldsymbol{\theta})$. The parameters ϕ of this distribution are now optimized, such that they best describe the posterior. For this the Kullback-Liebler divergence is used as a measure of similarity between two probability distributions p and q [**murphy**, **BNNdistill**]:

$$\begin{aligned} d_{KL}[q(\boldsymbol{\theta})||p(\theta|\mathcal{D})] &= \int_{\boldsymbol{\theta}'} q_\phi(\boldsymbol{\theta}) \log \frac{q_\phi(\boldsymbol{\theta})}{p(\boldsymbol{\theta}|\mathcal{D})} d\boldsymbol{\theta} \\ &= \mathbb{E}_{q_\phi(\boldsymbol{\theta})}[\log(q_\phi(\boldsymbol{\theta}) - \log p(\boldsymbol{\theta}|\mathcal{D})] \end{aligned} \tag{3}$$

The optimization problem can then be formulated as [**murphy**]:

$$q = \arg\min_q d_{KL}[q(\boldsymbol{\theta})||p(\theta|\mathcal{D}] \tag{4}$$

By this optimization, we can obtain the parameters ϕ of the variational distribution For solving this optimization problem, (**??**) has to be rewritten, since the posterior cannot be directly computed.

$$\phi = \arg\min_\phi d_{KL}[q(\boldsymbol{\theta})||p(\theta|\mathcal{D}] \tag{5}$$

$$= \arg\min_\phi \mathbb{E}_{q_\phi(\boldsymbol{\theta}}[\log q_\phi(\boldsymbol{\theta}) - \log p(\boldsymbol{\theta}|\mathcal{D})]$$

$$= \arg\max_\phi \underbrace{\mathbb{E}_{q_\phi(\boldsymbol{\theta}}[\log p(\boldsymbol{\theta},\mathcal{D}) - \log q_\phi(\boldsymbol{\theta})]}_{\tilde{\mathcal{L}}(\phi)=\text{ELBO}} + \log p(\mathcal{D}) \tag{6}$$

$$\tag{7}$$

where the marginal data likelihood $p(\mathcal{D})$ can be left out for the optimization of ϕ, since it does not depend on the variational parameters. Here the evidence lower bound (ELBO) is introduced. The name of the ELBO comes from the fact that it is the lower bound of the evidence $p(\mathcal{D})$, s.t. [**murphy**]:

$$\tilde{\mathcal{L}}(\phi) = \mathbb{E}_{q_\phi(\boldsymbol{\theta}}[\log p(\boldsymbol{\theta},\mathcal{D}) - \log q_\phi(\boldsymbol{\theta})] \leq p(\mathcal{D}) \tag{8}$$

With this reformulation, the optimization of the variational parameters is reformulated as the maximization of the ELBO $\tilde{\mathcal{L}}$ with respect to ϕ. For actually doing this optimization, there are several methods. One of the most popular is *Stochasitc Variationl inference* [**hoffman2013stochasticvariationalinference**]. SVI generally works just like stochastic gradient ascent [**BNNTut**] and comes with advantages like minibatching. Other methods are *Bayes by backpropagation*, *Laplace approximation* or *Deep ensembles*.
At last, we briefly want to demonstrate how to do VI in *NumPyro*:

Listing 3: Inference with VI

```
from numpyro.infer.autoguide import AutoDiagonalNormal, AutoLaplaceApproximation,
    AutoMultivariateNormal, AutoIAFNormal, AutoLowRankMultivariateNormal, AutoDelta
#define the
guide = AutoDiagonalNormal(BNN_model)

svi = SVI(
    wideband_dnn, guide, numpyro.optim.Adam(0.05), Trace_ELBO()
)

svi_result = svi.run(
    jax.random.PRNGKey(0),
    num_steps=3000,
    X=X_train,
    Y=Y_train,
    layers=LAYERS
)
```

2.4 Predictive distribution

After having obtained samples, or a variational approximation of the posterior $p(\boldsymbol{\theta}|\mathcal{D})$, predictions at unseen test points $\boldsymbol{x}_*$ can be made. For this we have to formulate a distribution $p(f(\boldsymbol{x}_*|\mathcal{D})$, which is obtained by marginalization of the calculated parameters $\boldsymbol{\theta}$:

$$p(\boldsymbol{f}(\boldsymbol{x}_*)|\mathcal{D}) = \int_{\boldsymbol{\theta}} p(\boldsymbol{f}(\boldsymbol{x}_*)|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} = \mathbb{E}_{p(\theta|\mathcal{D})}\left[p(\boldsymbol{f}(\boldsymbol{x}_*)|\boldsymbol{\theta})\right] \tag{9}$$

Writing the integral in terms of an conditional expectation of the posterior distribution of $\boldsymbol{\theta}$, gives an interesting view of the marginalized distribution, as the exptectation value of a single neural net output wheighted by the posterior ditribution of the parameters given the training data. In [**blundell2015weightuncertaintyneuralnetworks**] this is described as an *infinite ensenmble* of Neural Networks. In practice we can not calculate an infinite number of networks. However when having obtained enough iid. samples (monte carlo samples) of the posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$, the expectation value in (**??**) can be approximated. Generating samples of the posterior is slightly different for the two inference algorithms discussed above. When using MCMC, ($\approx$ independent) samples of the posterior are obtained directly. For every sample $\boldsymbol{\theta}_i$, which is a set of parameters, the networks output $p(\boldsymbol{f}(\boldsymbol{x}_*)|\boldsymbol{\theta}_i)$ is computed. We now approximate the expectation value by taking the mean over all the samples. When using VI the process is nearly the same. After having obtained the trained variational distribution q, we can sample from that distribution and again do the same thing as described above with the samples.

2.5 Physics informed Baysian Neural Network

In this section, the baysian equivalence of a PINN, a so called B-PINN [**Yang˙2021**], is quickly introduced and later on used for a simple example.
For PINNs the loss function $\mathcal{L}$ is augmented with an additional physics term:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_{data} + \mathcal{L}_{physics} \tag{10}$$

This physics loss is oftentimes defined through a differential equation:

$$\mathcal{P}_\phi u(\boldsymbol{x}) = f(\boldsymbol{x}) \tag{11}$$

with $\mathcal{P}_\phi$ being a parametric differential operator with parameters ϕ, u the solution of the differential equation and f the forcing term. Lets assume that we have a data set $\mathcal{D}$ consisting of noisy observations of both u and f $\{(\boldsymbol{x_u}); \boldsymbol{y_u}\}$ and $\{(\boldsymbol{x_f}); \boldsymbol{y_f}\}$ with $\boldsymbol{y_i} = u(\boldsymbol{x_i}) + \mathcal{N}(\boldsymbol{0}|\sigma_{ni}^2 I);\ i = u, f$. The solution term u is now parameterized with a Neural Network $\tilde{u}$:

$$\mathcal{P}_\phi \tilde{u}(\boldsymbol{x}; \boldsymbol{\theta}) = f(\boldsymbol{x}) \tag{12}$$

Since we are working with libraries that have access to automatic differentiation, the derivative of $\tilde{u}$, and therefore the forcing term f can be calculated directly:

$$\tilde{f}(\boldsymbol{x}; \boldsymbol{\theta}, \phi) = \mathcal{P}_\phi \tilde{u}(\boldsymbol{x}; \boldsymbol{\theta})) \tag{13}$$

with $\tilde{f}$ as the Since we have observations for both the u and f we can calculate the residuals for both, which leads to our two loss terms:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_{data}(\tilde{u}; \boldsymbol{\theta}) + \lambda \mathcal{L}_{physics}(\tilde{f}(\theta)) \tag{14}$$

were $\mathcal{L}$ would e.g be the MSE.

Listing 4: Physics Informed Neural Net

```
def physics_forward(
        W: List[jax.Array],
        b: List[jax.Array],
        X: jax.Array,
        activation: Callable,
):
    #input layer
    u = partial(forward, W, b, activation=activation)
    u_prime = jax.grad(u)
    u_double_prime = jax.grad(u_prime)

    #vectorize the functions
    u_prime = jax.vmap(u_prime)(X)
    u_double_prime = jax.vmap(lambda x: u_double_prime(x.squeeze()))(X)
    return u_prime, u_double_prime[:, None]
```

To formulate this in terms of baysian NN, the posterior is augmented with the additional likelihood for the function f:

$$p(\boldsymbol{\theta}, \phi|\mathcal{D}_u, \mathcal{D}_f) = \frac{p(\mathcal{D}_u, \mathcal{D}_f|\boldsymbol{\theta}, \phi)p(\boldsymbol{\theta})p(\phi)}{\int_{\theta'} p(\mathcal{D}_u, \mathcal{D}_f|\boldsymbol{\theta}')p(\boldsymbol{\theta}')} \tag{15}$$

with the prior distribution for the parameters of the differential equation. If the parameters ϕ are a-priori unknown, they can be assigned a prior function $p(\phi)$ and then be learned directly through inference [**raissi2017physicsIDL**, **Yang˙2021**]. The joint likelihood can be implemented in *NumPyro* as:

Listing 5: Joint likelihood for B-PINN

```
z = forward(W, b, X_u, activation)[:, None]
dz, ddz = physics_forward(W, b, X_f, activation)
f = m * ddz + gamma * dz + k * z

#joint likelihood
numpyro.sample(
    r"Y_u",
    dist.Normal(z, sigma_obs_u).to_event(1),
    obs=Y_u
)
numpyro.sample(
    r"Y_f",
    dist.Normal(f, sigma_obs_f).to_event(1),
    obs=Y_f
)
```

3 Setting up the model

A very default prior for the parameters of the BNN which has proven to work relatively well [**BNNTut**] is:

$$\boldsymbol{W} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}) \tag{16}$$

$$\boldsymbol{b} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}) \tag{17}$$

where we just use two normal distributions as priors for the wheigts and biases. In the used probabilistic framework NumPyro we can define the priors the following way:

Listing 6: Setting up priors for wheits

```
    W = numpyro.sample(f"W{i}", dist.Normal(0, 1).expand(n_in, n_out))
    b = numpyro.sample(f"b{i}", dist.Normal(0, 1).expand((n_out, )))
```

Generally, other priors can also be used, however the normal distributions lead to good results. Two other ways of initializing the weights are e.g Xavier initialization $\alpha_l^2 = \frac{2}{n_{in}+n_{out}}$ or LeCun initialization $\alpha_l^2 = \frac{1}{n_{in}}$ [**murphy**]. The weights would then be initialized according to:

$$\boldsymbol{W}_l \sim \mathcal{N}(\mathbf{0}, \alpha_l^2\boldsymbol{I}); \ \boldsymbol{b}_l \sim \mathcal{N}(\mathbf{0}, \alpha_l^2\boldsymbol{I}) \tag{18}$$

Here $\boldsymbol{W}_i$ and $\boldsymbol{b}_i$ are the corresponding parameters for the i-th layer. Another very important step is to normalize the data:

$$y_{i,\text{normalized}} = \frac{y_i - \mu_{\boldsymbol{Y}}}{\sigma_{\boldsymbol{Y}}} \tag{19}$$

since there was very poor performance without the normalization. This is necessary, since the priors of the weights are centered around 0 with a variance of 1. Without normalizing the data, one could run into problems getting valid results. For making predictions after training, the values are transformed back:

$$\hat{y}_i = \hat{y}_{i,\text{normalized}} \cdot \sigma_{\boldsymbol{Y}} + \mu_{\boldsymbol{Y}} \tag{20}$$

For the feed-forward BNN we can also define a forward pass, which can be written as:

$$\boldsymbol{f}(\boldsymbol{x}|\boldsymbol{\theta}) = \sum_i w_{ji}^{(L)} \left(\dots \phi \left(\sum_k w_{mk}^{(1)} x_k + \boldsymbol{b}^{(1)} \right) \right) + \boldsymbol{b}^{(L)} \tag{21}$$

After having obtained the samples through inference, the forward pass is the same as for a standard feed-forward neural networks In *JAX* this can be implemented as:

Listing 7: Feed forward Neural Network

```python
def forward(
        W: List[jax.Array],
        b: List[jax.Array],
        X: jax.Array,
        activation: Callable,
):
    #input layer
    z = activation(jnp.dot(X, W[0]) + b[0])

    #hidden layers
    for i in range(1, len(W) - 1):
        z = activation(jnp.dot(z, W[i]) + b[i])

    #output layer with no activation
    z = jnp.dot(z, W[-1]) + b[-1]
    return z.squeeze()
```

With this forward pass and the priors for the parameters θ, the whole model can be set up as a relativly easy function:

Listing 8: NumPyro model for baysian deep neural network

```python
def BNN_model(
        X: jax.Array,
        Y: jax.Array,
        layers: List[int],
):
    N, input_dim = X.shape
    activation = jax.nn.relu
    W = []
    b = []
    #build the layers with the given list
    for i, layer in enumerate(layers):
        W_, b_ = dense_layer(i, [input_dim, layer])
        W.append(W_)
        b.append(b_)
        input_dim = layer
    #forward pass through the network
    z = forward(W, b, X, activation)

    precision_obs = numpyro.sample(r"observation precision", dist.Gamma(3., 1.))
    sigma_obs = 1.0 / jnp.sqrt(precision_obs)

    with numpyro.plate("data", N):
        numpyro.sample(
            "Y",
            dist.Normal(z, sigma_obs).to_event(1),
            obs=Y
        )
```

When using Variational inference for the inference of the posterior of the parameters, there are no direct samples for parameters. Therefore, the trained variational distribution q (called guide here) has to be used to sample $\boldsymbol{\theta}$: For predicting new values at $\boldsymbol{x}_*$, we draw samples from the trained variational distribution q_θ

Listing 9: Prediction of new samples with VI

```python
def predict_new(
        svi_result,
        guide,
        X_test: jax.Array,
        num_samples: int = 500,
        *,
        key: jr.key,
        return_mean: bool = True,
    )-> Tuple:

    sample_key, prediction_key = jr.split(key, 2)
    #sample from the trained guide q_phi(theta)
    samples = guide.sample_posterior(
        sample_key, svi_result.params, sample_shape=(num_samples, )
    )
    #define predictive distribution with samples and the model.
    predictive_distribution = Predictive(
        BNN_model, samples, return_sites=["Y"]
    )
    #sample from the predictive distribution given new test points
    predictions = predictive_distribution(
        prediction_key, X_test, None, LAYERS
    )
    if return_mean is True:
        return jnp.mean(predictions["Y"], axis=0), jnp.var(predictions["Y"], axis=0)
    else:
        return jnp.mean(predictions["Y"], axis=0), jnp.var(predictions["Y"], axis=0), predictions
```

4 Numerical examples

The error functions to quantify the testing error are the relative L2 error and the Normalized Root Mean Squared Error (NRMSE):

$$rel\ L^2 = \sqrt{\frac{\sum_i(\hat{y}_i - y)^2}{\sum_i(y_i)^2}} \tag{22}$$

$$\text{NRMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y)^2} \tag{23}$$

4.1 Transistor surrogate model/Transformer model akash

Here the data set consists of an inputs $\boldsymbol{x}$, which specify the circuit layout and of the envelope of the fourier transformed output of the simulation data $\boldsymbol{y}$ over a specified frequency range.

$$X = \begin{bmatrix} \boldsymbol{x}_1^T, \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}, \ Y = \begin{bmatrix} \boldsymbol{y}_1^T \\ \boldsymbol{y}_2^T \\ \vdots \\ \boldsymbol{y}_N^T \end{bmatrix} \tag{24}$$

with $\boldsymbol{x}_i \in \mathbb{R}^d$ and $\boldsymbol{y}_i \in \mathbb{R}^p$. The vector $\boldsymbol{x}_i$ is the design vector for the circuit, with the parameters which are beeing adapted. For this example we have:

$$x_i = [...] \tag{25}$$

Therefore, we get two matrices $\boldsymbol{X}, \boldsymbol{Y}$ with shapes (N, d) and (N, p), where N is the number of training points, d is the dimensionality of the input and p is the number of frequency outputs. The BNN is designed as: $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta}) : \mathbb{R}^d \to \mathbb{R}^p$ to yield an output for the whole frequency range. It would also be possible to use the frequency as a part of the input $\boldsymbol{x}$. However, this would highly increase the complexity of the input space and would not yield better results. The used model architecture was 2 hidden layers with 128 neurons and an output layer of size p of the frequency points. The *ReLu* function was used as activation. In this application the goal is to define a surrogate model for the transformer LT spice model. For this example 500 configurations were simulated. For training we used 200 samples. BNNs are interesting for surrogate modelling, since they also give an uncertainty quantification. When looking at the samples and at the calculated predictions, it is interesting to see that the model quantifies the uncertainty somewhat wrong especially in the are from $10^6 - 10^7$Hz. Between different circuit parameterizations, there is nearly no difference in these regions. The mean of the drawn samples describes the function well, while the predicted uncertainty is predicted way to high. Here, more work would need to be done to find a more suitable model architecture. The construction of an accurate surrogate for problems in power electronics/EMC is an ongoing topic, for which baysian neural networks could be interesting.

`plots/transformer/data_samples.png`

Figure 1: Training samples $\boldsymbol{Y}_i$ plotted over the frequency domain. One curve is the response over the frequency range for a circuit parameterized by the respective $\boldsymbol{X}_i$.

4.2 Damped Oscillator

Let's consider a one-dimensional differential equation that models a damped oscillator, characterized by a mass m, damping coefficient γ, force constant k, and initial conditions. The equation governing the system is:

$$m\frac{\partial^2 u(t)}{\partial t^2} + \gamma\frac{\partial u(t)}{\partial t} + ku(t) = f(t), \tag{26}$$

$$u(0) = 0, \quad \left.\frac{\partial u(t)}{\partial t}\right|_{t=0} = 0 \tag{27}$$

The specific case we're examining uses the parameters $m = 1$, $\gamma = 0.2$, and $k = 2$, with an external force described by:

$$f(t) = 10\sin(\pi t)\cos(4\pi t) \tag{28}$$

For the data, a noise level of $\sigma_{\text{nu}} = 0.01$ and $\sigma_{\text{nf}} = 0.3$ was introduced, effectively creating noise-free data. The result was computed over the time interval $t \in [0, 3]$, from which the training points were sampled using a sobol sequence.

plots/transformer/prediction_transformer.png

Figure 2: Calculated posteriors for test samples $\boldsymbol{X}_{\text{test}}$. With mean function and 95 % confidence interval.

4.2.1 General feed forward BNN

At the start, we look at a simple feed forward neural network with two hidden layers and 20 neurons per layer. For training we use 20 training points. There also is the possibility to directly render a flowchart of the model, which describes the properties of the defined function or model: The predictions are very accurate, however the uncertainty is predicted to low for some reason.

4.2.2 B-PINN

For this example, also the data from the forcing term f is used. Furthermore, the physical parameters $m = 1$, $\gamma = 2$, and $k = 2$ are introduced as probability distributions. They are assigned a prior and are then learned in the inference process. Again we show the properties of the model with the flowchart **??**. Again a network with 2 hidden layers with 20 neurons is used, as well as the same 20 training points. The solution term u and the forcing term f can both, after the training stage, be predicted. The plots for the B-PINN can be seen in **??**. We can see that the predictions for both u and f are very accurate, even with the data having a small noise. In comparison to the general model, the accuracy increased by a small margin of 0.01, which is not that big. However, when checking different amounts of training samples, we saw that the B-PINN performed better for the cases we looked at. Since we formulated the problem in an inverse way, meaning that we do not have knowledge of the parameters of the differential equation, it is also interesting to look at the posterior distributions of the parameters m, γ and k. The model was training with n_{train}=60 and n_{train}=150 and the distributions of the physical parameters plotted in a histogram, together with mean, meadian and percentiles. The plots can be seen in figure **??**.

plots/bnn_oscilator1_pinn.pdf

Figure 3: Graph of the general feed forward network

plots/BNN/oscilator1_60_[20, 20, 1].png

Figure 4: Prediction of the general network with n_{train}=60. The mean had an relL^2 of $L_u^2 = 0.051$ and $L_f^2 = 0.110$.

5 Source Code

```python
##############################################
# Author: Tobias Leitgeb
# date: 09.2024
# Description: Functions for feedforward neural networks
##############################################

import jax
import jax.numpy as jnp
from typing import List, Callable
import numpyro
import numpyro.distributions as dist

def dense_layer(
        i: int,
        size: List[int],
):
    #Xavier initialization
    alpha_sq = 2/(size[0] + size[1])
    #alpha_sq = 1.0
    W = numpyro.sample(f"W{i}", dist.Normal(0, alpha_sq**0.5).expand(size))
    b = numpyro.sample(f"b{i}", dist.Normal(0, alpha_sq**0.5).expand((size[-1],)))
    return W, b

def forward(
        W: List[jax.Array],
        b: List[jax.Array],
        X: jax.Array,
        activation: Callable,
):
    #input layer
    z = activation(jnp.dot(X, W[0]) + b[0])

    #hidden layers
```

`plots/BPINN/bnn_oscilator1_pinn.pdf`

Figure 5: Graph of the physics informed BPINN model.

plots/BPINN/oscilator1_60_[20, 20, 1].png

Figure 6: Prediction of the B-PINN with n_{train}= 60 for both u and f. The predicted mean had an relL^2 of $L_u^2 = 0.051$ and $L_f^2 = 0.110$.

plots/BPINN/oscilator1_60_[20, 20, 1]_combined_distributions.png

(a) Probability densities for the physical parameters m, γ, and k with n_{train}=60

plots/BPINN/oscilator1_150_[20, 20, 1]_combined_distr

(b) Probability densities for the physical parameters m, γ, and k with n_{train}=150

Figure 7: Posterior distributions of the parameters of the differential equation.

```python
    for i in range(1, len(W) - 1):
        z = activation(jnp.dot(z, W[i]) + b[i])

    #output layer with no activation
    z = jnp.dot(z, W[-1]) + b[-1]
    return z.squeeze()

def relative_l2_error(
        y_true: jax.Array,
        y_pred: jax.Array,
):
    return jnp.linalg.norm(y_true - y_pred) / jnp.linalg.norm(y_true)
```

```python
#############################################
# Author: Tobias Leitgeb
# date: 09.2024
# Description: Bayesian Neural Network with numpyro
#############################################
import numpyro
#numpyro.set_platform("cpu")
import jax
import jax.numpy as jnp
import jax.random as jr
import numpyro.distributions as dist

from numpyro.infer import Predictive, NUTS, MCMC
from numpyro.optim import Adam

import matplotlib.pyplot as plt
import seaborn as sns
from functools import partial

from utils import sample_training_points_space_filling
from network_functions import dense_layer, forward, relative_l2_error

from typing import Tuple, List, Callable
import sys
sys.path.append("../")

sns.set_theme("paper", font_scale=1.5)

NOISE_LEVELS = (0.01, 0.3)

def bnn_model(
        X: jax.Array,
        Y: jax.Array,
        layers: List[int],
):
    N, input_dim = X.shape
    activation = jnp.tanh
    W = []
    b = []
    #build the layers with the given list
    for i, layer in enumerate(layers):
        W_, b_ = dense_layer(i, [input_dim, layer])
        W.append(W_)
        b.append(b_)
        input_dim = layer
    #forward pass through the network
    z = forward(W, b, X, activation)[:, None]

    if Y is not None:
        assert Y.shape == z.shape , f"Y shape {Y.shape} does not match z shape {z.shape}"

    sigma_obs = NOISE_LEVELS[0]*2
    with numpyro.plate("data", N):
        numpyro.sample(
            "Y",
            dist.Normal(z, sigma_obs).to_event(1),
            obs=Y
        )
```

```python
def main(
        layers, train, data_size, num_warmup, num_samples
):

    data = jnp.load('data/oscilator2_data.npy', allow_pickle=True).item()
    X, Y, Y_f = data['X'], data['Y'], data['Y_f']

    # Sample training points
    X_train, Y_train, _, _, X_test, y_test, _ = sample_training_points_space_filling(
                                                        X, Y, Y_f, data_size, seed=0, noise_levels=
                                                            NOISE_LEVELS
                                                    )
    #normalize the training data
    mean_y = jnp.mean(Y_train)
    std_y = jnp.std(Y_train)
    Y_train = (Y_train - mean_y) / std_y

    try:
        render = numpyro.render_model(bnn_model, (X_train, Y_train, layers))
        render.render("plots/bnn_oscilator1_pinn")
    except:
        render = None
        print("Module not installed. (pip install graphviz), (sudo apt-get install graphviz)")



    key = jr.PRNGKey(0)
    key, inf_key, pred_key = jr.split(key, 3)

    name = f"BNN_{data_size}_{layers}"
    path = f"data/mcmc_samples/"+name+"_samples.npy"
    if train:
        mcmc = MCMC(NUTS(bnn_model), num_warmup=num_warmup, num_samples=num_samples)
        mcmc.run(inf_key, X_train, Y_train, layers)
        mcmc.print_summary()
        samples = mcmc.get_samples()
        jnp.save(path, samples, allow_pickle=True)

    else:
        try:
            samples = jnp.load(path, allow_pickle=True).item()
        except:
            print("No samples found. Train the model first.")
            return

    #predictive distribution
    predictive = Predictive(bnn_model, samples, return_sites=["Y"])
    predictions = predictive(pred_key, X_test, None, layers)


    label = f"BNN/oscilator1_{data_size}_{layers}"

    plt.figure(figsize=(8, 6))
    plt.plot(X_test, y_test, 'b--', label='True function')
    plt.plot(X_train, Y_train*std_y + mean_y, 'ro', label=r' training samples')
    # Plot the predictions
    mean_prediction = jnp.mean(predictions["Y"], axis=0)
    stddev_prediction = jnp.std(predictions["Y"], axis=0)
    plt.plot(X_test, mean_prediction*std_y + mean_y, 'g', label='Predictive mean')
    plt.fill_between(
        X_test.flatten(),
        (mean_prediction - 2 * stddev_prediction).flatten()*std_y + mean_y,
        (mean_prediction + 2 * stddev_prediction).flatten()*std_y + mean_y,
        color='g',
        alpha=0.4,
        label=r"2\\sigma uncertainty",
    )
    plt.legend()
    plt.xlabel("$x$")
    plt.ylabel("$u$")
    plt.savefig("plots/"+label+".png", dpi=300, bbox_inches='tight')
    print("Plots saved in plots/ directory")
    print("relative L2 error: ", relative_l2_error(y_test, mean_prediction*std_y + mean_y))

import argparse
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--train", action="store_true")
    parser.add_argument("--layers", type=int, default=[40, 40, 1],nargs="+", help="Number of neurons
         in each layer")
    parser.add_argument("--data_size", type=int, default=50, help="Number of training points")
    parser.add_argument("--num_samples", type=int, default=1000, help="Number of samples")
    parser.add_argument("--num_warmup", type=int, default=1000, help="Number of warmup steps")
    args = parser.parse_args()
    main(**vars(args))
```

```
import numpyro
#numpyro.set_platform("cpu")
import jax
import jax.numpy as jnp
import jax.random as jr
import numpyro.distributions as dist

from numpyro.infer import Predictive, NUTS, MCMC

import matplotlib.pyplot as plt
import seaborn as sns
from functools import partial
sns.set_theme("paper", font_scale=1.5)

from typing import Tuple, List, Callable
import sys
sys.path.append("../")
from utils import sample_training_points_space_filling
from network_functions import dense_layer, forward, relative_l2_error


NOISE_LEVELS = (0.01, 0.3)
# forward pass through the physics network
def physics_forward(
        W: List[jax.Array],
        b: List[jax.Array],
        X: jax.Array,
        activation: Callable,
):
    #input layer
    u = partial(forward, W, b, activation=activation)
    u_prime = jax.grad(u)
    u_double_prime = jax.grad(u_prime)

    #vectorize the functions
    u_prime = jax.vmap(u_prime)(X)
    u_double_prime = jax.vmap(lambda x: u_double_prime(x.squeeze()))(X)
    return u_prime, u_double_prime[:, None]

def BPINN(
        X: jax.Array,
        Y: jax.Array,
        layers: List[int],
):
    X_u, X_f = X

    if Y is not None:
        Y_u, Y_f = Y
    else:
        Y_u, Y_f = None, None

    N, input_dim = X_u.shape
    activation = jnp.tanh
    W = []
    b = []
    #build the layers with the given list
    for i, layer in enumerate(layers):
        W_, b_ = dense_layer(i, [input_dim, layer])
        W.append(W_)
        b.append(b_)
        input_dim = layer
    #forward pass through the network
    z = forward(W, b, X_u, activation)[:, None]

    dz, ddz = physics_forward(W, b, X_f, activation)
    assert dz.shape == z.shape, f"dz shape {dz.shape} does not match z shape {z.shape}"
    assert ddz.shape == z.shape, f"ddz shape {ddz.shape} does not match z shape {z.shape}"

    m = numpyro.sample(r"m", dist.Uniform(0, 2.))
    gamma = numpyro.sample(r"gamma", dist.Uniform(0, .8))
    k = numpyro.sample(r"k", dist.Uniform(1, 3.))

    if Y is not None:
        assert Y_u.shape == z.shape , f"Y shape {Y_u.shape} does not match z shape {z.shape}"

    sigma_obs_u = NOISE_LEVELS[0]
    sigma_obs_f = NOISE_LEVELS[1]

    #oscillator equation
    f = m * ddz + gamma * dz + k * z

    #joint likelihood
    numpyro.sample(
        r"Y_u",
        dist.Normal(z, sigma_obs_u).to_event(1),
        obs=Y_u
    )
    numpyro.sample(
        r"Y_f",
        dist.Normal(f, sigma_obs_f).to_event(1),
        obs=Y_f
    )


def main(
        layers, train, data_size, num_warmup, num_samples
):

    data = jnp.load('data/oscilator2_data.npy', allow_pickle=True).item()
    X, Y, Y_f = data['X'], data['Y'], data['Y_f']


# Sample training points
    X_u, u_train , X_f, f_train, X_test, u_test, f_test = sample_training_points_space_filling(
                                                                X, Y, Y_f, data_size, noise_levels=
                                                                    NOISE_LEVELS,seed=0,
                                                            )

    X_train = (X_u, X_f)
    Y_train = (u_train, f_train)

    try:
        render = numpyro.render_model(BPINN, (X_train, Y_train, layers))
        render.render("plots/BPINN/bnn_oscilator1_pinn")
    except:
        render = None
        print("Module not installed. (pip install graphviz), (sudo apt-get install graphviz)")

    inf_key = jax.random.PRNGKey(0)

    name = f"BPINN_{data_size}_{layers}"
    path = f"data/mcmc_samples/"+name+"_samples.npy"
    if train:
        mcmc = MCMC(NUTS(BPINN), num_warmup=num_warmup, num_samples=num_samples)
        mcmc.run(inf_key, X_train, Y_train, layers)
        mcmc.print_summary()
        samples = mcmc.get_samples()
        jnp.save(path, samples, allow_pickle=True)
    else:
        try:
            samples = jnp.load(path, allow_pickle=True).item()
        except:
            print("No samples found. Train the model first.")
            return

    #predictive
    predictive = Predictive(BPINN, samples, return_sites=["Y_u", "Y_f"])
    predictions = predictive(jax.random.PRNGKey(1), (X_test, X_test), None, layers)


    # Plot the predictions
    mean_prediction_u = jnp.mean(predictions["Y_u"], axis=0).ravel()
    stddev_prediction_u = jnp.std(predictions["Y_u"], axis=0).ravel()

    mean_prediction_f = jnp.mean(predictions["Y_f"], axis=0).ravel()
    stddev_prediction_f = jnp.std(predictions["Y_f"], axis=0).ravel()

    label = f"BPINN/oscilator1_{data_size}_{layers}"
    plot(
        X, u_test, X_u, u_train, X_test, mean_prediction_u, stddev_prediction_u,
        f_test, X_f, f_train, mean_prediction_f, stddev_prediction_f, label
    )
    #boxplot_physical_parameters(
    #    samples,
    #    ["m", "gamma", "k"],
    #    title=f"BPINN/params_oscilator1_{data_size}_{layers}"
    #)
    print("Plots saved in plots/ directory")
    print("relative L2 error relL2(u, u_hat): ", relative_l2_error(u_test.ravel(), mean_prediction_u
        ))
    print("relative L2 error relL2(f, f_hat): ", relative_l2_error(f_test.ravel(), mean_prediction_f
        ))

    #plot_distribution(samples, "m", label)
    #plot_distribution(samples, "gamma", label)
    #plot_distribution(samples, "k", label)

    plot_distributions(samples, ["m", "gamma", "k"], label)
def plot(
        X,
        u_test,
        X_u,
        u_train,
        X_test,
        mean_prediction_u,
        stddev_prediction_u,
        f_test,
        X_f,
        f_train,
        mean_prediction_f,
        stddev_prediction_f,
        label,
):
    fig, ax = plt.subplots(2, 1, figsize=(10, 10))
    ax[0].plot(X, u_test, 'b--', label='True function')
    ax[0].plot(X_u, u_train, 'ro', label=r'$u$ training samples')
    ax[0].plot(X_test, mean_prediction_u, 'g', label='u prediction')
    ax[0].fill_between(
        X_test.ravel(),
        mean_prediction_u - 2 * stddev_prediction_u,
        mean_prediction_u + 2 * stddev_prediction_u,
        color='g', alpha=0.4, label=r"$2\sigma$ uncertainty"
    )
    ax[0].legend()
    ax[0].set_ylabel('u')
    #ax[0].set_xlabel('x')
    ax[0].set_title(r'$u$ prediction')

    ax[1].plot(X, f_test, 'b--', label='true function')
    ax[1].plot(X_f, f_train, 'ro', label=r'$f$ training samples')
    ax[1].plot(X_test, mean_prediction_f, 'g', label='Y_f prediction')
    ax[1].fill_between(
        X_test.ravel(),
        mean_prediction_f - 2 * stddev_prediction_f,
        mean_prediction_f + 2 * stddev_prediction_f,
        color='g', alpha=0.4, label=r"$2\sigma$ uncertainty"
    )
    ax[1].set_ylabel('f')
    ax[1].set_xlabel('x')
    ax[1].legend()
    ax[1].set_title(r'$f$ prediction')
    plt.savefig("plots/"+label+".png", dpi=300, bbox_inches='tight')

def boxplot_physical_parameters(
        samples,
        labels,
        title,
):
    fig, ax = plt.subplots(1, 3, figsize=(15, 5))
    labels_plot = ["$m$", r"$\gamma$", "$k$"]
    for i, label in enumerate(labels):
        ax[i].boxplot(samples[label], vert=False)
        ax[i].set_title(labels_plot[i])
    #fig.suptitle(title)
    plt.savefig("plots/"+title+".png", dpi=300, bbox_inches='tight')


def plot_distribution(samples, param_name, label):
    data = samples[param_name]

    # Calculate summary statistics
    mean = jnp.mean(data)
    std = jnp.std(data)
    median = jnp.median(data)
    perc_5 = jnp.percentile(data, 5)
    perc_95 = jnp.percentile(data, 95)

    # Plot the distribution
    plt.figure(figsize=(10, 6))
    sns.histplot(data, kde=True, bins=30, color='blue', stat='density')

    # Annotate the plot with summary statistics
    plt.axvline(mean, color='r', linestyle='--', label=f'Mean: {mean:.2f}')
    plt.axvline(median, color='g', linestyle='-', label=f'Median: {median:.2f}')
    plt.axvline(perc_5, color='b', linestyle='--', label=f'5th Percentile: {perc_5:.2f}')
    plt.axvline(perc_95, color='b', linestyle='--', label=f'95th Percentile: {perc_95:.2f}')

    plt.title(f'Distribution of {param_name}')
    plt.xlabel(param_name)
    plt.ylabel('Density')
    plt.legend()
    plt.savefig("plots/"+label+f"_{param_name}_distribution.png", dpi=300, bbox_inches='tight')

def plot_distributions(samples, param_names, label):
    num_params = len(param_names)
    fig, axes = plt.subplots(num_params, 1, figsize=(10, 6 * num_params))

    for i, param_name in enumerate(param_names):
        data = samples[param_name]

        # Calculate summary statistics
        mean = jnp.mean(data)
        std = jnp.std(data)
        median = jnp.median(data)
        perc_5 = jnp.percentile(data, 5)
        perc_95 = jnp.percentile(data, 95)

        # Plot the distribution
        sns.histplot(data, kde=True, bins=30, stat='density', ax=axes[i], color='blue')

        # Annotate the plot with summary statistics
        axes[i].axvline(mean, color='r', linestyle='--', label=f'Mean: {mean:.2f}')
        axes[i].axvline(median, color='g', linestyle='-', label=f'Median: {median:.2f}')
        axes[i].axvline(perc_5, color='b', linestyle='--', label=f'5th Percentile: {perc_5:.2f}')
        axes[i].axvline(perc_95, color='b', linestyle='--', label=f'95th Percentile: {perc_95:.2f}')

        axes[i].set_title(f'Distribution of {param_name}')
        axes[i].set_xlabel(param_name)
        axes[i].set_ylabel('Density')
        axes[i].legend()

    plt.tight_layout()
    plt.savefig(f"plots/{label}_combined_distribution.png", dpi=300)
    plt.show()


import argparse
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--train", action="store_true")
    parser.add_argument("--layers", type=int,nargs="+", default=[40, 40, 1], help="Number of neurons
         in each layer")
    parser.add_argument("--data_size", type=int, default=50, help="Number of training points")
    parser.add_argument("--num_samples", type=int, default=1000, help="Number of samples")
    parser.add_argument("--num_warmup", type=int, default=1000, help="Number of warmup steps")
    args = parser.parse_args()
    main(**vars(args))
```

```
#############################################
# Author: Tobias Leitgeb
# date: 09.2024
# Description: Bayesian Neural Network with numpyro
#############################################

import jax
from typing import Tuple
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
import jax.random as jr

def get_data(
        X: jax.Array,
        Y: jax.Array,
        f: jax.Array,
        *,
        split: Tuple[float, float, float] = (0.8, 0.1),
        dense: bool = False,
        key: jax.random.PRNGKey,
):
    #split data into training testing and validation
    n = X.shape[0]
    n_train = int(n * split[0])
    n_test = int(n * split[1])


    #shuffle data
    key, subkey = jax.random.split(key)
    idx = jax.random.permutation(subkey, n)
    X = X[idx, :]
    Y = Y[idx, :]

    #split data
    X_train, Y_train = jnp.array(X[:n_train, :]), jnp.array(Y[:n_train, :])
    X_test, Y_test = jnp.array(X[n_train:n_train + n_test, :]), jnp.array(Y[n_train:n_train + n_test
        , :])
    X_val, Y_val = jnp.array(X[n_train + n_test:, :]), jnp.array(Y[n_train + n_test:, :])

    if dense:
        X_train = jnp.repeat(X_train, len(f), axis=0)
        X_train = jnp.concatenate([X_train, jnp.tile(f, len(X_train)//len(f))[:,None]], axis=1)
        X_test = jnp.repeat(X_test, len(f), axis=0)
        X_test = jnp.concatenate([X_test, jnp.tile(f, len(X_test)//len(f))[:,None]], axis=1)
        X_val = jnp.repeat(X_val, len(f), axis=0)
        X_val = jnp.concatenate([X_val, jnp.tile(f, len(X_val)//len(f))[:,None]], axis=1)
        return (X_train, Y_train), (X_test, Y_test), (X_val, Y_val)
    else:
        return (X_train, Y_train), (X_test, Y_test), (X_val, Y_val)

def plot_testset(
            frequency: jax.Array,
            test_y: jax.Array,
            prediction: jax.Array,
            var_pred: jax.Array=None,
            grid: tuple = (4,4),
            figsize: tuple = (15, 15)
        )-> plt.figure:
        """
        Plots the test set predictions and true values.

        Args:
            frequency (ndarray): The frequency values.
            test_y (ndarray): The true output values of shape (n_test, p).
            prediction (ndarray): The predicted output values of shape (n_test, p).
            grid (tuple): The grid size for subplots.
        """
        fig, ax  = plt.subplots(grid[0], grid[1], figsize=figsize)

        if grid[0] == 1 and grid[1] == 1:
            ax = [ax]

        ax = ax.flatten()

        for i in range(grid[0]*grid[1]):
            ax[i].plot(frequency, test_y[i, :], color='black')
            ax[i].plot(frequency, prediction[i, :], color='blue')
            if var_pred is not None:
                pred_std = np.sqrt(var_pred[i, :])
                ax[i].fill_between(
                    frequency,
                    prediction[i, :] - 2*pred_std,
                    prediction[i, :] + 2*pred_std,
                    color='blue', alpha=0.2
                )

            ax[i].set_ylabel('dB')
            ax[i].legend(['True', 'Predicted'])
            #ax[i].grid()
            ax[i].set_xscale('log')

            error = minmaxrmspe(test_y[i, :][None,:], prediction[i, :][None,:])
            ax[i].set_title('minmaxRMSE: {:.2f}%'.format(error))
            #add axis only for the last row
            if i >= grid[0]*(grid[1]-1):
                ax[i].set_xlabel('Frequency (Hz)')
            plt.tight_layout()
        return ax, fig

def minmaxrmspe(
            test_y: jax.Array,
            prediction: jax.Array
        )-> float:
        """
        Calculates the Root Mean Square Percentage Error.

        Args:
            test_y (ndarray): The true output values of shape (n, p).
            prediction (ndarray): The predicted output values of shape (n, p).

        Returns:
            float: The Root Mean Square Percentage Error.
        """
        p = test_y.shape[1]
        error = jnp.mean(
            jnp.sqrt(1 / p * jnp.sum((test_y - prediction) ** 2, axis=1))
            * 100 / (jnp.max(test_y, axis=1) - jnp.min(test_y, axis=1))
        )
        return error

def plot_samples(
          f,
          Y_test,
          predicted_mean,
          samples,
):
    fig, ax = plt.subplots()
    ax.set_ylabel('dB')
    ax.set_xscale('log')
    ax.plot(f, samples, alpha = 0.6)
    ax.plot(f, Y_test, "blue", label="Test sample")
    ax.plot(f, predicted_mean, "r--", label="mean")

    ax.legend()


from scipy.stats.qmc import Sobol

def sample_training_points_space_filling(
        X,
        Y_1,
        Y_2,
        num_samples,
        noise_levels: Tuple[float, float] = (1e-2, 1e-2),
        seed=0
):
    """
    Sample training points from X, Y_1, and Y_2 using Sobol sequences.
    Generates different X positions for Y_1 and Y_2.
    """
    num_total_samples = X.shape[0]

    # Initialize Sobol sequence samplers for Y_1 and Y_2
    sobol_1 = Sobol(d=1, scramble=True, seed=seed)
    sobol_2 = Sobol(d=1, scramble=True, seed=seed + 1)  # Different seed for different sequence

    sobol_samples_1 = sobol_1.random(num_samples)
    sobol_samples_2 = sobol_2.random(num_samples)

    indices_1 = (sobol_samples_1 * num_total_samples).astype(int).flatten()
    indices_2 = (sobol_samples_2 * num_total_samples).astype(int).flatten()

    indices_1 = jnp.clip(indices_1, 0, num_total_samples - 1)
    indices_2 = jnp.clip(indices_2, 0, num_total_samples - 1)

    # Sample from the arrays using the Sobol sequence indices
    X_samples_1 = X[indices_1]
    Y_1_samples = Y_1[indices_1] + jr.normal(jr.PRNGKey(seed), shape=(num_samples, )) * noise_levels
        [0]
    X_samples_2 = X[indices_2]
    Y_2_samples = Y_2[indices_2]+ jr.normal(jr.PRNGKey(seed), shape=(num_samples, )) * noise_levels[
        1]

    # test set without the training points



    return X_samples_1[:,None], Y_1_samples[:,None], X_samples_2[:,None], Y_2_samples[:,None], X[:,
        None], Y_1[:,None], Y_2[:,None]
```