HW_07-PartB_RohanDekate

December 10, 2023

1 Homework 7 - Part B

Note that there are two different notebooks for HW assignment 7. This is part A. There will be two different assignments in gradescope for each part. The deadlines are the same for both parts.

1.1 References

- Lectures 27-28 (inclusive).

1.2 Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
import seaborn as sns
sns.set_context("paper")
sns.set_style("ticks")

import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
```

```
    url             -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)
```

1.3 Student details

- **First Name:** Rohan
- **Last Name:** Dekate
- **Email:** dekate@purdue.edu

[ ]:
```
# Run this on Google colab
!pip install pyro-ppl
```

```
Requirement already satisfied: pyro-ppl in /usr/local/lib/python3.10/dist-
packages (1.8.6)
Requirement already satisfied: numpy>=1.7 in /usr/local/lib/python3.10/dist-
packages (from pyro-ppl) (1.23.5)
Requirement already satisfied: opt-einsum>=2.3.2 in
/usr/local/lib/python3.10/dist-packages (from pyro-ppl) (3.3.0)
Requirement already satisfied: pyro-api>=0.1.1 in
/usr/local/lib/python3.10/dist-packages (from pyro-ppl) (0.1.2)
Requirement already satisfied: torch>=1.11.0 in /usr/local/lib/python3.10/dist-
packages (from pyro-ppl) (2.1.0+cu118)
Requirement already satisfied: tqdm>=4.36 in /usr/local/lib/python3.10/dist-
packages (from pyro-ppl) (4.66.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from torch>=1.11.0->pyro-ppl) (3.13.1)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from torch>=1.11.0->pyro-ppl) (4.5.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch>=1.11.0->pyro-ppl) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch>=1.11.0->pyro-ppl) (3.2.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch>=1.11.0->pyro-ppl) (3.1.2)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch>=1.11.0->pyro-ppl) (2023.6.0)
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-
packages (from torch>=1.11.0->pyro-ppl) (2.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.11.0->pyro-ppl)
(2.1.3)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-
packages (from sympy->torch>=1.11.0->pyro-ppl) (1.3.0)
```

[ ]:
```
import pyro
import pyro.distributions as dist
from pyro.infer import MCMC, NUTS
import torch
```

1.4 Problem 1 - Bayesian Linear regression on steroids

The purpose of this problem is to demonstrate that we have learned enough to do very complicated things. In the first part, we will do Bayesian linear regression with radial basis functions (RBFs) in which we characterize the posterior of all parameters, including the length-scales of the RBFs. In the second part, we are going to build a model that has an input-varying noise. Such models are called heteroscedastic models.

We need to write some `pytorch` code to compute the design matrix. This is absolutely necessary so that `pyro` can differentiate through all expressions.

[ ]:
```
class RadialBasisFunctions(torch.nn.Module):
    """Radial basis functions basis.

    Arguments:
    X   -  The centers of the radial basis functions.
    ell -  The assumed length scale.
    """
    def __init__(self, X, ell):
        super().__init__()
        self.X = X
        self.ell = ell
        self.num_basis = X.shape[0]
    def forward(self, x):
        distances = torch.cdist(x, self.X)
        return torch.exp(-.5 * distances ** 2 / self.ell ** 2)
```

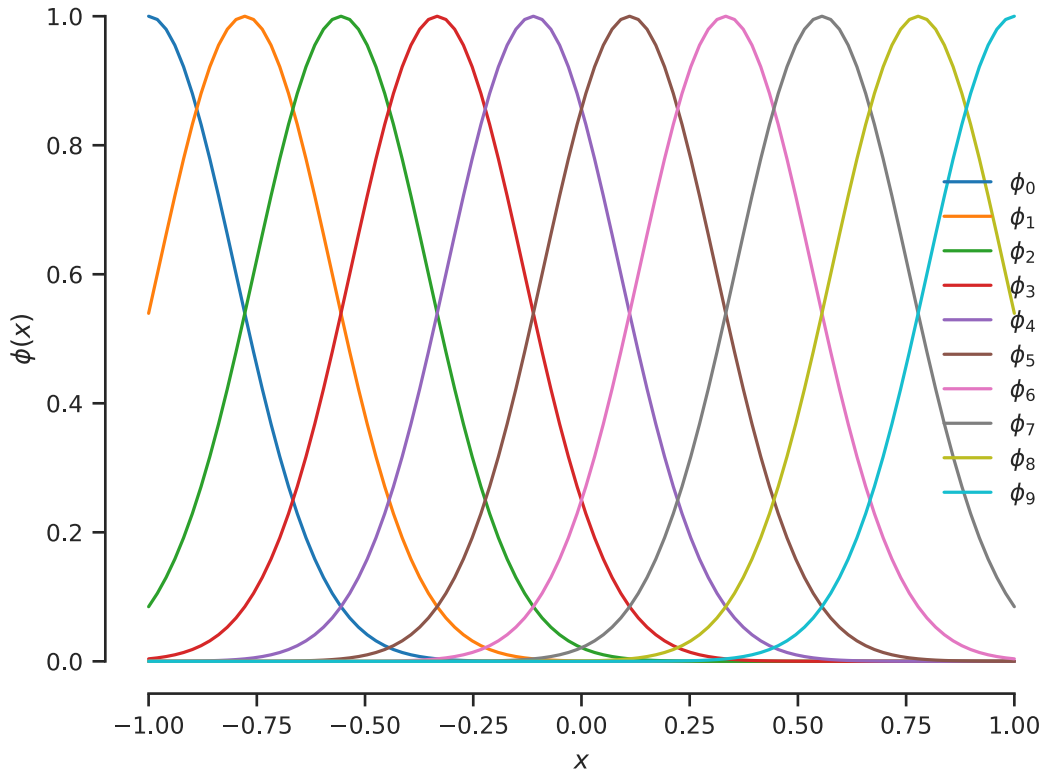
Here is how you can use them:

[ ]:
```
# Make the basis
x_centers = torch.linspace(-1, 1, 10).unsqueeze(-1)
ell = 0.2
basis = RadialBasisFunctions(x_centers, ell)

# Some points (need to be N x 1)
x = torch.linspace(-1, 1, 100).unsqueeze(-1)

# Evaluate the basis
Phi = basis(x)

# Here is the shape of Phi
print(Phi.shape)
```

```
torch.Size([100, 10])
```

Here is how they look like:

[ ]:
```
fig, ax = plt.subplots()
for i in range(Phi.shape[1]):
    ax.plot(x, Phi[:, i], label=f"$\phi_{i}$")
ax.set(xlabel="$x$", ylabel="$\phi(x)$")
ax.legend(loc="best", frameon=False)
sns.despine(trim=True);
```



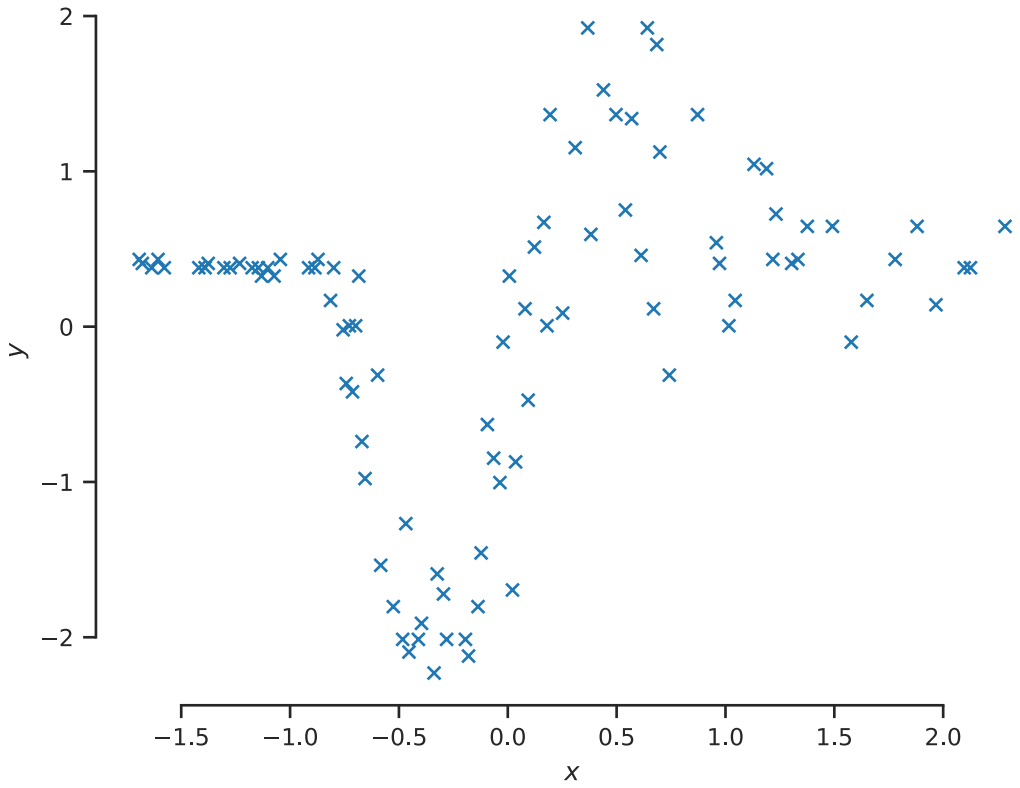
1.4.1 Part A - Hierarchical Bayesian linear regression with input-independent noise

We will analyze the motorcycle dataset. The data is loaded below.

[ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
  ↪lecturebook/data/motor.dat"
download(url)
```

We will work with the scaled data:

[ ]:
```
from sklearn.preprocessing import StandardScaler

data = np.loadtxt('motor.dat')
scaler = StandardScaler()
data = scaler.fit_transform(data)
X = torch.tensor(data[:, 0], dtype=torch.float32).unsqueeze(-1)
Y = torch.tensor(data[:, 1], dtype=torch.float32)

fig, ax = plt.subplots()
ax.plot(X, Y, 'x')
ax.set(xlabel="$x$", ylabel="$y$")
sns.despine(trim=True);
```



1.4.2 Part A.I

Your goal is to implement the model described below. We use the radial basis functions (`RadialBasisFunction`) with centers, x_i at $m = 50$ equidistant points between the minimum and maximum of the observed inputs:

$$\phi_i(x; \ell) = \exp\left(-\frac{(x - x_i)^2}{2\ell^2}\right),$$

for $i = 1, \ldots, m$. We denote the vector of RBFs evaluated at x as $\phi(x; \ell)$.

We are not going to pick the length-scales ℓ by hand. Instead, we will put a prior on it:

$$\ell \sim \text{Exponential}(1).$$

The corresponding weights have priors:

$$w_j | \alpha_i \sim N(0, \alpha_j^2),$$

and its α_j has a prior:

$$\alpha_j \sim \text{Exponential}(1),$$

for $j = 1, \ldots, m$.

Denote our data as:

$$x_{1:n} = (x_1, \ldots, x_n)^T, \text{ (inputs)},$$

and

$$y_{1:n} = (y_1, \ldots, y_n)^T, \text{ (outputs)}.$$

The likelihood of the data is:

$$y_i | \mathbf{w}, \sigma \sim N(\mathbf{w}^T \phi(x_i; \ell), \sigma^2),$$

for $i = 1, \ldots, n$.

$$y_n | \ell, \mathbf{w}, \sigma \sim N(\mathbf{w}^T \phi(x_n; \ell), \sigma^2).$$

Complete the `pyro` implementation of that model:

Answer:

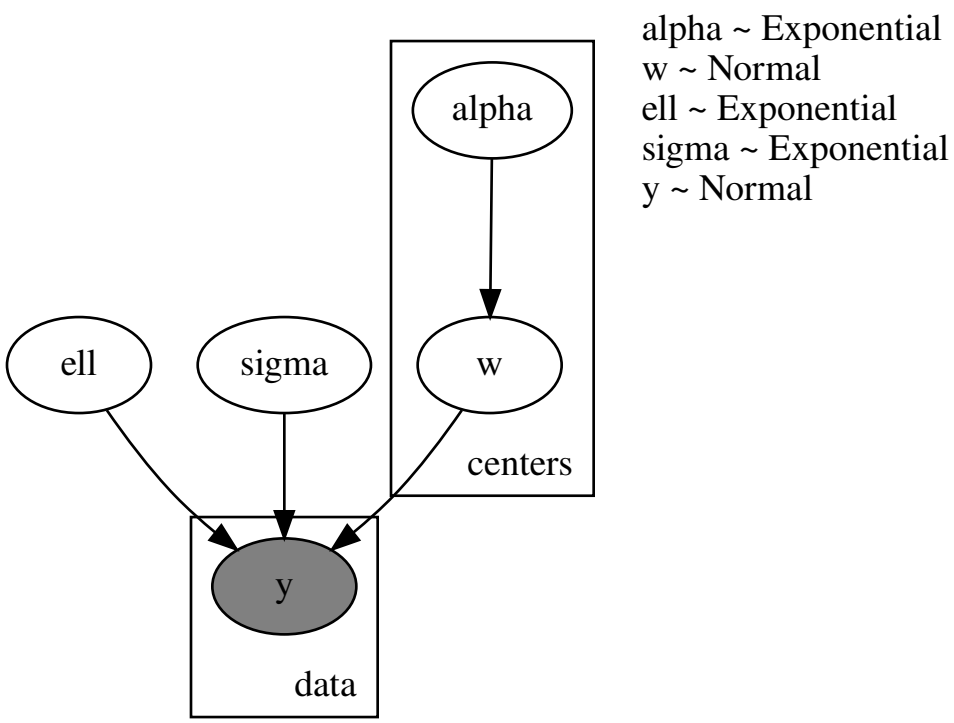
[ ]:
```
def model(X, y, num_centers=50):
    with pyro.plate("centers", num_centers):
        alpha = pyro.sample("alpha", dist.Exponential(1.0))
        # Notice below that dist.Normal needs the standard deviation - not the␣
↪variance
        # We follow a different convention in the lecture notes
        w = pyro.sample("w", dist.Normal(0.0, alpha))
    ell = pyro.sample("ell", dist.Exponential(1))# Complete the code assign to␣
↪ell the correct prior distribution (an Exponential(1)).
    # Hint: Look at alpha.
```

```
    sigma = pyro.sample("sigma", dist.Exponential(1))# Complete the code assign␣
↪to sigma the correct prior distribution (an Exponential(1))
    x_centers = torch.linspace(X.min(), X.max(), num_centers).unsqueeze(-1)
    Phi = RadialBasisFunctions(x_centers, ell)(X)
    with pyro.plate("data", X.shape[0]):
        pyro.sample("y", dist.Normal(Phi @ w, sigma), obs=y)
    # Notice that I'm making the model return all the variables that I have␣
↪made.
    # This is not essential for characterizing the posterior, but it does␣
↪reduce redundant code
    # when we are trying to get the posterior predictive.
    return locals()
```

The graph will help to understand the model:

[ ]:
```
pyro.render_model(model, (X, Y), render_distributions=True)
```

[ ]:



Use `pyro.infer.autoguide.AutoDiagonalNormal` to make the guide:

[ ]:
```
guide = pyro.infer.autoguide.AutoDiagonalNormal(model)
```

We will use variational inference. Here is the training code from the hans-on activity:

[ ]:
```python
def train(model, guide, data, num_iter=5_000):
    """Train a model with a guide.

    Arguments
    ---------
    model    -- The model to train.
    guide    -- The guide to train.
    data     -- The data to train the model with.
    num_iter -- The number of iterations to train.

    Returns
    -------
    elbos -- The ELBOs for each iteration.
    param_store -- The parameters of the model.
    """

    pyro.clear_param_store()

    optimizer = pyro.optim.Adam({"lr": 0.001})

    svi = pyro.infer.SVI(
        model,
        guide,
        optimizer,
        loss=pyro.infer.JitTrace_ELBO()
    )

    elbos = []
    for i in range(num_iter):
        loss = svi.step(*data)
        elbos.append(-loss)
        if i % 1_000 == 0:
            print(f"Iteration: {i} Loss: {loss}")

    return elbos, pyro.get_param_store()
```

1.4.3 Part A.II

Train the model for 20,000 iterations. Call the `train()` function we defined above to do it. Make sure you store the returned elbo values because you will need them later.

Answer:

[ ]:
```python
# Your code here
elbo_a2, param_a2 = train(model,guide,(X,Y),num_iter=20000)
```

```
Iteration: 0 Loss: 344.4841003417969
Iteration: 1000 Loss: 177.7049560546875
```

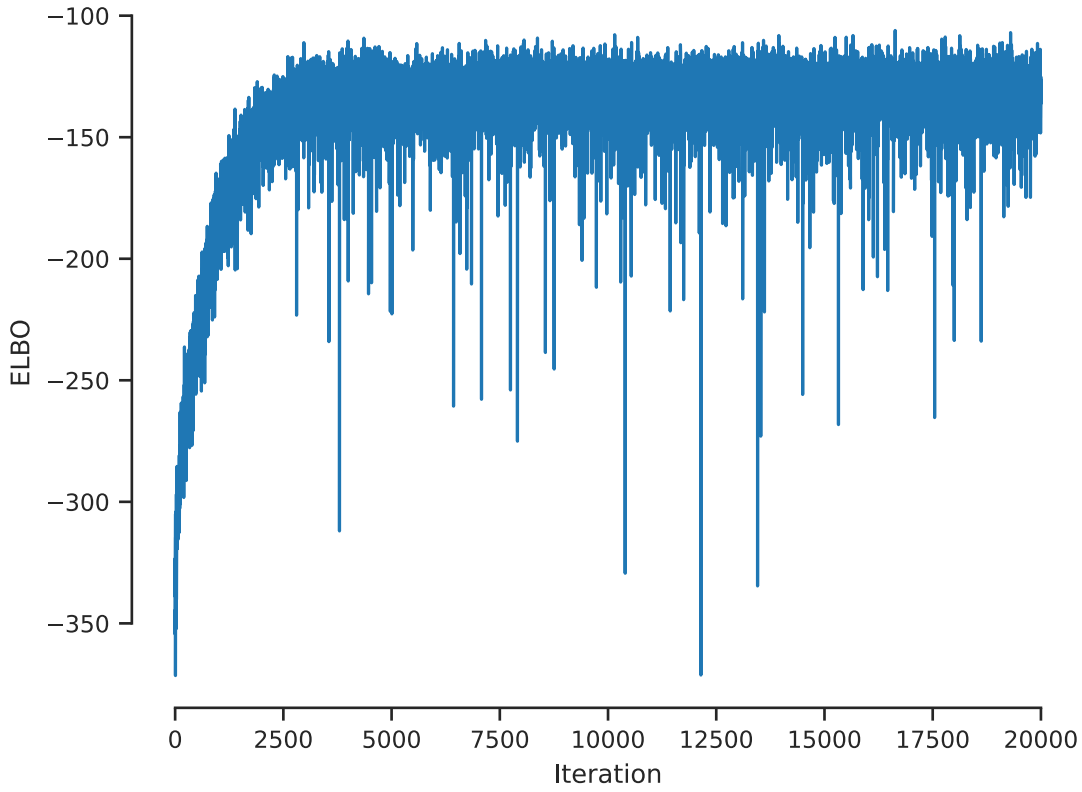
```
Iteration: 2000 Loss: 143.35397338867188
Iteration: 3000 Loss: 140.1400604248047
Iteration: 4000 Loss: 133.86328125
Iteration: 5000 Loss: 130.1626739501953
Iteration: 6000 Loss: 126.19853210449219
Iteration: 7000 Loss: 138.15557861328125
Iteration: 8000 Loss: 120.63284301757812
Iteration: 9000 Loss: 134.07684326171875
Iteration: 10000 Loss: 141.12298583984375
Iteration: 11000 Loss: 131.5647430419922
Iteration: 12000 Loss: 129.57150268554688
Iteration: 13000 Loss: 127.53336334228516
Iteration: 14000 Loss: 116.71770477294922
Iteration: 15000 Loss: 130.14065551757812
Iteration: 16000 Loss: 131.07748413085938
Iteration: 17000 Loss: 125.29229736328125
Iteration: 18000 Loss: 133.89187622070312
Iteration: 19000 Loss: 142.22801208496094
```

1.4.4 Part A.III

Plot the evolution of the ELBO.

Answer:

[ ]:
```
# Your code here
fig, ax = plt.subplots()
ax.plot(elbo_a2)
ax.set(xlabel='Iteration', ylabel='ELBO')
sns.despine(trim=True);
```



1.4.5 Part A.IV

Take 1,000 posterior samples.

Answer:

I'm giving you this one because it is a bit tricky. You need to use the `pyro.infer.Predictive` class to do it. Here is how you can use it:

[ ]:
```
post_samples = pyro.infer.Predictive(model, guide=guide, num_samples=1000)(X, Y)
# Just modify the call to get the right number of samples
```

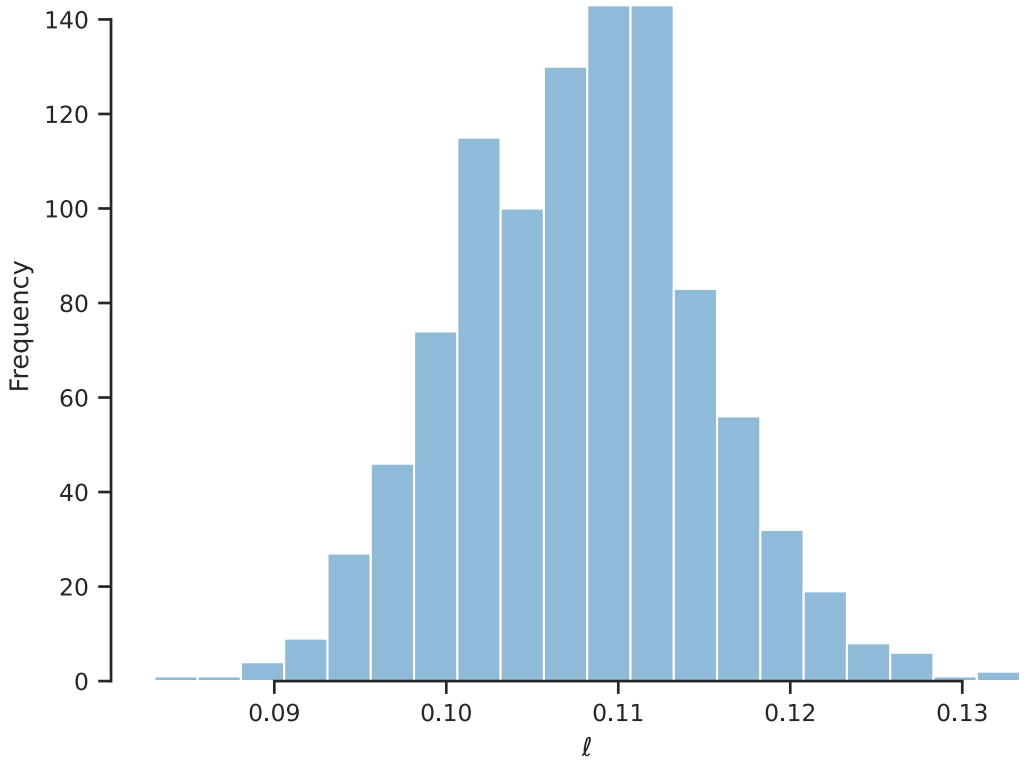
1.4.6 Part A.V

Plot the histograms of the posteriors of ℓ, σ, α_{10} and w_{10}.

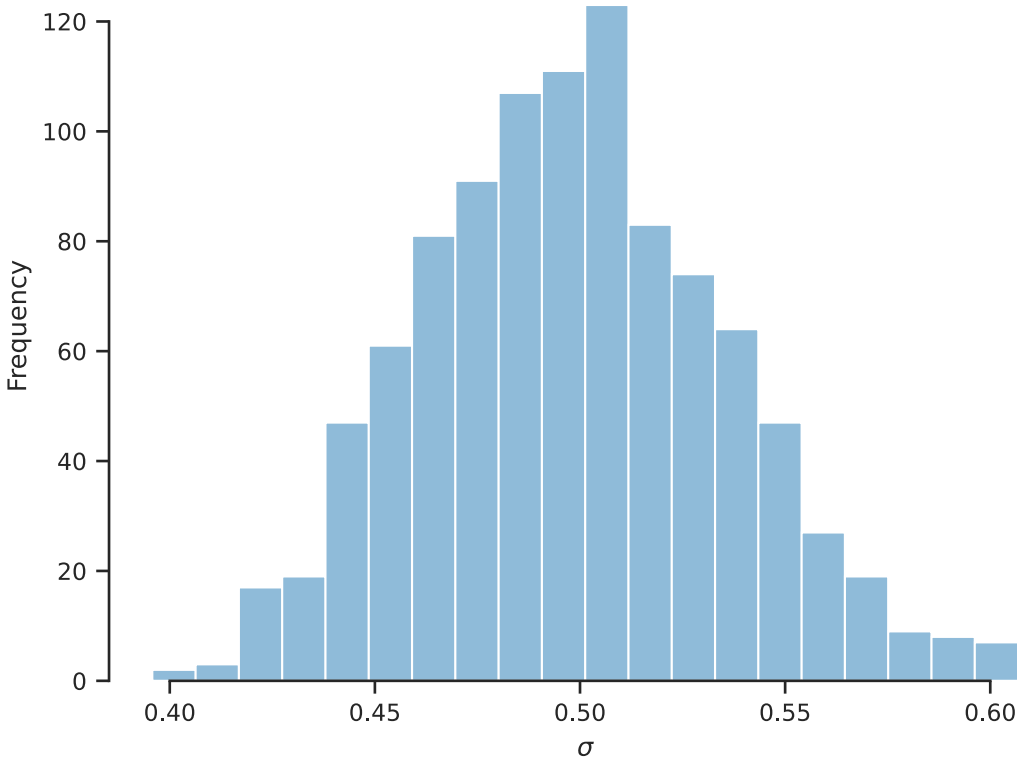
Answer:

[ ]:
```
# First, here is how to extract the samples.
ell = post_samples["ell"]
# You can do `post_samples.keys()` to see all the keys.
# But they should correspond to the names of the latent variables in the model.
sigma = post_samples["sigma"]# Your code here
```

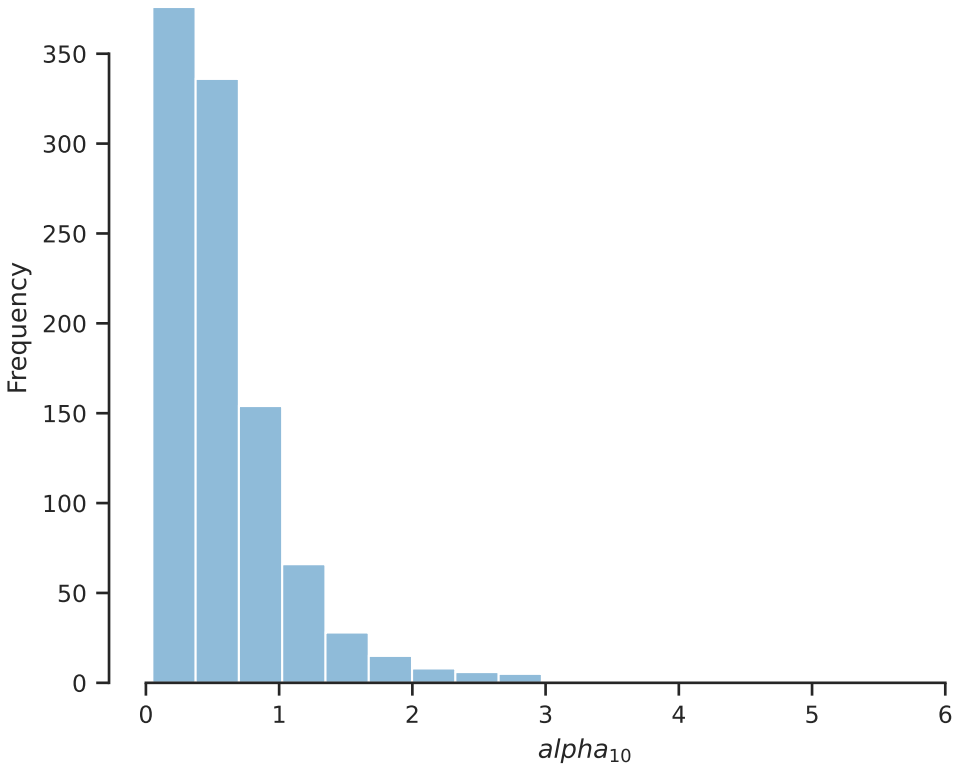
```
alphas = post_samples["alpha"][:,9]# Your code here
ws = post_samples["w"][:,9]# Your code here

# Here is the code to make the histogram for the length scale.
fig, ax = plt.subplots()
# **VERY IMPORTANT** - You need to detach the tensor from the computational␣
 ↪graph.
# Otherwise, you will get very very strange behavior.
ax.hist(ell.detach().numpy(), bins=20, alpha=.5)
ax.set(xlabel="$\ell$", ylabel="Frequency")
sns.despine(trim=True);

# Your code for the other histograms here
```



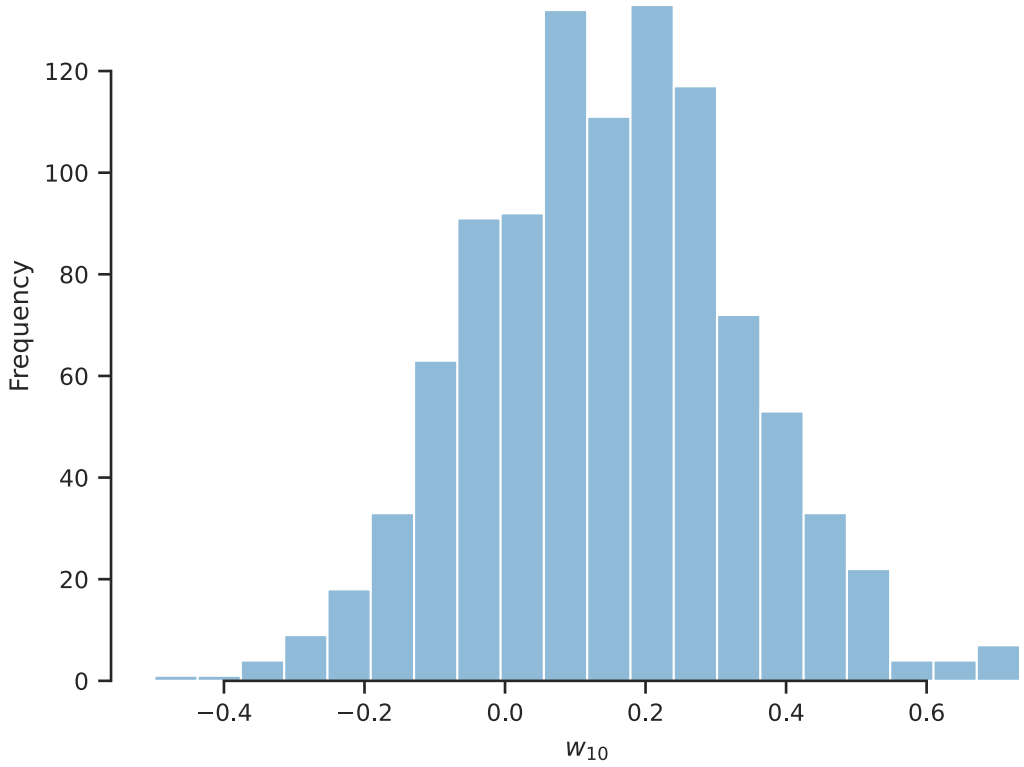
[ ]:
```
fig, ax = plt.subplots()
ax.hist(sigma.detach().numpy(), bins=20, alpha=.5)
ax.set(xlabel="$\sigma$", ylabel="Frequency")
sns.despine(trim=True);
```



[ ]:
```
fig, ax = plt.subplots()
ax.hist(alphas.detach().numpy(),bins=20, alpha=.5)
ax.set(xlabel="$alpha_{10}$", ylabel="Frequency")
sns.despine(trim=True);
```



[ ]:
```
fig, ax = plt.subplots()
ax.hist(ws.detach().numpy(),bins=20, alpha=.5)
ax.set(xlabel="$w_{10}$", ylabel="Frequency")
sns.despine(trim=True);
```



1.4.7 Part A.VI

Let's extend them model to make predictions.

Answer:

[ ]:
```
# Again, I'm giving you most of the code here.

def predictive_model(X, y, num_centers=50):
    # First we run the original model get all the variables
    params = model(X, y, num_centers)
    # Here is how you can access the variables
    w = params["w"]
    ell = params["ell"]# Access the length scale
    sigma = params["sigma"]# Access the standard deviation of the measurement␣
 ↪noise
    x_centers = params["x_centers"]# Access the centers of the radial basis␣
 ↪functions
    # Here are the points where we want to make predictions
    xs = torch.linspace(X.min(), X.max(), 100).unsqueeze(-1)
    # Evaluate the basis on the prediction points
    Phi = RadialBasisFunctions(x_centers, ell)(xs)
```

```
    # Make the predictions - we use a deterministic node here because we want to
    # save the results of the predictions.
    predictions = pyro.deterministic("predictions", Phi @ w)
    # Finally, we add the measurement noise
    predictions_with_noise = pyro.sample("predictions_with_noise", dist.
↪Normal(predictions, sigma))
    return locals()
```

1.4.8 Part A.VII

Extract the posterior predictive distribution using 10,000 samples. Separate aleatory and epistemic uncertainty.

Answer:

[ ]:
```
# Here is how to make the predictions. Just change the number of samples to the␣
↪right number.
post_pred = pyro.infer.Predictive(predictive_model, guide=guide,␣
↪num_samples=10000)(X, Y)
# We will predict here:
xs = torch.linspace(X.min(), X.max(), 100).unsqueeze(-1)
# You can extract the predictions from post_pred like this:
predictions = post_pred["predictions"]
# Note that we extracted the deterministic node called "predictions" from the␣
↪model.
# Get the epistemic uncertainty in the usual way:
p_500, p_025, p_975 = np.percentile(predictions, [50, 2.5, 97.5], axis=0)
# Extract predictions with noise
predictions_with_noise = post_pred["predictions_with_noise"]# Your code here
# Get the aleatory uncertainty
ap_025, ap_975 = np.percentile(predictions_with_noise, [2.5, 97.5], axis=0)#␣
↪Your code here
```

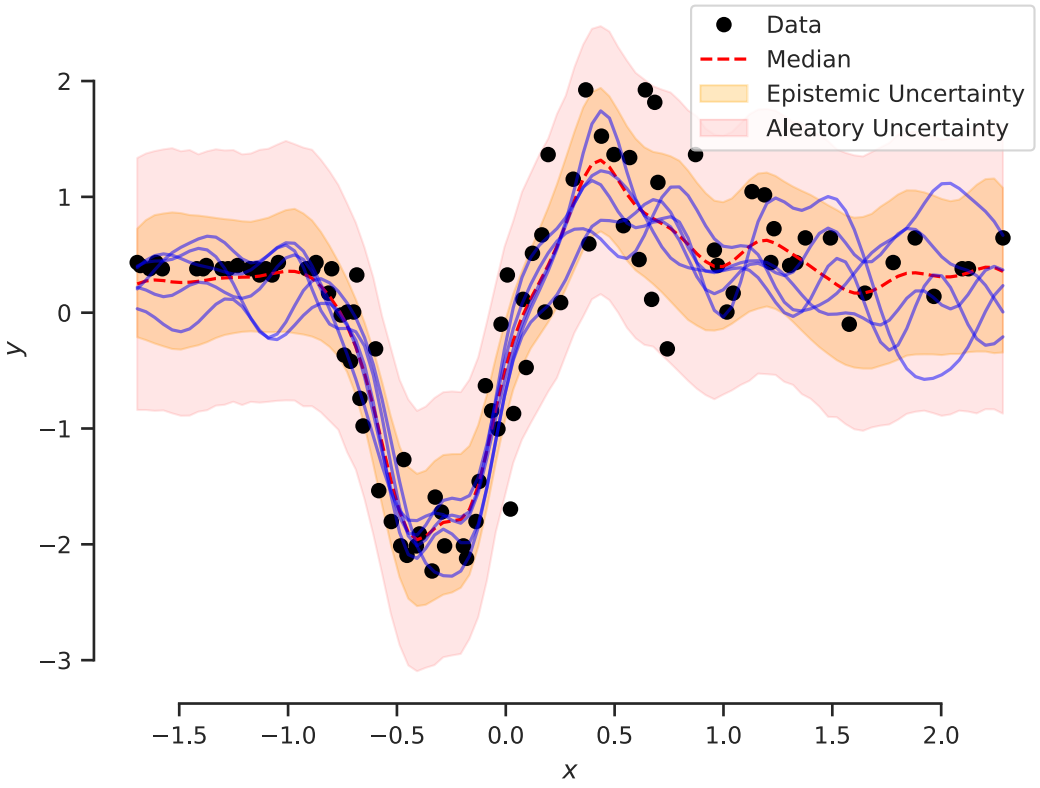
1.4.9 Part A.VIII

Plot the data, the median, the 95% credible interval of epistemic uncertainty and the 95% credible interval of aleatory uncertainty, along with five samples from the posterior.

Answer:

[ ]:
```
# Your code here. You have everything you need to make the plot.
fig, ax = plt.subplots()
ax.plot(X, Y, 'ko', label="Data")
ax.plot(xs.flatten(), p_500.flatten(), 'r--', label="Median")
ax.fill_between(xs.flatten(), p_025.flatten(), p_975.flatten(), color='orange',␣
↪alpha=0.25, label="Epistemic Uncertainty")
ax.fill_between(xs.flatten(), ap_025.flatten(), ap_975.flatten(), color='r',␣
↪alpha=0.1, label="Aleatory Uncertainty")
```

```
ax.set(xlabel="$x$", ylabel="$y$")
# Plot a few samples
for i in range(5):
    ax.plot(xs.flatten(), predictions[i].flatten(), 'b', alpha=0.5)
ax.legend()
sns.despine(trim=True);
```



1.4.10 Part B - Heteroscedastic regression

We are going to build a model that has an input-varying noise. Such models are called heteroscedastic models. Here I will let you do more of the work.

Everything is as before for ℓ, the α_j's, and the w_j's. We now introduce a model for the noise that is input dependent. It will use the same RBFs as the mean function. But let's use a different length-scale, ℓ_σ. So, we add:

$$\ell_\sigma \sim \text{Exponential}(1),$$

$$\alpha_{\sigma,j} \sim \text{Exponential}(1),$$

and

$$w_{\sigma,j}|\alpha_{\sigma,j} \sim N(0, \alpha_{\sigma,j}^2),$$

for $j = 1, \ldots, m$.

Our model for the input-dependent noise variance is:

$$\sigma(x; \mathbf{w}_\sigma, \ell) = \exp\left(\mathbf{w}_\sigma^T \phi(x; \ell_\sigma)\right).$$

So, the likelihood of the data is:

$$y_i|\mathbf{w}, \mathbf{w}_\sigma \sim N\left(\mathbf{w}^T \phi(x_i; \ell), \sigma^2(x_i; \mathbf{w}_\sigma, \ell)\right),$$

You will implement this model.

1.4.11 Part B.I

Complete the code below:

[ ]:
```
def model(X, y, num_centers=50):
    with pyro.plate("centers", num_centers):
        alpha = pyro.sample("alpha", dist.Exponential(1.0))
        w = pyro.sample("w", dist.Normal(0.0, alpha))
        # Let's add the generalized linear model for the log noise.
        alpha_noise = pyro.sample("alpha_noise", dist.Exponential(1.0))# Your␣
 ↪code here
        w_noise = pyro.sample("w_noise", dist.Normal(0.0, alpha_noise))# Your␣
 ↪code here
    ell = pyro.sample("ell", dist.Exponential(1.))
    ell_noise = pyro.sample("ell_noise", dist.Exponential(1.))# Your code here
    x_centers = torch.linspace(X.min(), X.max(), num_centers).unsqueeze(-1)
    Phi = RadialBasisFunctions(x_centers, ell)(X)
    Phi_noise = RadialBasisFunctions(x_centers, ell_noise)(X)# Your code here
    # This is the new part 2/2
    model_mean = Phi @ w
    sigma = torch.exp(Phi_noise @ w_noise)# Your code here (torch.
 ↪exp(<something>))
    with pyro.plate("data", X.shape[0]):
        pyro.sample("y", dist.Normal(model_mean, sigma), obs=y)
    return locals()
```

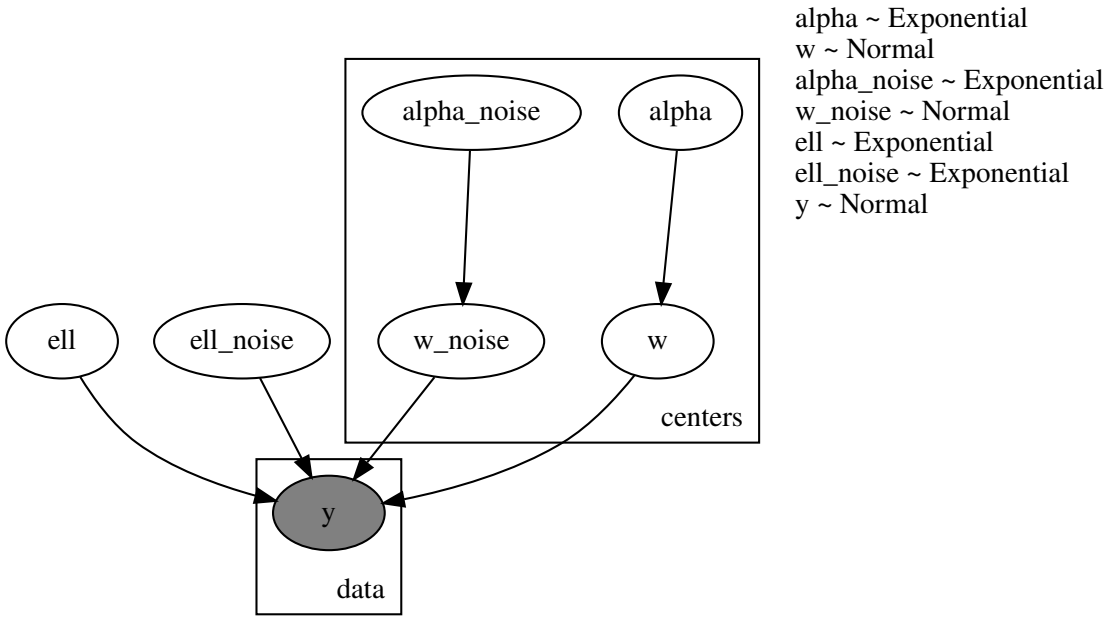
Make a `pyro.infer.autoguide.AutoDiagonalNormal` guide:

[ ]:
```
# Your code here
guide = pyro.infer.autoguide.AutoDiagonalNormal(model)
```

Make the graph of the model using `pyro` functionality:

[ ]:
```
# Your code here
pyro.render_model(model, (X, Y), render_distributions=True)
```

[ ]:



1.4.12 Part B.II

Train the model using 20,000 iterations. Then plot the evolution of the ELBO.

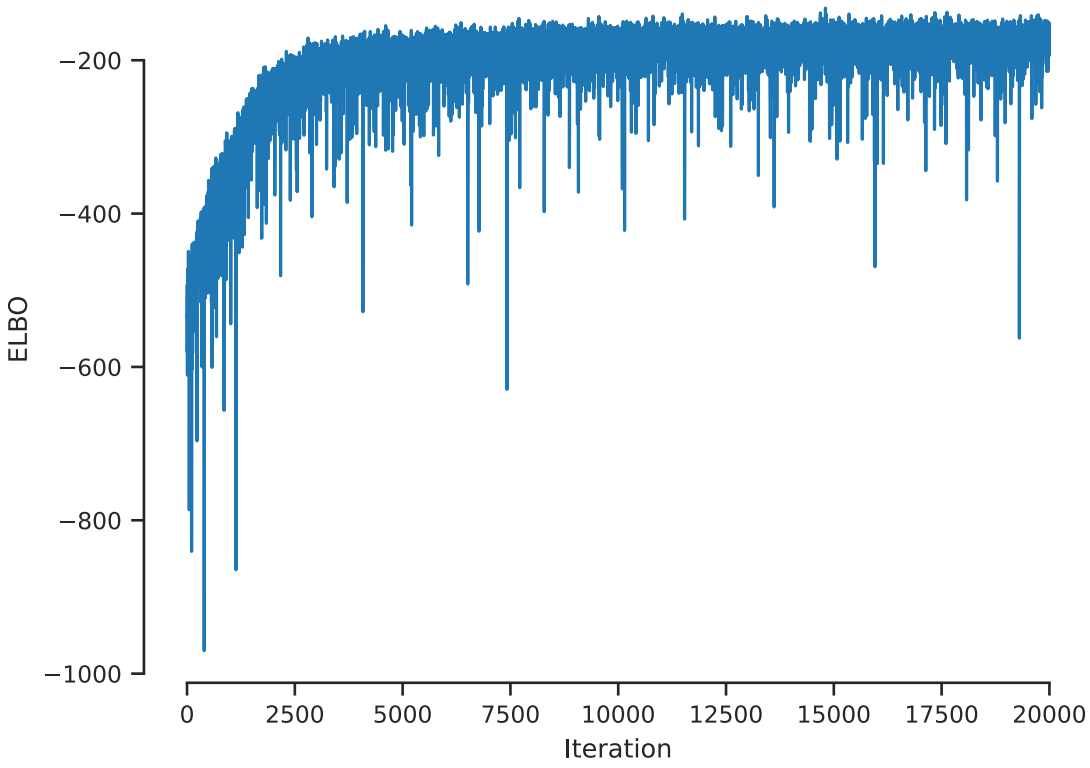
Answer:

[ ]:
```
# Your code here
elbo_b2, param_b2 = train(model,guide,(X,Y),num_iter=20000)
```

```
Iteration: 0 Loss: 580.2542724609375
Iteration: 1000 Loss: 352.0727233886719
Iteration: 2000 Loss: 257.3203430175781
Iteration: 3000 Loss: 203.912841796875
Iteration: 4000 Loss: 200.47584533691406
Iteration: 5000 Loss: 180.3367462158203
Iteration: 6000 Loss: 192.03884887695312
Iteration: 7000 Loss: 180.16738891601562
Iteration: 8000 Loss: 200.92495727539062
Iteration: 9000 Loss: 165.03445434570312
Iteration: 10000 Loss: 167.5047149658203
Iteration: 11000 Loss: 176.40805053710938
Iteration: 12000 Loss: 165.33364868164062
Iteration: 13000 Loss: 153.4978485107422
Iteration: 14000 Loss: 175.86474609375
```

```
Iteration: 15000 Loss: 189.97161865234375
Iteration: 16000 Loss: 173.80911254882812
Iteration: 17000 Loss: 172.23963928222656
Iteration: 18000 Loss: 159.88137817382812
Iteration: 19000 Loss: 186.75997924804688
```

[ ]:
```python
# Your code here
fig, ax = plt.subplots()
ax.plot(elbo_b2)
ax.set(xlabel='Iteration', ylabel='ELBO')
sns.despine(trim=True);
```



1.4.13 Part B.III

Extend the model to make predictions.

Answer:

[ ]:
```python
def predictive_model(X, y, num_centers=50):
    params = model(X, y, num_centers)
    w = params["w"]
    w_noise = params["w_noise"]# Your code here
```

```
    ell = params["ell"]# Your code here
    ell_noise = params["ell_noise"]# Your code here
    #sigma = params["sigma"]# Your code here
    x_centers = params["x_centers"]
    xs = torch.linspace(X.min(), X.max(), 100).unsqueeze(-1)
    Phi = RadialBasisFunctions(x_centers, ell)(xs)
    Phi_noise = RadialBasisFunctions(x_centers, ell_noise)(xs)# Your code here
    predictions = pyro.deterministic("predictions", Phi @ w)
    sigma = pyro.deterministic("sigma", torch.exp(Phi_noise @ w_noise))# Your␣
↪code here (pyro.deterministic("sigma", <something>))
    # Finally, we add the measurement noise
    predictions_with_noise = pyro.sample("predictions_with_noise", dist.
↪Normal(predictions, sigma))
    # predictions_with_noise = pyro.deterministic("predictions_with_noise", Phi␣
↪@ w)# Your code here
    return locals()
```

1.4.14 Part B.IV

Now, make predictions and calculate the epistemic and aleatory uncertainties as in part A.VII.

Answer:

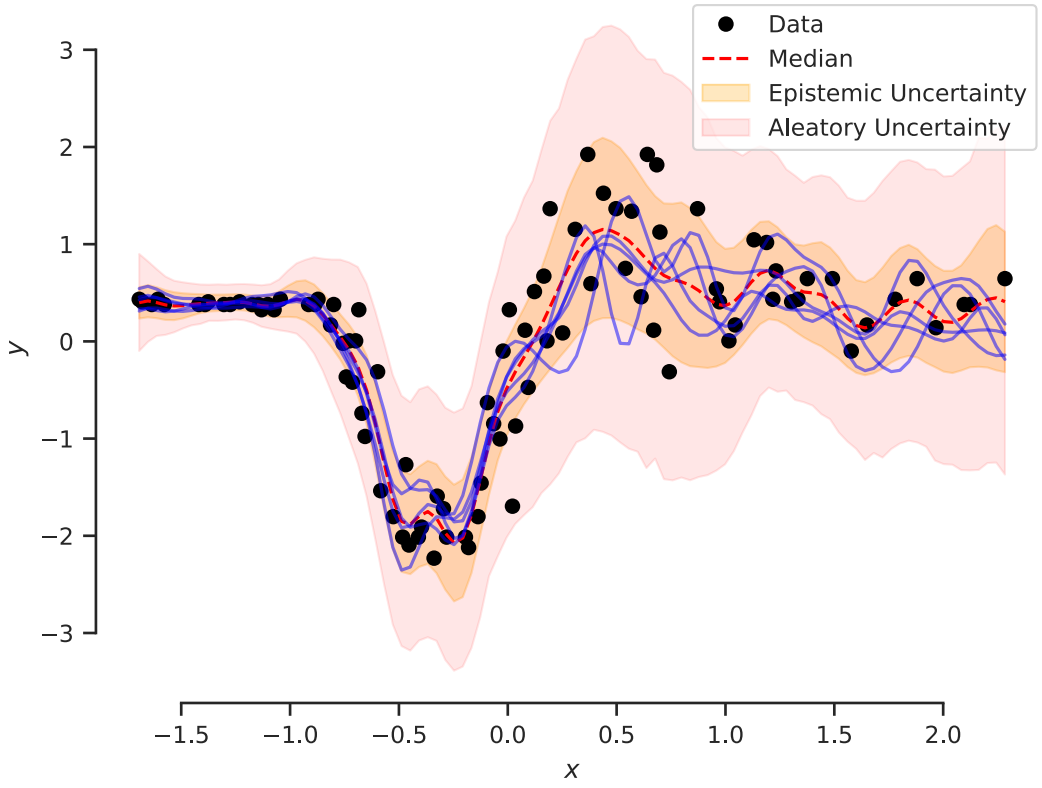
[ ]:
```
# Your code here
# Here is how to make the predictions. Just change the number of samples to the␣
↪right number.
post_pred = pyro.infer.Predictive(predictive_model, guide=guide,␣
↪num_samples=10000)(X, Y)
# We will predict here:
xs = torch.linspace(X.min(), X.max(), 100).unsqueeze(-1)
# You can extract the predictions from post_pred like this:
predictions = post_pred["predictions"]
# Note that we extracted the deterministic node called "predictions" from the␣
↪model.
# Get the epistemic uncertainty in the usual way:
p_500, p_025, p_975 = np.percentile(predictions, [50, 2.5, 97.5], axis=0)
# Extract predictions with noise
predictions_with_noise = post_pred["predictions_with_noise"]# Your code here
# Get the aleatory uncertainty
ap_025, ap_975 = np.percentile(predictions_with_noise, [2.5, 97.5], axis=0)#␣
↪Your code here
```

1.4.15 Part B.V

Make the same plot as in part A.VIII.

Answer:

[ ]:
```
# Your code here
# Your code here. You have everything you need to make the plot.
fig, ax = plt.subplots()
ax.plot(X, Y, 'ko', label="Data")
ax.plot(xs.flatten(), p_500.flatten(), 'r--', label="Median")
ax.fill_between(xs.flatten(), p_025.flatten(), p_975.flatten(), color='orange',␣
 ↪alpha=0.25, label="Epistemic Uncertainty")
ax.fill_between(xs.flatten(), ap_025.flatten(), ap_975.flatten(), color='r',␣
 ↪alpha=0.1, label="Aleatory Uncertainty")
ax.set(xlabel="$x$", ylabel="$y$")
# Plot a few samples
for i in range(5):
    ax.plot(xs.flatten(), predictions[i].flatten(), 'b', alpha=0.5)
ax.legend()
sns.despine(trim=True);
```

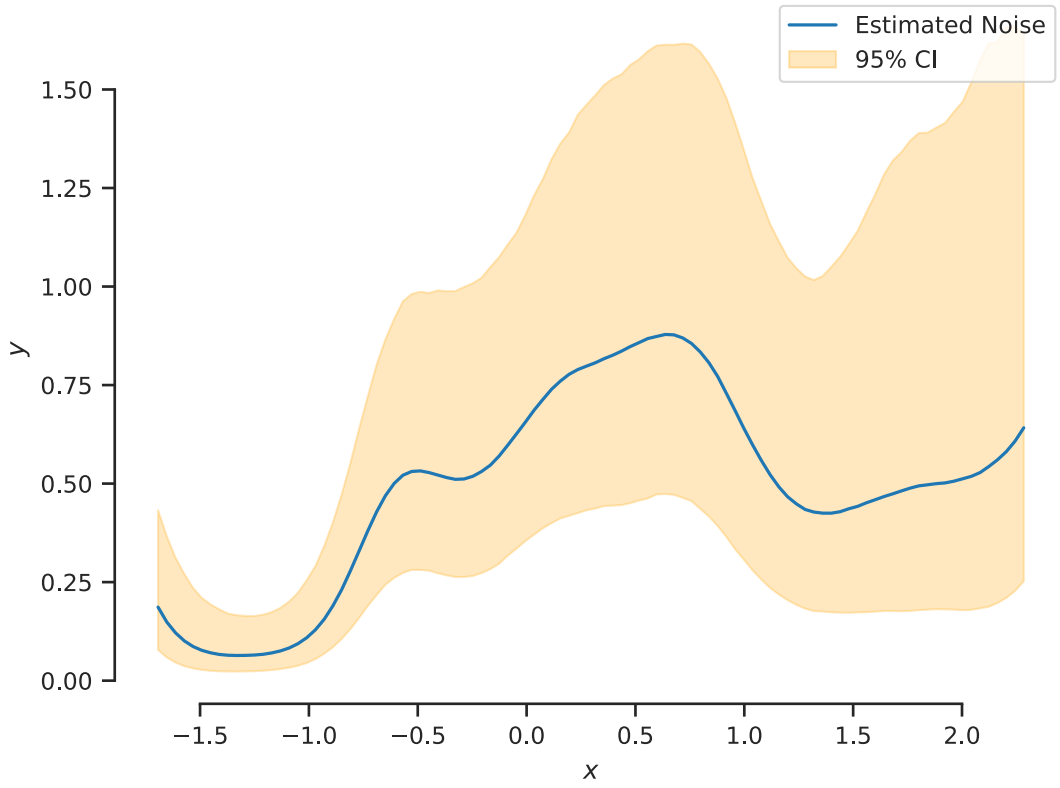


1.4.16 Part B.VI

Plot the estimated noise standard deviation as a function of of the input along with a 95% credible interval.

Answer:

```
[ ]: # Your code here
sigmas = post_pred["sigma"]
fig, ax = plt.subplots()
xs = torch.linspace(X.min(), X.max(), 100).unsqueeze(-1)
p_500, p_025, p_975 = np.percentile(sigmas, [50, 2.5, 97.5], axis=0)
ax.plot(xs.flatten(), p_500.flatten(), label="Estimated Noise")
ax.fill_between(xs.flatten(), p_025.flatten(), p_975.flatten(), color='orange',␣
 ↪alpha=0.25, label="95% CI")
ax.set(xlabel="$x$", ylabel="$y$")
ax.legend()
sns.despine(trim=True);
```



1.4.17 Part B.VII

Which model do you prefer? Why?

Answer: I will prefer the model made in Part B over Part A as the samples more closely follow the median. The addition of input-varying noise makes the model more application oriented compared to Part A model. The initial uncertainty in Part B is also less compared to Part A.

1.4.18 Part B.IX

Can you think of any way to improve the model? Go crazy! This is the last homework assignment! There is no right or wrong answer here. But if you have a good idea, we will give you extra credit.

[ ]:
```
## Your code and answers here
```