AM207_HW8

November 5, 2019

1 Homework #8 (Due 11/06/2019, 11:59pm)

1.1 Variational Inference for Bayesian Neural Networks

AM 207: Advanced Scientific Computing Instructor: Weiwei Pan Fall 2019

Name: Rylan Schaeffer

Students collaborators: Dimitris Vamvourellis, Dmitry Vukolov

1.1.1 Instructions:

Submission Format: Use this notebook as a template to complete your homework. Please intersperse text blocks (using Markdown cells) amongst `python` code and results -- format your submission for maximum readability. Your assignments will be graded for correctness as well as clarity of exposition and presentation -- a “right” answer by itself without an explanation or is presented with a difficult to follow format will receive no credit.

Code Check: Before submitting, you must do a ”Restart and Run All” under ”Kernel” in the Jupyter or colab menu. Portions of your submission that contains syntactic or run-time errors will not be graded.

Libraries and packages: Unless a problems specifically asks you to implement from scratch, you are welcomed to use any `python` library package in the standard Anaconda distribution.

[1]:
```
import numpy
import autograd.numpy as np
import autograd.numpy.random as npr
import autograd.scipy.stats.multivariate_normal as mvn
import autograd.scipy.stats.norm as norm
from autograd import grad
from autograd.misc.optimizers import adam
```

1.2 Problem Description: Bayesian Neural Network Regression

In Homework #7, you explored sampling from the posteriors of ***Bayesian neural networks*** using HMC. In Lab #8 you’ll explore the extent to which HMC can be inefficient or ineffective for sampling from certain types of posteriors. In this homework, you will study variational approximations of

BNN posteriors, especially when compared to the posteriors obtained by sampling (in Homework #7). The data is the same as the one for Homework #7.

1.2.1 Part I: Implement Black-Box Variational Inference with the Reparametrization Trick

1. (**BBVI with the Reparametrization Trick**) Implement BBVI with the reparametrization trick for approximating an arbitrary posterior $p(w|\text{Data})$ by an isotropic Gaussian $\mathcal{N}(\mu, \Sigma)$, where Σ is a diagonal matrix. See Lecture #15 or the example code from autograd's github repo.

[2]:
```python
# adapted from Lecture 15
def black_box_variational_inference(logprob, D, num_samples):
    """
    Implements http://arxiv.org/abs/1401.0118, and uses the
    local reparameterization trick from http://arxiv.org/abs/1506.02557
    code taken from:
    https://github.com/HIPS/autograd/blob/master/examples/black_box_svi.py
    """

    def unpack_params(params):
        # Variational dist is a diagonal Gaussian.
        mean, log_std = params[:D], params[D:]
        return mean, log_std

    def gaussian_entropy(log_std):
        return 0.5 * D * (1.0 + np.log(2 * np.pi)) + np.sum(log_std)

    rs = npr.RandomState(0)

    def variational_objective(params, t):
        """Provides a stochastic estimate of the variational lower bound."""
        mean, log_std = unpack_params(params)
        samples = rs.randn(num_samples, D) * np.exp(log_std) + mean
        lower_bound = gaussian_entropy(log_std) + np.mean(logprob(samples, t))
        return -lower_bound

    gradient = grad(variational_objective)

    return variational_objective, gradient, unpack_params
```

[3]:
```python
# adapted from lecture 15
def variational_inference(Sigma_W, y_train, x_train, S, max_iteration,␣
↪step_size, verbose):
    '''implements wrapper for variational inference via bbb for bayesian␣
↪regression'''
    D = Sigma_W.shape[0]
    Sigma_W_inv = np.linalg.inv(Sigma_W)
    Sigma_W_det = np.linalg.det(Sigma_W)
    variational_dim = D

    # define the log prior on the model parameters
    def log_prior(W):
        constant_W = -0.5 * (D * np.log(2 * np.pi) + np.log(Sigma_W_det))
        exponential_W = -0.5 * np.diag(np.dot(np.dot(W, Sigma_W_inv), W.T))
        log_p_W = constant_W + exponential_W
        return log_p_W

    # define the log likelihood
    def log_lklhd(W):
        log_odds = np.matmul(W, x_train) + 10
        p = 1 / (1 + np.exp(-log_odds))
        log_likelihood = y_train * np.log(p)
        return log_likelihood

    # define the log joint density
    log_density = lambda w, t: log_lklhd(w) + log_prior(w)

    # build variational objective.
    objective, gradient, unpack_params =␣
↪black_box_variational_inference(log_density, D, num_samples=S)

    def callback(params, t, g):
        if verbose:
            if verbose:
                if t % 250 == 0:
                    var_means = params[:D]
                    var_variance = np.diag(np.exp(params[D:]) ** 2)
                    print("Iteration {} lower bound {}; gradient mag: {}".
↪format(
                        t, -objective(params, t), np.linalg.
↪norm(gradient(params, t))))
                    print('Variational Mean: ', var_means)
                    print('Variational Variances: ', var_variance)
    print("Optimizing variational parameters...")
    # initialize variational parameters
    init_mean = 0 * np.ones(D)
    init_log_std = -1 * np.ones(D)
    init_var_params = np.concatenate([init_mean, init_log_std])

    # perform gradient descent using adam (a type of gradient-based optimizer)
    variational_params = adam(gradient, init_var_params, step_size=step_size,␣
↪num_iters=max_iteration,
                              callback=callback)
    return variational_params
```

[4]:
```
# adapted from lecture 15
def variational_bernoulli_regression(Sigma_W, x_train, y_train, S=2000,␣
↪max_iteration=2000, step_size=1e-2,
                                     verbose=True):
    '''perform bayesian regression: infer posterior, visualize posterior␣
↪predictive, compute log-likelihood'''

    D = Sigma_W.shape[0]

    # approximate posterior with mean-field gaussian
    variational_params = variational_inference(
        Sigma_W=Sigma_W,
        y_train=y_train,
        x_train=x_train,
        S=S,
        max_iteration=max_iteration,
        step_size=step_size,
        verbose=verbose)

    # sample from the variational posterior
    var_means = variational_params[:D]
    var_variance = np.diag(np.exp(variational_params[D:]) ** 2)

    return var_means, var_variance
```

2. (**Unit Test**) Check that your implementation is correct by approximating the posterior of the following Bayesian logistic regression model:

$$w \sim \mathcal{N}(0,1) \tag{1}$$
$$Y^{(n)} \sim Ber(\mathrm{sigm}(wX^{(n)} + 10)) \tag{2}$$

where w, $Y^{(n)}$, $X^{(n)}$ are a real scalar valued random variables, and where the data consists of a single observation $(Y = 1, X = -20)$.

The true posterior $p(w|Y = 1, X = -20)$ should look like the following (i.e. the true posterior is left-skewed): Your mean-field variational approximation should be a Gaussian with mean -0.321 and standard deviation 0.876 (all approximate).

[5]:
```
Sigma_W = np.eye(1)
ys = np.array([[1.]])
xs = np.array([[-20]])
```

[6]:
```
variational_mean, variational_variance = variational_bernoulli_regression(
    Sigma_W=Sigma_W,
x_train=xs,
y_train=ys,
S=4000,
max_iteration=2001,
step_size=1e-1,
verbose=True)
```

```
Optimizing variational parameters…
Iteration 0 lower bound -0.9000607591755687; gradient mag: 1.843344191550512
Variational Mean:  [0.]
Variational Variances:  [[0.13533528]]
Iteration 250 lower bound -0.5308554096777611; gradient mag: 0.04342946499575649
Variational Mean:  [-0.55278621]
Variational Variances:  [[0.3123244]]
Iteration 500 lower bound -0.5495332434781084; gradient mag:
0.024267477526000565
Variational Mean:  [-0.56220134]
Variational Variances:  [[0.29677678]]
Iteration 750 lower bound -0.5123105974402793; gradient mag: 0.10105491669518347
Variational Mean:  [-0.53714465]
Variational Variances:  [[0.30552718]]
Iteration 1000 lower bound -0.5076064397016626; gradient mag:
0.08304561811120983
Variational Mean:  [-0.56753442]
Variational Variances:  [[0.29998762]]
Iteration 1250 lower bound -0.5194934970867114; gradient mag:
0.019093999482919944
Variational Mean:  [-0.56728798]
Variational Variances:  [[0.29148606]]
Iteration 1500 lower bound -0.5539681364821437; gradient mag:
0.021034908574081055
Variational Mean:  [-0.5707026]
Variational Variances:  [[0.29863054]]
Iteration 1750 lower bound -0.5098963620858575; gradient mag:
0.13166963442394106
Variational Mean:  [-0.55570346]
Variational Variances:  [[0.2907907]]
Iteration 2000 lower bound -0.520567964484722; gradient mag:
0.043056366842566744
Variational Mean:  [-0.57835457]
Variational Variances:  [[0.310878]]
Iteration 2250 lower bound -0.5518084031691652; gradient mag:
0.027318127627461396
Variational Mean:  [-0.55354756]
Variational Variances:  [[0.30298749]]
Iteration 2500 lower bound -0.5356105093819084; gradient mag:
0.07485458364315366
```

```
Variational Mean:  [-0.55281665]
Variational Variances:  [[0.30796863]]
Iteration 2750 lower bound -0.549684746140452; gradient mag: 0.20416939071172485
Variational Mean:  [-0.54666155]
Variational Variances:  [[0.31864228]]
Iteration 3000 lower bound -0.5152972888029462; gradient mag: 0.1771629587640075
Variational Mean:  [-0.58965551]
Variational Variances:  [[0.29280793]]
```

[7]:
```
print('Variational Mean: ', variational_mean)
print('Variational Standard Deviation: ', np.sqrt(variational_variance))
```

```
Variational Mean:  [-0.56999084]
Variational Standard Deviation:  [[0.55224858]]
```

1.2.2 Part II: Approximate the Posterior of a Bayesian Neural Network

[8]:
```
import numpy
import autograd.numpy as np
import autograd.numpy.random as npr
import autograd.scipy.stats.multivariate_normal as mvn
import autograd.scipy.stats.norm as norm
from autograd import grad
from autograd.misc.optimizers import adam
```

[9]:
```
class Feedforward:
    def __init__(self, architecture, random=None, weights=None):
        self.params = {'H': architecture['width'],
                       'L': architecture['hidden_layers'],
                       'D_in': architecture['input_dim'],
                       'D_out': architecture['output_dim'],
                       'activation_type': architecture['activation_fn_type'],
                       'activation_params':␣
↪architecture['activation_fn_params']}

        self.D = ((architecture['input_dim'] * architecture['width'] +␣
↪architecture['width'])
                  + (architecture['output_dim'] * architecture['width'] +␣
↪architecture['output_dim'])
                  + (architecture['hidden_layers'] - 1) *␣
↪(architecture['width'] ** 2 + architecture['width'])
                  )

        if random is not None:
            self.random = random
        else:
            self.random = np.random.RandomState(0)

        self.h = architecture['activation_fn']

        if weights is None:
            self.weights = self.random.normal(0, 1, size=(1, self.D))
        else:
            self.weights = weights

        self.objective_trace = np.empty((1, 1))
        self.weight_trace = np.empty((1, self.D))

    def forward(self, weights, x):

        ''' Forward pass given weights and input '''
        H = self.params['H']
        D_in =  self.params['D_in']
        D_out = self.params['D_out']

        assert weights.shape[1] == self.D

        if len(x.shape) == 2:
            assert x.shape[0] == D_in
            x = x.reshape((1, D_in, -1))
        else:
            assert x.shape[1] == D_in

        weights = weights.T

        # input to first hidden layer
        W = weights[:H * D_in].T.reshape((-1, H, D_in))
        b = weights[H * D_in:H * D_in + H].T.reshape((-1, H, 1))
        input = self.h(np.matmul(W, x) + b)
        index = H * D_in + H

        assert input.shape[1] == H

        # additional hidden layers
        for _ in range(self.params['L'] - 1):
            before = index
            W = weights[index:index + H * H].T.reshape((-1, H, H))
            index += H * H
            b = weights[index:index + H].T.reshape((-1, H, 1))
            index += H
            output = np.matmul(W, input) + b
            input = self.h(output)
            assert input.shape[1] == H

        # output layer
        W = weights[index:index + H * D_out].T.reshape((-1, D_out, H))
        b = weights[index + H * D_out:].T.reshape((-1, D_out, 1))
        output = np.matmul(W, input) + b
        assert output.shape[1] == self.params['D_out']
        # output = output.squeeze(1)  # Rylan added
        return output

    def make_objective(self, x_train, y_train, reg_param=None):
        ''' Make objective functions: depending on whether or not you want to␣
↪apply l2 regularization '''

        if reg_param is None:

            def objective(W, t):
                squared_error = np.linalg.norm(y_train - self.forward(W,␣
↪x_train), axis=1) ** 2
                sum_error = np.sum(squared_error)
                return sum_error

            return objective, grad(objective)

        else:

            def objective(W, t):
                squared_error = np.linalg.norm(y_train - self.forward(W,␣
↪x_train), axis=1) ** 2
                mean_error = np.mean(squared_error) + reg_param * np.linalg.
↪norm(W)
                return mean_error

            return objective, grad(objective)

    def fit(self, x_train, y_train, params, reg_param=None):
        ''' Wrapper for MLE through gradient descent '''
        assert x_train.shape[0] == self.params['D_in']
        assert y_train.shape[0] == self.params['D_out']

        ### make objective function for training
        self.objective, self.gradient = self.make_objective(x_train, y_train,␣
↪reg_param)

        ### set up optimization
        step_size = 0.01
        max_iteration = 5000
        check_point = 100
        weights_init = self.weights.reshape((1, -1))
        mass = None
        optimizer = 'adam'
        random_restarts = 5

        if 'step_size' in params.keys():
            step_size = params['step_size']
        if 'max_iteration' in params.keys():
            max_iteration = params['max_iteration']
        if 'check_point' in params.keys():
            self.check_point = params['check_point']
        if 'init' in params.keys():
            weights_init = params['init']
        if 'call_back' in params.keys():
            call_back = params['call_back']
        if 'mass' in params.keys():
            mass = params['mass']
        if 'optimizer' in params.keys():
            optimizer = params['optimizer']
        if 'random_restarts' in params.keys():
            random_restarts = params['random_restarts']

        def call_back(weights, iteration, g):
            ''' Actions per optimization step '''
            objective = self.objective(weights, iteration)
            self.objective_trace = np.vstack((self.objective_trace, objective))
            self.weight_trace = np.vstack((self.weight_trace, weights))
            if iteration % check_point == 0:
                print("Iteration {} lower bound {}; gradient mag: {}".
↪format(iteration, objective, np.linalg.norm(
                    self.gradient(weights, iteration))))

        ### train with random restarts
        optimal_obj = 1e16
        optimal_weights = self.weights

        for i in range(random_restarts):
            if optimizer == 'adam':
                adam(self.gradient, weights_init, step_size=step_size,␣
↪num_iters=max_iteration, callback=call_back)
            local_opt = np.min(self.objective_trace[-100:])

            if local_opt < optimal_obj:
                opt_index = np.argmin(self.objective_trace[-100:])
                self.weights = self.weight_trace[-100:][opt_index].reshape((1,␣
↪-1))
```

```
                weights_init = self.random.normal(0, 1, size=(1, self.D))

        self.objective_trace = self.objective_trace[1:]
        self.weight_trace = self.weight_trace[1:]
```

[10]:
```
###define rbf activation function
alpha = 1
c = 0
h = lambda x: np.exp(-alpha * (x - c)**2)

###neural network model design choices
width = 5
hidden_layers = 1
input_dim = 1
output_dim = 1

architecture = {'width': width,
               'hidden_layers': hidden_layers,
               'input_dim': input_dim,
               'output_dim': output_dim,
               'activation_fn_type': 'rbf',
               'activation_fn_params': 'c=0, alpha=1',
               'activation_fn': h}

params = {'step_size':1e-3,
          'max_iteration': 6001,
          'random_restarts':1,
          'check_point':200}

#set random state to make the experiments replicable
rand_state = 0
random = np.random.RandomState(rand_state)

#instantiate a Feedforward neural network object
nn = Feedforward(architecture, random=random)
```

[11]:
```
import pandas as pd


df = pd.read_csv('HW7_data.csv')
print(df.head())
xs = df['x'].values
ys = df['y'].values
```

```
     x         y
0 -6.0 -3.380284
1 -5.6 -2.892117
```

```
2 -5.2 -2.690059
3 -4.8 -2.040000
4 -4.4 -1.399942
```

[12]:
```
# train the netework
nn.fit(xs.reshape((1, -1)), ys.reshape((1, -1)), params)
```

```
Iteration 0 lower bound 65.11668053773148; gradient mag: 164.7321094609157
Iteration 100 lower bound 52.864360467566954; gradient mag: 56.954586390884565
Iteration 200 lower bound 49.42633385370296; gradient mag: 28.31627548203755
Iteration 300 lower bound 47.733261895623144; gradient mag: 18.31175187956584
Iteration 400 lower bound 46.64774569354082; gradient mag: 13.47755254054697
Iteration 500 lower bound 45.886915428058316; gradient mag: 10.503599898310592
Iteration 600 lower bound 45.25067576634; gradient mag: 8.654932255566825
Iteration 700 lower bound 30.562858257343013; gradient mag: 24.1925753228009
Iteration 800 lower bound 26.977925085924852; gradient mag: 20.354176994190933
Iteration 900 lower bound 24.278363637665738; gradient mag: 18.77912591689822
Iteration 1000 lower bound 22.02986716940495; gradient mag: 17.427416991221943
Iteration 1100 lower bound 20.10610040886032; gradient mag: 16.21045518866592
Iteration 1200 lower bound 18.434367471681025; gradient mag: 15.092044558814075
Iteration 1300 lower bound 16.967826824623163; gradient mag: 14.051620576729906
Iteration 1400 lower bound 15.673763205688545; gradient mag: 13.07650688789188
Iteration 1500 lower bound 14.527850196693981; gradient mag: 12.15858303432993
Iteration 1600 lower bound 13.51104118089244; gradient mag: 11.292691333772902
Iteration 1700 lower bound 12.607737627992936; gradient mag: 10.475901826346675
Iteration 1800 lower bound 11.804614404003857; gradient mag: 9.707342852428729
Iteration 1900 lower bound 11.08977327305976; gradient mag: 8.988599356616401
Iteration 2000 lower bound 10.452003162028154; gradient mag: 8.324954075223918
Iteration 2100 lower bound 9.879926845203448; gradient mag: 7.728107510229664
Iteration 2200 lower bound 9.360728932888751; gradient mag: 7.221146412450928
Iteration 2300 lower bound 8.878118698993092; gradient mag: 6.843182251011429
Iteration 2400 lower bound 8.410371873546008; gradient mag: 6.627021939587406
Iteration 2500 lower bound 7.937214402172411; gradient mag: 6.461888287390539
Iteration 2600 lower bound 7.471807925987622; gradient mag: 6.054967192028708
Iteration 2700 lower bound 7.048560738747636; gradient mag: 5.623261826592469
Iteration 2800 lower bound 6.657649714610486; gradient mag: 5.301596015489052
Iteration 2900 lower bound 6.286937924356821; gradient mag: 5.005817266509646
Iteration 3000 lower bound 5.932348941430825; gradient mag: 4.738230196194087
Iteration 3100 lower bound 5.591133620620948; gradient mag: 4.496972816298515
Iteration 3200 lower bound 5.26127545540811; gradient mag: 4.278187646372161
Iteration 3300 lower bound 4.941457959341479; gradient mag: 4.078179065269128
Iteration 3400 lower bound 4.6310060561160205; gradient mag: 3.8933742950112133
Iteration 3500 lower bound 4.329797769584352; gradient mag: 3.7204132445853806
Iteration 3600 lower bound 4.038157311705763; gradient mag: 3.556269206709465
Iteration 3700 lower bound 3.7567400639178525; gradient mag: 3.398336991338033
Iteration 3800 lower bound 3.4864181903710856; gradient mag: 3.2444781370885503
Iteration 3900 lower bound 3.2281738180915545; gradient mag: 3.09302784318385
Iteration 4000 lower bound 2.983004984081998; gradient mag: 2.9427729212293867
Iteration 4100 lower bound 2.7518476624500336; gradient mag: 2.7929113696219288
Iteration 4200 lower bound 2.535515098891551; gradient mag: 2.643003494153074
Iteration 4300 lower bound 2.3346536991188067; gradient mag: 2.4929221870323546
Iteration 4400 lower bound 2.149713314288466; gradient mag: 2.3428066637339215
Iteration 4500 lower bound 1.9809292604652133; gradient mag: 2.193020647459896
Iteration 4600 lower bound 1.8283137632422362; gradient mag: 2.0441136627500742
Iteration 4700 lower bound 1.6916553649649706; gradient mag: 1.896783257659039
Iteration 4800 lower bound 1.5705256812596617; gradient mag: 1.7518364695730388
Iteration 4900 lower bound 1.464293373837468; gradient mag: 1.6101500586762574
Iteration 5000 lower bound 1.372145194334405; gradient mag: 1.472630238301942
Iteration 5100 lower bound 1.293113555323543; gradient mag: 1.3401733579691384
Iteration 5200 lower bound 1.2261095200674328; gradient mag: 1.2136291284746006
Iteration 5300 lower bound 1.1699595835060193; gradient mag: 1.0937676769971116
Iteration 5400 lower bound 1.1234442741889101; gradient mag: 0.9812512421063398
Iteration 5500 lower bound 1.085336480987702; gradient mag: 0.8766108846616596
Iteration 5600 lower bound 1.0544374790480324; gradient mag: 0.7802283270250203
Iteration 5700 lower bound 1.0296088526691245; gradient mag: 0.6923229786568087
Iteration 5800 lower bound 1.0097988475753614; gradient mag: 0.6129443440319642
Iteration 5900 lower bound 0.9940621024101455; gradient mag: 0.5419702880196495
Iteration 6000 lower bound 0.9815721882281818; gradient mag: 0.4791119652013911
```

1. (**Variational Inference for BNNs**) We will implement the following Bayesian model for the data:

$$\mathbf{W} \sim \mathcal{N}(0, 5^2\mathbf{I}_{D\times D}) \tag{3}$$
$$\mu^{(n)} = g_{\mathbf{W}}(\mathbf{X}^{(n)}) \tag{4}$$
$$Y^{(n)} \sim \mathcal{N}(\mu^{(n)}, 0.5^2) \tag{5}$$

   where $g_{\mathbf{W}}$ is a neural network with parameters $\mathbf{W}$ represented as a vector in $\mathbb{R}^D$ with D being the total number of parameters (including biases). Just as in HW #7, use a network with a single hidden layer, 5 hidden nodes and rbf activation function.

Implement the log of the joint distribution in `autograd`'s version of `numpy`, i.e. implement $\log\left[p(\mathbf{W})\prod_{n=1}^{N} p(Y^{(n)}|\mathbf{X}^{(n)}, \mathbf{W})\right]$.

Hint: you'll need to write out the log of the various Gaussian pdf's and implement their formulae using `autograd`'s numpy functions.

We first define the log joint distribution and then drop all terms that do not depend on W:

$$
\begin{aligned}
\log P(W, Y|X) &= \log\left[p(W)\prod_{n=1}^{N} p(Y^{(n)}|X^{(n)}, W)\right] && (6)\\
&= \log p(W) + \sum_{n=1}^{N}\log p(Y^{(n)}|X^{(n)}, W) && (7)\\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma| - \frac{1}{2}W^T(5^2 I)^{-1}W + \sum_{n=1}^{N}\log p(Y^{(n)}|X^{(n)}, W) && (8)\\
&\propto -\frac{1}{2}W^T(5^2 I)^{-1}W + \sum_{n=1}^{N}\log p(Y^{(n)}|X^{(n)}, W) && (9)\\
&= -\frac{1}{2}W^T(5^2 I)^{-1}W + \sum_{n=1}^{N} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log|0.5^2| - \frac{1}{2}(Y^{(n)} - \mu^{(n)})^T(0.5^2)^{-1}(Y^{(n)} - \mu^{(n)}) && (10)\\
&\propto -\frac{1}{2}W^T(5^2 I)^{-1}W + \sum_{n=1}^{N} -\frac{1}{2}(Y^{(n)} - g_W(X^{(n)}))^T(0.5^2)^{-1}(Y^{(n)} - g_W(X^{(n)})) && (11)
\end{aligned}
$$

[13]:
```
def log_joint(W, ys, xs):
    term1 = -0.5 * W @ W.T / 25
    yhats = nn.forward(W, xs.reshape(1, -1))
    diff = (yhats - ys).reshape((-1, 1))  # shape [S*12, 1]
    term2 = -0.5 * 4 * diff.T @ diff
    return term1 + term2
```

4. (**Approximate the Posterior**) Use BBVI with the reparametrization trick to approximate the posterior of the Bayesian neural network with a mean-field Gaussian variational family (i.e. an isotropic Gaussian). Please set learning rate and maximum iteration choices as you see fit!

[14]:
```
# adapted from Lecture 15
def black_box_variational_inference(logprob, D, num_samples):
    """
    Implements http://arxiv.org/abs/1401.0118, and uses the
    local reparameterization trick from http://arxiv.org/abs/1506.02557
    code taken from:
    https://github.com/HIPS/autograd/blob/master/examples/black_box_svi.py
    """

    def unpack_params(params):
        # Variational dist is a diagonal Gaussian.
        mean, log_std = params[:D], params[D:]
        return mean, log_std

    def gaussian_entropy(log_std):
        return 0.5 * D * (1.0 + np.log(2 * np.pi)) + np.sum(log_std)
    rs = npr.RandomState(0)

    def variational_objective(params, t):
        """Provides a stochastic estimate of the variational lower bound."""
        mean, log_std = unpack_params(params)
        samples = rs.randn(num_samples, D) * np.exp(log_std) + mean
        lower_bound = gaussian_entropy(log_std) + np.mean(logprob(samples, t))
        return -lower_bound

    gradient = grad(variational_objective)

    return variational_objective, gradient, unpack_params
```

[15]:
```
# adapted from lecture 15
def variational_nn_inference(Sigma_W, y_train, x_train, S, max_iteration,␣
 ↪step_size, verbose):
    '''implements wrapper for variational inference via bbb for bayesian␣
 ↪regression'''
    D = Sigma_W.shape[0]
    Sigma_W_inv = np.linalg.inv(Sigma_W)
    Sigma_W_det = np.linalg.det(Sigma_W)
    variational_dim = D

    # define the log prior on the model parameters
    #     def log_prior(W):
    #         constant_W = -0.5 * (D * np.log(2 * np.pi) + np.log(Sigma_W_det))
    #         exponential_W = -0.5 * np.diag(np.dot(np.dot(W, Sigma_W_inv), W.
 ↪T))
    #         log_p_W = constant_W + exponential_W
    #         return log_p_W

    # define the log likelihood
    #     def log_lklhd(W):
    #         log_odds = np.matmul(W, x_train) + 10
    #         p = 1 / (1 + np.exp(-log_odds))
    #         log_likelihood = y_train * np.log(p)
    #         return log_likelihood

    def log_joint(W, ys, xs):
        term1 = -0.5 * W @ W.T / 25
        yhats = nn.forward(W, xs.reshape(1, -1))
        diff = (yhats - ys).reshape((-1, 1))  # shape [S*12, 1]
        term2 = -0.5 * 4 * diff.T @ diff
        return term1 + term2

    # define the log joint density
    #     log_density = lambda w, t: log_lklhd(w) + log_prior(w)
    log_density = lambda w, t: log_joint(w, xs=x_train, ys=y_train)

    # build variational objective.
    objective, gradient, unpack_params =␣
↪black_box_variational_inference(log_density, D, num_samples=S)

    def callback(params, t, g):
        if verbose:
            if verbose:
                if t % 100 == 0:
                    var_means = params[:D]
                    var_variance = np.exp(params[D:]) ** 2
                    # var_variance = np.diag(np.exp(params[D:]) ** 2)
                    print("Iteration {} lower bound {}; gradient mag: {}".
↪format(
                        t, -objective(params, t), np.linalg.
↪norm(gradient(params, t))))
                    print('Variational Mean: ', var_means)
                    print('Variational Variances: ', var_variance)

    print("Optimizing variational parameters...")
    # initialize variational parameters
    init_mean = 0 * np.ones(D)
    init_log_std = -1 * np.ones(D)
    init_var_params = np.concatenate([init_mean, init_log_std])

    # perform gradient descent using adam (a type of gradient-based optimizer)
    variational_params = adam(gradient, init_var_params, step_size=step_size,␣
↪num_iters=max_iteration,
                              callback=callback)

    return variational_params
```

[16]:
```
D = 16
Sigma_W = np.eye(D)
num_samples = 5000
max_iterations = 3001
step_size = 1e-2

# infer variational parameters
variational_params = variational_nn_inference(
        Sigma_W=Sigma_W,
        y_train=ys,
        x_train=xs,
        S=num_samples,
        max_iteration=max_iterations,
        step_size=step_size,
        verbose=True)
var_means = variational_params[:D]
var_variance = np.diag(np.exp(variational_params[D:])**2)
```

```
Optimizing variational parameters…
Iteration 0 lower bound -499088.85060063924; gradient mag: 124748.37585189997
Variational Mean:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Variational Variances:  [0.13533528 0.13533528 0.13533528 0.13533528 0.13533528
0.13533528
 0.13533528 0.13533528 0.13533528 0.13533528 0.13533528 0.13533528
 0.13533528 0.13533528 0.13533528 0.13533528]
Iteration 100 lower bound -326942.78145667096; gradient mag: 110582.8277529181
Variational Mean:  [ 0.11549171 -0.32702823 -0.60360318  0.52325774 -0.31336212
0.05697346
 -1.26810076 -1.02546221  1.10871979 -1.30505927  0.16506251 -0.70370501
 -0.34246525 -0.44299465 -0.74577595 -0.13193039]
Variational Variances:  [0.48437386 0.04990461 0.12465691 0.09840125 0.04469691
0.25487718
 0.06530163 0.14198429 0.12331333 0.06069639 0.03717247 0.03845898
 0.03890301 0.03955482 0.03761358 0.030959  ]
Iteration 200 lower bound -55308.88250175553; gradient mag: 32460.827054098463
Variational Mean:  [ 0.09727019 -0.33058951 -0.33065975  0.34228221 -0.33259613
-0.52766484
 -2.154778   -2.26841459  2.27788651 -2.15967015  0.40783225 -1.38710838
 -0.85082594 -1.08477299 -1.42161311  0.94698434]
Variational Variances:  [0.02486612 0.00448313 0.00581169 0.00504466 0.00435645
0.03805812
 0.00602882 0.00911005 0.00711926 0.00622181 0.01416633 0.0141562
 0.01682216 0.01571289 0.01403617 0.01443875]
Iteration 300 lower bound -33512.88767288942; gradient mag: 23277.652456028245
Variational Mean:  [ 0.21863701 -0.40529777 -0.4358092   0.43487966 -0.4070042
-1.6452272
 -2.52825497 -2.79440195  2.7604017  -2.53208262  0.29226162 -1.30565369
 -0.92618234 -1.08611026 -1.34697917  1.12804574]
Variational Variances:  [0.00868246 0.00206857 0.00189788 0.00189826 0.00205878
0.01879937
 0.00252165 0.00258343 0.00231933 0.0026183  0.00714307 0.00768993
 0.00898952 0.00861819 0.00770003 0.00864918]
Iteration 400 lower bound -27565.82579857899; gradient mag: 6661.508085014373
Variational Mean:  [ 0.42436917 -0.44451931 -0.49257577  0.48481418 -0.44788297
-2.66226975
 -2.69330782 -3.04622337  2.97799004 -2.69678173  0.35842176 -1.22607911
 -0.97715584 -1.07044308 -1.27097321  1.11017711]
Variational Variances:  [0.00497842 0.00138026 0.00105408 0.00115785 0.00137889
0.00879905
 0.00159865 0.00137981 0.00135305 0.0016635  0.00500639 0.00490821
 0.00574062 0.00552716 0.00493654 0.00585239]
Iteration 500 lower bound -23204.299024144344; gradient mag: 6787.410037722172
Variational Mean:  [ 0.7102206  -0.46286971 -0.51583803  0.50531681 -0.46555911
-3.86940722
 -2.78520479 -3.19026714  3.09914558 -2.78860545  0.57431472 -1.14752461
 -0.98474034 -1.02910262 -1.19521605  0.95426935]
Variational Variances:  [0.00244774 0.0010501  0.00070185 0.00083049 0.00104715
0.00362461
 0.00118026 0.00091679 0.00094657 0.00123542 0.00361115 0.00343548
 0.00404944 0.00391777 0.00347053 0.00426498]
Iteration 600 lower bound -18416.498731376018; gradient mag: 6220.203879395312
Variational Mean:  [ 0.98945356 -0.47457615 -0.53733442  0.52318425 -0.47682811
-5.11147818
 -2.86837972 -3.3057382   3.19789318 -2.8723455   0.94076528 -1.06491326
 -0.96978805 -0.97308756 -1.11600889  0.73974018]
Variational Variances:  [0.00095915 0.00085091 0.00051494 0.00064644 0.00084612
0.00137871
 0.00094146 0.00066584 0.0007314  0.00098149 0.00274423 0.00258873
 0.00306168 0.002957   0.00260846 0.0032661 ]
Iteration 700 lower bound -16484.981188597536; gradient mag: 3443.48853535722
Variational Mean:  [ 1.08041529 -0.48708108 -0.56095538  0.54153346 -0.48952749
-5.53459351
 -2.95472316 -3.41601665  3.29301379 -2.96021065  1.19840788 -0.99296451
 -0.97587473 -0.93538483 -1.04664503  0.58913718]
Variational Variances:  [0.00042988 0.00071806 0.00040164 0.00053025 0.00071117
0.00062909
 0.00078456 0.00051596 0.00059309 0.00081171 0.00222618 0.00204266
 0.00240119 0.00233105 0.00205992 0.0025916 ]
Iteration 800 lower bound -15549.512960670785; gradient mag: 8919.668859065718
Variational Mean:  [ 1.04390587 -0.50068441 -0.5830532   0.55896369 -0.50400546
-5.35774258
 -3.03261657 -3.5118315   3.37566912 -3.03989879  1.3152232  -0.93351124
 -0.99871171 -0.91479736 -0.98965725  0.49017366]
Variational Variances:  [0.00024341 0.00062316 0.00032359 0.00044862 0.00061489
0.00036232
 0.00067526 0.00041516 0.00049914 0.00069692 0.00185137 0.00166483
 0.00193696 0.00188627 0.0016794  0.0021119 ]
Iteration 900 lower bound -14921.29843202288; gradient mag: 4088.900086760849
Variational Mean:  [ 0.99396859 -0.51142646 -0.60104486  0.5727486  -0.51436764
-5.11180122
 -3.10218071 -3.59375018  3.44723796 -3.11184793  1.40879686 -0.87936451
 -1.02131934 -0.89620475 -0.93751212  0.39903985]
Variational Variances:  [0.00015553 0.00055246 0.00026666 0.00038769 0.00054281
0.0002354
 0.0005946  0.00034195 0.00042944 0.00061296 0.0015539  0.00138938
 0.0015985  0.00156387 0.00140705 0.00175736]
Iteration 1000 lower bound -14421.336337659934; gradient mag: 4236.592938758901
Variational Mean:  [ 0.95503766 -0.51879901 -0.61832986  0.58507916 -0.52406458
```

```
-4.91224014
 -3.16720371 -3.66822626  3.512666   -3.1787652   1.49629868 -0.82870014
 -1.04488612 -0.87951028 -0.8890619   0.31346231]
Variational Variances:  [0.00010696 0.00049765 0.00022383 0.00034045 0.00048693
0.00016412
 0.0005341  0.00028584 0.00037813 0.00054824 0.00131426 0.0011844
 0.00133908 0.00131989 0.00119736 0.00148725]
Iteration 1100 lower bound -14058.640680266426; gradient mag: 5748.490066333744
Variational Mean:  [ 0.92420404 -0.52674083 -0.63628295  0.59603321 -0.53206001
-4.75640527
 -3.2276466  -3.73974347  3.57451524 -3.24222395  1.5749137  -0.78243876
 -1.07192555 -0.86682581 -0.84424383  0.23638987]
Variational Variances:  [7.73882831e-05 4.53807854e-04 1.90045801e-04
3.02531063e-04
 4.41926296e-04 1.20632341e-04 4.87362698e-04 2.41873562e-04
 3.37059263e-04 4.97482173e-04 1.12166846e-03 1.02765086e-03
 1.13881985e-03 1.13133687e-03 1.03536495e-03 1.27525031e-03]
Iteration 1200 lower bound -13766.539870042536; gradient mag: 5626.939070999725
Variational Mean:  [ 0.89976069 -0.53078991 -0.65463579  0.60505823 -0.53672416
-4.63277756
 -3.28455088 -3.81036354  3.63277257 -3.30197044  1.64487732 -0.73964758
 -1.10431606 -0.85800628 -0.80347443  0.16865287]
Variational Variances:  [5.82654256e-05 4.17562805e-04 1.62957166e-04
2.71114330e-04
 4.04975384e-04 9.17640672e-05 4.49926204e-04 2.06089547e-04
 3.02243527e-04 4.57445518e-04 9.66237495e-04 9.02962047e-04
 9.76773782e-04 9.80367594e-04 9.10020038e-04 1.10514863e-03]
Iteration 1300 lower bound -13547.085663074988; gradient mag: 3957.2374296912203
Variational Mean:  [ 0.87863168 -0.53271684 -0.67379787  0.61173632 -0.54011388
-4.53448945
 -3.3365294  -3.88281947  3.68600804 -3.35673202  1.70572834 -0.70111401
 -1.14334699 -0.8542198  -0.76704315  0.10932344]
Variational Variances:  [4.52289949e-05 3.86933526e-04 1.40539464e-04
2.44361797e-04
 3.73836478e-04 7.19659921e-05 4.18631895e-04 1.78568915e-04
 2.73106182e-04 4.23887778e-04 8.38674485e-04 8.04029565e-04
 8.45802407e-04 8.61722046e-04 8.08729448e-04 9.67464046e-04]
Iteration 1400 lower bound -13374.40991327039; gradient mag: 2405.9344365803213
Variational Mean:  [ 0.86249802 -0.53186788 -0.69844471  0.61187105 -0.54211476
-4.45388445
 -3.38397586 -3.96084728  3.73420005 -3.40669927  1.75866214 -0.66903341
 -1.19360304 -0.8583735  -0.73658122  0.05748966]
Variational Variances:  [3.59972901e-05 3.60463266e-04 1.21460607e-04
2.20607235e-04
 3.46948124e-04 5.79033720e-05 3.92914631e-04 1.54611114e-04
 2.48872895e-04 3.95441166e-04 7.33399899e-04 7.26296210e-04
 7.37714353e-04 7.66867325e-04 7.29030959e-04 8.53275467e-04]
Iteration 1500 lower bound -13251.71339940953; gradient mag: 5801.352064892366
```

```
Variational Mean:  [ 0.85080421 -0.52955305 -0.72557222  0.60682257 -0.54049126
-4.38602625
 -3.42784214 -4.05615058  3.77722861 -3.45378098  1.80518452 -0.64634055
 -1.26170738 -0.87657491 -0.71452529  0.01254802]
Variational Variances:  [2.92674495e-05 3.36746643e-04 1.04406149e-04
1.98461127e-04
 3.22800728e-04 4.76353591e-05 3.70016843e-04 1.32700602e-04
 2.25774553e-04 3.71093997e-04 6.45807978e-04 6.63617378e-04
 6.47216509e-04 6.91064211e-04 6.64402252e-04 7.57564086e-04]
Iteration 1600 lower bound -13083.718157511503; gradient mag: 1707.0986015823298
Variational Mean:  [ 0.83865248 -0.52550946 -0.75535396  0.5959501  -0.5363893
-4.33192039
 -3.46956193 -4.170275    3.81777139 -3.49825261  1.84559045 -0.6332515
 -1.34450756 -0.90821858 -0.7009411  -0.02636157]
Variational Variances:  [2.42279631e-05 3.14974912e-04 8.88989171e-05
1.76800138e-04
 3.00565511e-04 3.98205212e-05 3.49470589e-04 1.13717413e-04
 2.03981098e-04 3.48597562e-04 5.72531223e-04 6.14078159e-04
 5.71940589e-04 6.33469098e-04 6.12916310e-04 6.77065298e-04]
Iteration 1700 lower bound -12970.163482795559; gradient mag: 3286.6236993888588
Variational Mean:  [ 0.82924725 -0.51959644 -0.7838094   0.58547441 -0.52900536
-4.28435965
 -3.51068924 -4.29091459  3.85934899 -3.54166682  1.88160263 -0.63067461
 -1.4342434  -0.95888619 -0.69683315 -0.06120996]
Variational Variances:  [2.03667676e-05 2.94386473e-04 7.48844664e-05
1.55583929e-04
 2.79537072e-04 3.38349901e-05 3.30484512e-04 9.65275924e-05
 1.83677157e-04 3.28163810e-04 5.10509369e-04 5.74264588e-04
 5.09174043e-04 5.90393044e-04 5.72463695e-04 6.08438891e-04]
Iteration 1800 lower bound -12851.152315186264; gradient mag: 4560.906331292819
Variational Mean:  [ 0.82138669 -0.51345845 -0.80928395  0.57921397 -0.52133237
-4.24310904
 -3.55186362 -4.39901049  3.90648018 -3.58462289  1.91320226 -0.63780793
 -1.51245693 -1.02790903 -0.70116839 -0.09256256]
Variational Variances:  [1.73209277e-05 2.74881937e-04 6.26288496e-05
1.35262305e-04
 2.59626398e-04 2.91651751e-05 3.13156275e-04 8.16469848e-05
 1.64044796e-04 3.09218851e-04 4.57842399e-04 5.42849918e-04
 4.56487980e-04 5.57790855e-04 5.40750611e-04 5.49146522e-04]
Iteration 1900 lower bound -12736.842738713565; gradient mag: 2895.4816108335417
Variational Mean:  [ 0.81522013 -0.50977019 -0.82921578  0.57924816 -0.51700317
-4.20901415
 -3.59420039 -4.48862493  3.96262758 -3.62812289  1.93980379 -0.65084124
 -1.56174935 -1.11216994 -0.71212493 -0.11854489]
Variational Variances:  [1.48934638e-05 2.56170898e-04 5.24268325e-05
1.16289364e-04
 2.40990053e-04 2.53706101e-05 2.96779946e-04 6.92843331e-05
 1.45140377e-04 2.91509999e-04 4.12444920e-04 5.18261184e-04
 4.12086502e-04 5.32224794e-04 5.16508473e-04 4.97977132e-04]
Iteration 2000 lower bound -12641.863638967858; gradient mag: 2442.3237055590052
Variational Mean:  [ 0.81033864 -0.50921555 -0.84711125  0.58394417 -0.5162099
-4.18546131
 -3.63892695 -4.57003894  4.03064732 -3.67386225  1.9587069  -0.66978612
 -1.57386366 -1.2126317  -0.72909003 -0.13715481]
Variational Variances:  [1.29318504e-05 2.38565838e-04 4.42916303e-05
9.91914491e-05
 2.23708038e-04 2.23256013e-05 2.81395851e-04 5.93250664e-05
 1.27204046e-04 2.75239134e-04 3.72913840e-04 4.98518273e-04
 3.74085433e-04 5.10584535e-04 4.97400519e-04 4.53107332e-04]
Iteration 2100 lower bound -12533.461326346267; gradient mag: 1149.973193918241
Variational Mean:  [ 0.80746187 -0.51312    -0.87195214  0.59429565 -0.52027222
-4.17158823
 -3.6872741  -4.66037945  4.11233128 -3.72311557  1.96970749 -0.69475151
 -1.54461549 -1.33214712 -0.75178563 -0.14773047]
Variational Variances:  [1.13324187e-05 2.21768032e-04 3.79278888e-05
8.38685836e-05
 2.07391053e-04 1.98612241e-05 2.66414010e-04 5.15673325e-05
 1.10552993e-04 2.59813434e-04 3.38706943e-04 4.81192733e-04
 3.40913755e-04 4.91783945e-04 4.80565879e-04 4.13740641e-04]
Iteration 2200 lower bound -12393.757180152843; gradient mag: 1099.5401305736445
Variational Mean:  [ 0.80683405 -0.52234921 -0.90721453  0.61156341 -0.52965204
-4.16731276
 -3.74144023 -4.78970604  4.21168643 -3.77811257  1.97246505 -0.72790582
 -1.47006689 -1.47804222 -0.78277774 -0.15023716]
Variational Variances:  [1.00161873e-05 2.05422107e-04 3.30263583e-05
7.01250190e-05
 1.91780588e-04 1.78398563e-05 2.51367628e-04 4.56775121e-05
 9.50354086e-05 2.44655531e-04 3.08846270e-04 4.65324715e-04
 3.12512708e-04 4.74433367e-04 4.64752145e-04 3.78780334e-04]
Iteration 2300 lower bound -12196.62285730493; gradient mag: 8601.607048097887
Variational Mean:  [ 0.80785906 -0.53675243 -0.9660177   0.63734534 -0.54439463
-4.17392276
 -3.80439251 -5.00292127  4.3309711  -3.84164221  1.96612635 -0.77050204
 -1.35438495 -1.6561799  -0.82305882 -0.14437499]
Variational Variances:  [8.92537920e-06 1.89155093e-04 2.93040721e-05
5.78411613e-05
 1.76475682e-04 1.61719847e-05 2.35574870e-04 4.13175493e-05
 8.01231927e-05 2.29131316e-04 2.82639469e-04 4.49625448e-04
 2.87875090e-04 4.56967973e-04 4.48943622e-04 3.47973106e-04]
Iteration 2400 lower bound -11913.746793149705; gradient mag: 8024.477214356123
Variational Mean:  [ 0.81070748 -0.55007575 -1.05598782  0.66400755 -0.5583857
-4.18742748
 -3.8751584  -5.34369014  4.46034891 -3.91258728  1.9546785  -0.81120056
 -1.25605546 -1.84544744 -0.86245601 -0.13328563]
Variational Variances:  [8.01105846e-06 1.72849626e-04 2.65256560e-05
4.70077164e-05
```

```
 1.61363351e-04 1.47631780e-05 2.19348861e-04 3.81639208e-05
 6.62937141e-05 2.13288146e-04 2.59475492e-04 4.33363653e-04
 2.67188917e-04 4.38611980e-04 4.32626317e-04 3.20335527e-04]
Iteration 2500 lower bound -11684.555931217004; gradient mag: 10886.849541738618
Variational Mean:  [ 0.81046201 -0.5586385  -1.14695905  0.68824494 -0.56794254
-4.18674684
 -3.94211501 -5.73540925  4.57300521 -3.97967699  1.95784726 -0.83225255
 -1.24171215 -2.00033626 -0.88356624 -0.13638176]
Variational Variances:  [7.23922829e-06 1.57617718e-04 2.43454965e-05
3.80449158e-05
 1.47184308e-04 1.35724204e-05 2.03386222e-04 3.57622786e-05
 5.44861789e-05 1.97701865e-04 2.38853521e-04 4.17358583e-04
 2.49532654e-04 4.19294727e-04 4.16913740e-04 2.95442503e-04]
Iteration 2600 lower bound -11474.102618114006; gradient mag: 2902.643119648348
Variational Mean:  [ 0.80547826 -0.56018297 -1.21972812  0.7064779  -0.57030839
-4.15962677
 -4.0006311  -6.06487662  4.67042268 -4.03876527  1.98209353 -0.83561089
 -1.25409044 -2.11912884 -0.8879371  -0.1601296 ]
Variational Variances:  [6.56313882e-06 1.44358793e-04 2.24853354e-05
3.10527279e-05
 1.34730894e-04 1.25369962e-05 1.89619737e-04 3.36666938e-05
 4.50609393e-05 1.84029631e-04 2.20495525e-04 4.03854270e-04
 2.34529546e-04 4.00197818e-04 4.03048913e-04 2.73013938e-04]
Iteration 2700 lower bound -11353.592386958384; gradient mag: 6255.730008471211
Variational Mean:  [ 0.79842674 -0.55724975 -1.27284018  0.72366455 -0.56986374
-4.11998837
 -4.05028137 -6.31608174  4.75344457 -4.08924005  2.01601301 -0.82845143
 -1.26106864 -2.21434871 -0.88268242 -0.1933229 ]
Variational Variances:  [5.96062366e-06 1.33210723e-04 2.08697235e-05
2.56244686e-05
 1.24190505e-04 1.16064825e-05 1.78231990e-04 3.18799818e-05
 3.76264346e-05 1.72222554e-04 2.03819816e-04 3.92817550e-04
 2.21287820e-04 3.81673227e-04 3.91280750e-04 2.52885413e-04]
Iteration 2800 lower bound -11268.766604736407; gradient mag: 2147.608164414503
Variational Mean:  [ 0.79041208 -0.5523097  -1.31343977  0.73979177 -0.56753631
-4.08326664
 -4.09206794 -6.50976829  4.82735301 -4.13256854  2.04859276 -0.81763605
 -1.26079447 -2.29856906 -0.87325737 -0.22501414]
Variational Variances:  [5.42134902e-06 1.24130287e-04 1.94493714e-05
2.14144563e-05
 1.15381159e-04 1.07665291e-05 1.68989260e-04 3.03277649e-05
 3.17965335e-05 1.62718552e-04 1.88790014e-04 3.84471354e-04
 2.09396922e-04 3.63200706e-04 3.81906352e-04 2.34704997e-04]
Iteration 2900 lower bound -11199.015477970854; gradient mag: 4263.257728278334
Variational Mean:  [ 0.78402761 -0.54509195 -1.34651827  0.75517897 -0.56406503
-4.05346177
 -4.12756116 -6.66466265  4.8965541  -4.17008331  2.07543278 -0.80622992
 -1.25509228 -2.37769604 -0.86269325 -0.25105394]
```

```
Variational Variances:  [4.93799181e-06 1.16846558e-04 1.81913670e-05
1.81208581e-05
 1.08116812e-04 9.99309758e-06 1.61937063e-04 2.90072526e-05
 2.72884169e-05 1.54712717e-04 1.75190775e-04 3.78491630e-04
 1.98491966e-04 3.44901786e-04 3.74621070e-04 2.18155126e-04]
Iteration 3000 lower bound -11160.047825152069; gradient mag: 2232.567591780154
Variational Mean:  [ 0.77996523 -0.53884772 -1.37319492  0.76989135 -0.55895376
-4.02974979
 -4.15672499 -6.7921125   4.96176959 -4.20266981  2.09619483 -0.79628949
 -1.24957913 -2.45440147 -0.85238651 -0.27152554]
Variational Variances:  [4.50595630e-06 1.11116038e-04 1.70713545e-05
1.54928317e-05
 1.02131973e-04 9.30190805e-06 1.56778984e-04 2.78417946e-05
 2.35742184e-05 1.48622656e-04 1.62848094e-04 3.74566897e-04
 1.88448135e-04 3.26971636e-04 3.69112946e-04 2.03078313e-04]
```

4. (**Visualize the Posterior Predictive**) Visualize 100 samples $\mathbf{W}^s$ from your approximate posterior of $\mathbf{W}$ by ploting the neural network outputs with weight $\mathbf{W}^s$ plus a random noise $\epsilon \sim \mathcal{N}(0, 0.5^2)$ at 100 equally spaced x-values between -8 and 8:

   ```
   x_test = np.linspace(-8, 8, 100)
   y_test = nn.forward(sample, x_test.reshape((1, -1)))
   ```

   where `sample` is a sample from the approximate posterior of $\mathbf{W}$.

[17]:
```python
# sample from the variational posterior
posterior_sample_size = 100
posterior_samples = numpy.random.multivariate_normal(var_means, var_variance,␣
 ↪size=posterior_sample_size)

# sample from posterior predictive
x_test = numpy.linspace(-8, 8, 100)
y_tests, x_tests = [], []
for posterior_sample in posterior_samples:
    y_test = nn.forward(
        posterior_sample.reshape((1, -1)),
        x_test.reshape((1, -1))).squeeze()
    y_test += numpy.random.normal(loc=0, scale=0.5, size=y_test.shape)
    x_tests.append(x_test)
    y_tests.append(y_test)
x_tests = numpy.concatenate(x_tests)
y_tests = numpy.concatenate(y_tests)

# append MLE
y_mle = nn.forward(nn.weights, x_test.reshape((1, -1))).squeeze()
```

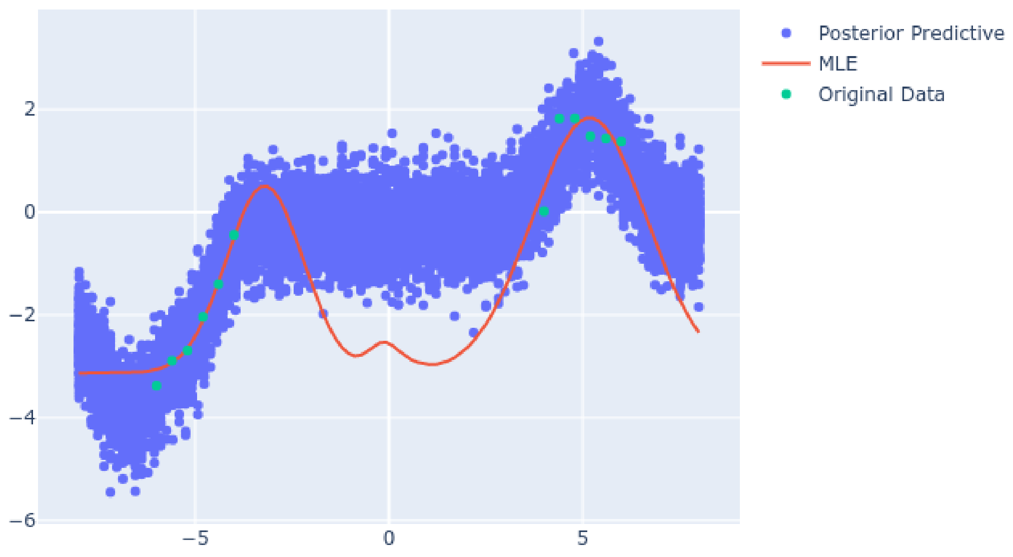
[18]:
```python
# plot posterior predictive

from IPython.display import Image
```

```
import plotly.graph_objects as go
import plotly.io as pio

plot_data = [
    go.Scatter(x=x_tests, y=y_tests, mode='markers', name='Posterior␣
 ↪Predictive'),
    go.Scatter(x=x_test, y=y_mle, name='MLE'),
    go.Scatter(x=xs, y=ys, mode='markers', name='Original Data')]
fig = go.Figure(data=plot_data)
# fig.show()
Image(pio.to_image(fig, format='png'))
```

[18]:



5. **(Computing the Fit)** Compute the posterior predictive log likelihood of the observed data under your model.

[21]:
```
from scipy.stats import norm


log_likelihoods = np.zeros(posterior_samples.shape[0])
for i, posterior_sample in enumerate(posterior_samples):
    y_mean = nn.forward(
        posterior_sample.reshape((1, -1)),
        xs.reshape((1, -1))).squeeze()
    log_like = norm.logpdf(ys, loc=y_mean, scale=0.5)
    log_likelihoods[i] = log_like.sum()
print('Posterior Predictive Log Likelihood: ', numpy.mean(log_likelihoods))
```

```
Posterior Predictive Log Likelihood:  -4.9322034067980045
```

6. **(Model Evaluation)** Compare the posterior predictive visualization and the posterior predictive log likelihood obtained from BBVI with the reparametrization trick to the ones you obtained in HW #7. Can you say whether or not your posterior approximation is good? How does approximating the posterior effect our estimation of epistemic and aleatoric uncertainty?

Compared against Homework 7's Hamiltonian Monte Carlo posterior samples, it appears that BBVI is a better model based on its more positive posterior predictive log likelihood (approximately -4.9 versus HMC's -8.72). In fairness, I'm not sure that this is a meaningful conclusion since the competition doesn't have comparable parameters. What I mean by this is that for HMC, I generated 10000 total samples, and took 50 steps per leapfrog step, whereas for BBVI, I generated 4000 samples and optimized for 3000 iterations, and I don't know how to compare these parameters against each other. Was equivalent computational work performed? I don't know.

Empirically, I can say that it appears neither model converged. HMC's trace plots were still trending, while BBVI's lower bound was still increasing and its gradients still large.

Subjectively, I think that HMC provided a better posterior approximation. I say this because although both HMC and BBVI have low aleatoric uncertainty near known points (x=-5, x=5), HMC generated a very wide posterior predictive distribution in the range $x \in [-3, 3]$, which accords with our intuition that we should have high epistemic uncertainty because we have no data in this range. In contrast, BBVI has much lower uncertainty in the range $x \in [-3, 3]$, which makes me concerned.

```
[ ]:
```