INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

AI61004: Statistical Foundations of AI and ML

Uncertainty in Neural Networks

Under the supervision of

Prof. Adway Mitra
Center of Excellence in Artificial Intelligence

Contents

§1 Introduction and Problem Statement

The concept of Deep Learning(DL) is quite prevalent today in the world of Machine Learning. Deep Learning tackles complex problems that were traditionally difficult to solve. However, DL models are often prone to overfitting which might make them less generalizable [3] as well as prone to different attacks (eg: membership inference attacks [6]). The less generalizability problem can lead to drastic outcomes in some cases like warning systems, finance-related, etc.

The traditional DL models are thus prone to uncertainties that need to be mitigated. There have been various approaches taken by researchers to understand and quantify the uncertainty in Neural Networks like **Bayesian Neural Networks**, **ensemble of neural networks**, and **test-time data augmentation** [1], etc. One of the most common ones is Bayesian Neural Networks(BNNs) which as the name suggests is based on Bayesian Statistics.

Supporting the development of learning algorithms in general, the Bayesian paradigm offers a rigorous framework for analyzing and training neural networks that account for uncertainty. The Bayesian statistics in contrast to the frequentist paradigm supports a probabilistic paradigm. The core idea of Bayesian statistics is as follows:

1. Contrary to what the frequentist paradigm assumes regarding the frequency limit of occurrence as the number of samples approaches infinity, probability represents the belief in the **occurrence of events**.

2. **Prior** beliefs influence **posterior** beliefs.

Theorem 1.1 (Bayes Theorem)

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} = \frac{P(D|H)P(H)}{\int_H P(D|H')P(H')\,dH'}$$

where, H is a hypothesis and D is some data, $P(H)$ is the prior, $P(D|H)$ is the likelihood and $P(D)$ is the evidence.

Thus, in addition to providing a reliable method for quantifying uncertainty in deep learning models, the Bayesian paradigm also offers a mathematical framework for comprehending numerous regularisation techniques and learning strategies that are already implemented in traditional deep learning.

Our work mainly focuses on how Bayesian Neural Networks address the problem of uncertainty in NNs and discusses the different methods to do Bayesian inference in BNNs like Variational Inference, Markov Chain Monte Carlo(MCMC), etc. It also explores the efficiency of these methods on different parameters like time, accuracy, etc.

§2 Bayesian Neural Networks

§2.1 Overview

Bayesian Neural Networks (BNNs) combine the power of neural networks with bayesian learning theory that uses the Bayes theorem to estimate the parameters. In this way we get a more wider picture on how the parameters should look. Denoting the input variables by x, the target variables by y and the set of parameters by θ we have,

$$p(\theta|x,y) = \frac{p(y|x,\theta)p(\theta)}{p(y|x)} \propto p(y|x,\theta)p(\theta)$$

The normalization constant $p(y|x)$ is the model evidence defined as,

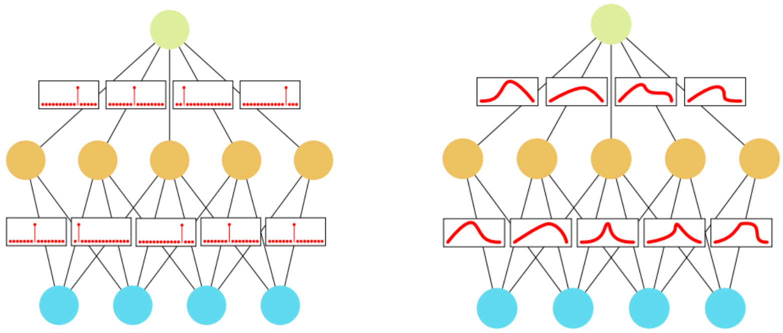
$$p(y|x) = \int_\theta p(y|x,\theta)p(\theta)d\theta$$

In general the basic goal of a traditional aritificial neural network is to estimate an arbitrary function Φ such that $y = \Phi(x)$. In the simplest architecture of feed-forward networks eachh layer applies a linear transformation on the values of the previous layer followed by applying a non-linear function on it. So we have the layers $l = \{l_0,\ l_1, ...\}$ and the set of functions $s = \{s_1,\ s_2, ...\}$ in each layer along with the weights $w = \{w_1,\ w_1, ...\}$ and biases $b = \{b_1,\ b_2, ...\}$ wired together by the equations,

$$l_0 = x$$

$$l_i = s_i(w_i l_{i-1} + b_i)$$

$$y = l_n$$



(a) Point estimate neural network

(b) stochastic neural network with a probability distribution over the weights.

Figure 1: Neural Networks and Bayesian Neural Networks Source: [3]

Here our parameter set $\theta = (w, b)$. Such models however suffer from the drawback of giving over-confident outputs on out-of-context data instead of just saying "I do not know". However this techniques can be mitigated by stochastic neural networks that incorporate stochastic elements into the traditional neural network architectures by setting,

$$\theta \sim p(\theta)$$

$$y = \Phi_\theta(x) + \epsilon$$

Here ϵ accounts for some amount of random noise on the output. A BNN in the simplest sense is a combination of the traditional neural networks with these stochastic elements. This involves choosing first a deep neural network architecture called the functional model followed by choosing a prior distribution ($p(\theta)$) on the model parameters and a distribution on the output variables ($p(y'|x', \theta)$) given the input variables and the model parameters. Thus we get,

$$p(\theta|D) \propto p(D_y|D_x, \theta)p(\theta)$$

Once the posterior distribution is known we can then marginalize with respect to the model parameters to obtain a distribution on the prediction y' on some test input x' as follows,

$$p(y'|x', D) = \int_\theta p(y'|x', \theta)p(\theta|D)d\theta$$

Observe that this essentially can be simulated through sampling θ' from $p(\theta|D)$ and then sample y' from $p(y|x', \theta')$, resulting in the following inference algorithm,

Algorithm 2.1 BNN Inference

```
Sample $\theta_i \sim p(\theta|D)$ for $i = 1$ to $n$
Sample $y_i \sim p(y|x', \theta_i)$ for $i = 1$ to $n$
**return** $Y = \{y_i\}_{i=1}^n$ and $\Theta = \{\theta_i\}_{i=1}^n$
```

Y is a set of samples from $p(y|x, D)$ and Θ a collection of samples from $p(\theta|D)$. In order to derive an estimator for the output y and to summarise the BNN's uncertainty, aggregates are often computed on those samples. The symbol for this estimator is $\hat{y}$.
Model averaging is the process that is typically utilised during **regression** to compile a BNN's predictions.

$$\hat{y} = \frac{1}{|\Theta|} \sum_{\theta_i \in \Theta} \Phi_{\theta_i}(\mathbf{x})$$

This method of group instruction is so popular that it is occasionally referred to as "ensembling". This is how the **covariance matrix** can be constructed to quantify uncertainty:

$$\Sigma_{y|x,D} = \frac{1}{|\Theta| - 1} \sum_{\theta_i \in \Theta} (\Phi_{\theta_i}(\mathbf{x}) - \hat{y})(\Phi_{\theta_i}(\mathbf{x}) - \hat{y})^T$$

The relative likelihood of each class will be provided by the average model prediction during **classification**, and this can be thought of as a measure of uncertainty:

$$\hat{p} = \frac{1}{|\Theta|} \sum_{\theta_i \in \Theta} \Phi_{\theta_i}(\mathbf{x})$$

The final prediction is taken as the most likely class:

$$\hat{y} = \underset{i}{\operatorname{argmax}}\ p_i \in \hat{p}$$

Let's discuss the training of the Bayesian Neural networks. There are two ways of training a BNN:

- **Active Learning:** data points in the training set with high epistemic uncertainty are scheduled to be labeled with higher priority.
- **Online Learning:** previous posteriors can be recycled as priors when new data become available to avoid the so-called problem of catastrophic forgetting

§2.2 Advantages of Bayesian Neural Networks

Uncertainty Quantification: Compared to classical neural networks, BNNs provide a more natural way to quantify uncertainty in deep learning, leading to better calibration and more consistent uncertainty estimates. They aid in data efficiency and better management of out-of-distribution points during forecasts by differentiating between different kinds of uncertainty.

Epistemic and Aleatoric Uncertainty: BNNs distinguish between two types of uncertainty: aleatoric uncertainty (inherent unpredictability in observed data) and epistemic uncertainty (connected to model uncertainty given restricted data). By making this distinction, data efficiency is improved and BNNs can learn from tiny datasets efficiently without overfitting.

Explicit Prior Representation: Although previous knowledge is implicit in all supervised learning algorithms, Bayesian techniques explicitly reflect it. Soft constraints, like regularisation, make it possible to incorporate past knowledge into BNNs and provide a methodical framework for doing so.

Interpretability and Analysis: The interpretation of different learning strategies is made possible by the Bayesian paradigm. Approximate Bayesian approaches are what standard deep learning techniques like ensembling and regularisation are. A methodical framework for creating novel regularisation and learning techniques is provided by BNNs.

Applications Across Diverse Fields: Applications for BNNs can be found in a wide range of fields, such as computer vision, aviation, civil engineering, astronomy, and medical. They are helpful in online learning by recycling prior knowledge as priors to avoid catastrophic forgetting, and in active learning by ranking data points for labelling based on uncertainty estimations.

All of these benefits put BNNs in a strong position for quantifying uncertainty, learning from small amounts of data efficiently, representing priors explicitly, interpreting learning processes, and having a wide range of applications.

§3 Setting up Bayesian Neural Networks

BNN design involves choosing of both functional model and stochastic model. Any model for point estimate networks can be used for the functional model. Below is description of how to setup the stochastic model.

§3.1 Probabilistic Graphical Models

Probabilistic Graphic Models (**PGM**) use graphs to represent the interdependence of multivariate stochastic variables and subsequently decompose their probability distributions. Here Bayesian Belief Networks (BBN) is described which is type of PGM.

Let variable v_i be a node in the acyclic graph. Then the joint probability distribution of all the variables in the graph is:

$$p(v_1, \ldots, v_n) = \prod_{i=1}^{n} p(v_i | parents(v_i))$$

Here the conditional probabilities depend on the context. It represents a data generation process where parents are sampled before the children.

In a PGM, the observed variables are treated as the data and the unobserved, also called latent variables are treated as the hypothesis. These can be represented by a directed graph. Finally, we need the joint distribution $p(v_{obs}, v_{latent})$, which can be calculated using Bayes theorem, for different inference algorithms.

§3.2 Defining the stochastic model of a BNN from a PGM

The data generation for a BNN with stochastic weights, which needs to perform regression, can be represented in the following way.

$$\theta \sim p(\theta) = \mathcal{N}(\mu, \Sigma)$$

$$y \sim p(y|x, \theta) = \mathcal{N}(\Phi_\theta(x), \Sigma)$$

Below is the representation of the same for classification.

$$\theta \sim p(\theta) = \mathcal{N}(\mu, \Sigma)$$

$$y \sim p(y|x, \theta) = Cat(\Phi_\theta(x))$$

Multiple points can be combined in the following way to get the probability for the entire training set.

$$p(D_y | D_x, \theta) = \prod_{(x,y) \in D} p(y|x, \theta)$$

§3.3 Setting the priors

For Bayesian regression, the standard way for parameter initialization is a zero mean and a diagonal covariance:

$$p(\theta) = \mathcal{N}(\mathbf{0}, \sigma \mathbf{I})$$

Normal distribution is generally preferred due to its special mathematical properties and simple representations while using log.

§4 Bayesian Inference

A Bayesian Neural Network or BNN has no learning phase because all it requires is the posterior of the result given the data points. To obtain this value, we need 3 components: $P(D|H)$, $P(H)$ which can be obtained very easily, and the evidence term, $\int_H P(D|H')P(H')\,dH'$, whose estimation requires a decent amount of effort. Sometimes, even if the evidence value is known, posterior sampling is difficult due to higher dimensionality. So, algorithms like Markov Chain Monte Carlo (MCMC) methods, or variational inference for approximating the posterior are used.

§4.1 Markov Chain Monte Carlo

It primarily works by generating a sequence of samples known as Markov Chain, where each sample depends on the previous sample. However, we need all the samples to be independent. So, to ensure that there is no correlation among the samples, we ignore some terms at the beginning, known as the burn-in phase, and then sample terms at regular intervals. This provides a good sample although it is quite slow.

Among various MCMC algorithms that exist, Metropolis Hastings [2] is considered the best candidate for usage in BNNs. It works because it does not require sampling from an exact probability distribution, rather sampling can be done from a function proportional to the required probability distribution, which is the formula of posterior without the evidence term in this case.

The working of the Metropolis Hastings algorithm is as follows:

Algorithm 4.1 Metropolis Hastings algorithm

```
Draw $\theta_0 \sim \textit{Initial Probability Distribution}$
while $n = 0$ to $N$ do
    Draw $\theta' \sim g(\theta'|\theta_n)$
    $A(\theta', \theta_n) = min(1, \frac{f(\theta')g(\theta_n|\theta')}{f(\theta_n)g(\theta'|\theta_n)})$
    Draw $k \sim \mathcal{U}(0, 1)$
    if $k < A(\theta', \theta_n)$ then
        accept
    else
        reject
    end if
end while
```

- An initial guess θ_0 is made from a initial probability distribution.
- The new candidate points θ' are selected around previous theta using a proposal distribution $g(\theta|\theta')$.
- Next, the acceptance ratio $A(\theta', \theta_n)$ is calculated. This is followed by sampling a value k from uniform distribution $\mathcal{U}(0, 1)$
- if $k < A(\theta', \theta_n)$ then accept the point, else reject it.

If the new point is more likely than the previous point according to the target distribution, it is accepted. Otherwise, it may be accepted or rejected with some probability. The proposal distribution is generally a normal distribution with the previous point as the mean.

It is also needed that the spread of g is not very high, otherwise it will lead to too much rejection. Also it cannot be very low, otherwise there will be correlation. To mitigate this problem, **Hamiltonian Monte Carlo**, a form of Metropolis Hastings algorithm, is used. It balances the spread of g and also has a short burn-in period and hence is very fast. **No-U-Turn** sampler or **NUTS** is an example of this and is used in our implementation of **MCMC**.

§4.2 Variational Inference

Variational Inference doesn't sample from the exact posterior. Rather, it tries to build a variational distribution $q_\phi(H)$ that is as similar to the posterior distribution as possible. KL divergence [4] is used to measure the similarity of these two distributions.
For Bayesian Inference, this translates to:

$$D_{KL}(q_\phi||P) = \int_H q_\phi(H') log \frac{q_\phi(H')}{P(H'|D)} dH'$$

For computing the above described KL-Divergence loss, we would also need to compute $P(H|D)$. To avoid this **ELBO** or *evidence lower bound* is considered. The expression is as follows:

Theorem 4.2 (**ELBO** - Evidence Lower Bound)

$$\int_H q_\phi(H') log \frac{P(H', D)}{q_\phi(H')} dH' = log(P(D)) - D_{KL}(q_\phi||P)$$

Proof. We can prove the expression using a simple manipulation as follows:

$$\int_H q_\phi(H') log \frac{P(H', D)}{q_\phi(H')} dH' = \int_H q_\phi(H') log \frac{P(H'|D)P(D)}{q_\phi(H')} dH'$$

$$Or, \int_H q_\phi(H') log \frac{P(H', D)}{q_\phi(H')} dH' = \int_H q_\phi(H')(log(P(D)) - log \frac{q_\phi(H')}{P(H'|D)}) dH'$$

$$Or, \int_H q_\phi(H') log \frac{P(H', D)}{q_\phi(H')} dH' = log(P(D)) \int_H q_\phi(H') dH' - \int_H q_\phi(H') log \frac{q_\phi(H')}{P(H'|D)} dH'$$

This gives, $log(P(D)) - D_{KL}(q_\phi||P)$ since, $\int_H q_\phi(H') dH' = 1$ □

Since $log(P(D))$ depends only on prior, minimizing KL-Divergence loss is equivalent to maximizing ELBO.

For this purpose, *stochastic variational inference* or **SVI** is used. It is a form of stochastic gradient descent applied to variational inference. Since the convergence rate is low, it takes a large no. of iterations to give a satisfactory result. It works very well with minibatches and hence scales even for large datasets.

§4.3 Bayes by Backpropagation

Although variational inference works well, the stochasticity does not work very well with the back propagation at internal nodes. To alleviate this, Bayes-by-backprop is used, which is the combination of SVI with reparameterization trick.

It involves a random variable which is a nonvariational source of noise. The involved variable depends on this as a deterministic tranformation. So, although the random variable changes after each iteration, it still remains effectively unchanged w.r.t. other parameters because it is independent. Because of this, backpropagation works as usual. Hence, it can be used similar to a non-stochastic neural network. Additionally, for faster processing, known optimization techniques can also be incorporated.

Due to its highly complex nature, we have not implemented this.

§5 Summary of Bayesian Neural Networks

Bayesian Neural Networks (BNNs) represent a paradigm shift from traditional neural networks by integrating Bayesian inference techniques. They enable the quantification of uncertainty in predictions, a crucial aspect absent in standard neural networks.

Unlike traditional networks that provide point estimates, BNNs offer probabilistic outputs that include uncertainty measures. This aspect aids decision-making processes, especially in scenarios where understanding uncertainty is critical.

The table below explicitly draws a comparative analysis among three different types of Bayesian Neural Network (BNN) approaches namely, Variational Inference, Sampling Approaches and Laplace Approximation.

	Variational Inference	**Sampling Approaches**
Description	By optimising over a family of tractable distributions, variational inference algorithms approximate the (in general intractable) posterior distribution. Achieved via reducing the KL divergence.	The target random variable from which realisations can be sampled is represented. These techniques are based on Markov Chain Monte Carlo and its extensions.
Analytic Expression	Yes	No
Applied to Pre-trained Networks(Yes/No)	No	No
Deterministic or not	Yes	No
Unbaised or not	No	Yes
Optimality	Local optimum	Hard to mix between nodes
Computational effort at training time	Medium - Regularisation may cause convergence to slow down. Extra variables for the representation of uncertainty.	High-M forward passes with sampled parameters; otherwise, it becomes unmanageable.

Practical implementation of BNN can be done by several techniques such as Monte Carlo dropout, variational inference, and Markov Chain Monte Carlo methods are elucidated. Monte Carlo dropout involves applying dropout during test time to generate multiple predictions, enabling uncertainty estimation. Variational inference approximates complex posterior distributions with simpler ones, enhancing scalability. Meanwhile, Markov Chain Monte Carlo methods sample from the posterior distribution to estimate parameter uncertainty.

Bayesian Neural Networks leverage diverse inference methods—MCMC and VI. The forthcoming table outlines their unique principles, benefits, limitations, and specific applications, essential for understanding uncertainty in BNNs.

Markov Chain Monte Carlo (MCMC):

	Benefits	**Limitations**	**Use-Cases**
MCMC	Directly samples the posterior	Requires to store a very large number of samples	Small and average models
Classic methods(HMC, NUTS)	State of the art samplers limit autocorrelation between samples	Do not scale well to large models	Small and critical models
SGLD and derivates	Provide a well behaved Markov Chain with minibatches	Focus on a single mode of the posterior	Models with larger datasets
Warm restarts	Help a MCMC method explore different modes of the posterior	Requires a new burn-in sequence for each restart	Combined with a MCMC sampler

Variational Inference:

	Benefits	**Limitations**	**Use-Cases**
Variational inference	DThe variational distribution is easy to sample	Is an approximation	Large Scale models
Bayes by backprop	SFit any parametric distribution as posterior	Noisy gradient descent	Large scale models
Laplace approximation	By analyzing standard SGD get a BNN from a MAP	Focus on a single mode of the posterior	Unimodals large scale models
Deep ensembles	Help focusing on different modes of the posterior	Cannot detect local uncertainty if used alone	Multimodals models and combined with other VI methods
Monte Carlo-Dropout	Can transform a model using dropout into a BNN	Lack expressive power	Dropout based models

§6 Our Work

We conducted 4 experiments with different publicly available dataset. These are as follows:

§6.1 Experiment 1[Notebook link]

In this experiment, we trained a shallow(2 hidden layer) bayesian neural network with a synthetically generated dataset and tested it on the testing dataset.

§6.1.1 Tools and Libraries

- **Numpy:** It is a library providing comprehensive mathematical functions and tools. Used for different utilities like `numpy.ndarray` etc.
- **Matplotlib:** It is a plotting library in python. Used for plotting plots like scatter plots, line curves etc.
- **PyTorch:** PyTorch is a machine learning framework. It is used for specific purposes like defining layers in neural networks (`nn.Linear`), tensors etc.
- **Pyro:** It is a probabilistic programming language. It is used for various purposes like bayesian inference, sampling from distributions etc.

§6.1.2 Methodology

We first created a neural network with two hidden layers whose weights(θ) and bias has been initialised with a normal distribution as follows:

$$p(\theta) \sim \mathcal{N}(\mathbf{0}, 5 \cdot I)$$

For training the data the likelihood distribution is calculated as:

$$p(y_i|x_i, \theta) \sim \mathcal{N}(NN_\theta(x_i), \sigma^2),\ \sigma^2 \sim \Gamma(0.5, 1)$$

where, y_i is the actual output, x_i is the input, NN_θ is the neural network defined over parameter θ and σ is the standard deviation of the normal distribution. The code snippet is as follows:

```python
class BNN(PyroModule):
  def __init__(self, inp_dim=1, out_dim=1, hid_dim=5, pri_scl=5.,
                                                   activ="tanh"):
    super().__init__()
    self.activation=nn.Tanh()
    if(activ=="relu"):
      self.activation = nn.ReLU()
    self.layer1 = PyroModule[nn.Linear](inp_dim, hid_dim) #first hidden layer
    self.layer2 = PyroModule[nn.Linear](hid_dim, hid_dim) #second hidden layer
    self.layer3 = PyroModule[nn.Linear](hid_dim, out_dim) #output layer

    #initialising layer configs
    self.layer1.weight = PyroSample(dist.Normal(0., pri_scl).expand([hid_dim,
                                                     inp_dim]).to_event(2))
    self.layer1.bias = PyroSample(dist.Normal(0.,pri_scl)
    .expand([hid_dim]).to_event(1))

    self.layer2.weight = PyroSample(dist.Normal(0., pri_scl)
    .expand([hid_dim, hid_dim]).to_event(2))
    self.layer2.bias = PyroSample(dist.Normal(0., pri_scl)
    .expand([hid_dim]).to_event(1))

    self.layer3.weight = PyroSample(dist.Normal(0., pri_scl)
    .expand([out_dim, hid_dim]).to_event(2))
    self.layer3.bias = PyroSample(dist.Normal(0., pri_scl)
      .expand([out_dim]).to_event(1))

  def forward(self, x, y=None):
    x = x.reshape(-1, 1)
    x = self.activation(self.layer1(x))
    x = self.activation(self.layer2(x))
    mu = self.layer3(x).squeeze()
    sigma = pyro.sample("sigma", dist.Gamma(0.5, 1)) # for noise

    # for sampling
    with pyro.plate("data", x.shape[0]):
      obs = pyro.sample("obs", dist.Normal(mu, sigma * sigma), obs=y)

    return mu
```

The data for training is generated synthetically using the following function:

$$y = x + sin(\frac{\pi}{2}x) + cos(\pi x)$$

Now, it can be observed that,

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int_{\theta} p(\mathbf{y}|\mathbf{x}, \theta')P(\theta'|\mathcal{D})\, d\theta' = \mathbf{E}_{\theta \sim p(\theta|\mathcal{D})}[p(\mathbf{y}|\mathbf{x}, \theta)]$$

Now, we can approximate the $\mathbf{E}_{\theta \sim p(\theta|\mathcal{D})}[p(\mathbf{y}|\mathbf{x}, \theta)]$ as follows:

$$\mathbf{E}_{\theta \sim p(\theta|\mathcal{D})}[p(\mathbf{y}|\mathbf{x}, \theta)] \approx \frac{1}{N}\sum_{i=1}^{N} p(\mathbf{y}|\mathbf{x}, \theta_i)$$

θ_i is drawn from $p(\theta|\mathcal{D})$ using MCMC. Thus, we used the NUTS kernel and MCMC to train the neural network with number of samples as 100. The code snippet for training is as follows:

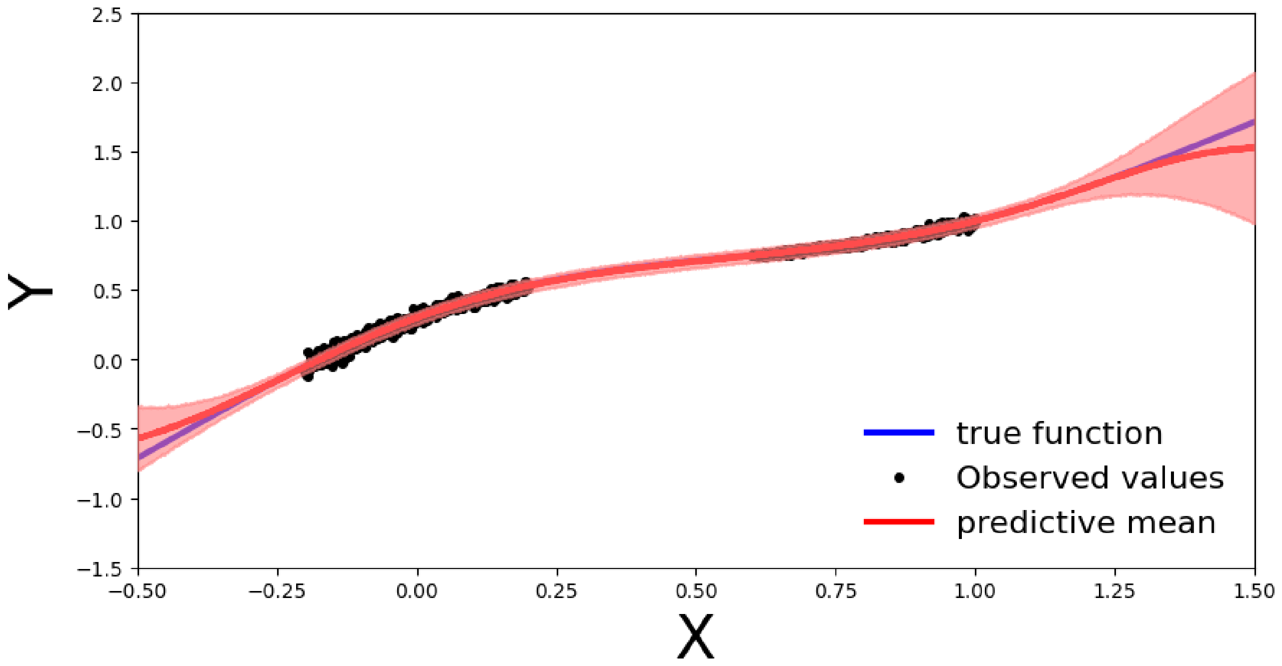
```
model = BNN()
pyro.set_rng_seed(int(time.time()))
nuts_kernel = NUTS(model, jit_compile=True)
mcmc = MCMC(nuts_kernel, num_samples=100)
x_train = torch.from_numpy(x_obs).float()
y_train = torch.from_numpy(y_obs).float()
mcmc.run(x_train, y_train)
```

Remark 6.1. The `jit_compile` flag is for "just in time" compile option. It makes the NUTS kernel run faster.

§6.1.3 Results

The model is trained using the data generated. It took 8 minutes and 14 seconds to train with acceptance probability = 0.957

We ran the model trained on a test dataset and the output is plotted. The output is as follows:



§6.2 Experiment 2[Notebook link]

In this experiment, we trained a deep Bayesian Neural Network using three datasets, one synthetically generated and two publicly available datasets.

§6.2.1 Tools and Libraries

Same as experiment 1

§6.2.2 Methodology

We created a deep bayesian neural network similar to the BNN created in experiment 1. All the initial parameters are same. The code snippet is as follows:

```
class BNN(PyroModule):
  def __init__(self, inp_dim=1, n_layers=2, out_dim=1, hid_dim=10, pri_scl=5.,
                                               activ="tanh", reshape = True):
    super().__init__()
    self.activation=nn.Tanh()
    if(activ=="relu"):
      self.activation = nn.ReLU()

    self.reshape = reshape

    self.l_sz_list = []
    self.l_sz_list.append(inp_dim)
    for i in range(n_layers):
      self.l_sz_list.append(hid_dim)

    self.l_sz_list.append(out_dim)
    # layer info
    self.layer_configs = []
    for i in range(1, len(self.l_sz_list)):
      self.layer_configs.append(PyroModule[nn.Linear](self.l_sz_list[i-1],
                                                   self.l_sz_list[i]))

    self.layers= PyroModule[torch.nn.ModuleList](self.layer_configs)

    for index, layer in enumerate(self.layers):
```

```python
      layer.weight = PyroSample(dist.Normal(0., pri_scl*np.sqrt(2 / self.l_sz_list[index])).expand(
      [self.l_sz_list[index + 1], self.l_sz_list[index]]).to_event(2))
      layer.bias = PyroSample(dist.Normal(0.,pri_scl)
      .expand([self.l_sz_list[index + 1]]).to_event(1))

  def forward(self, x, y=None):
    if self.reshape is True:
      x = x.reshape(-1, 1)
    x = self.activation(self.layers[0](x))
    for i in range(1, len(self.layers)-1):
      x = self.activation(self.layers[i](x))

    mu = self.layers[-1](x).squeeze()
    sigma = pyro.sample("sigma", dist.Gamma(.5, 1)) # for noise

    # for sampling
    with pyro.plate("data", x.shape[0]):  # pyro plate is a context manager for handling collection of data points
      obs = pyro.sample("obs", dist.Normal(mu, sigma * sigma), obs=y)

    return mu
```

The BNN is trained using three datasets these are:

- Synthetically created dataset from the function:

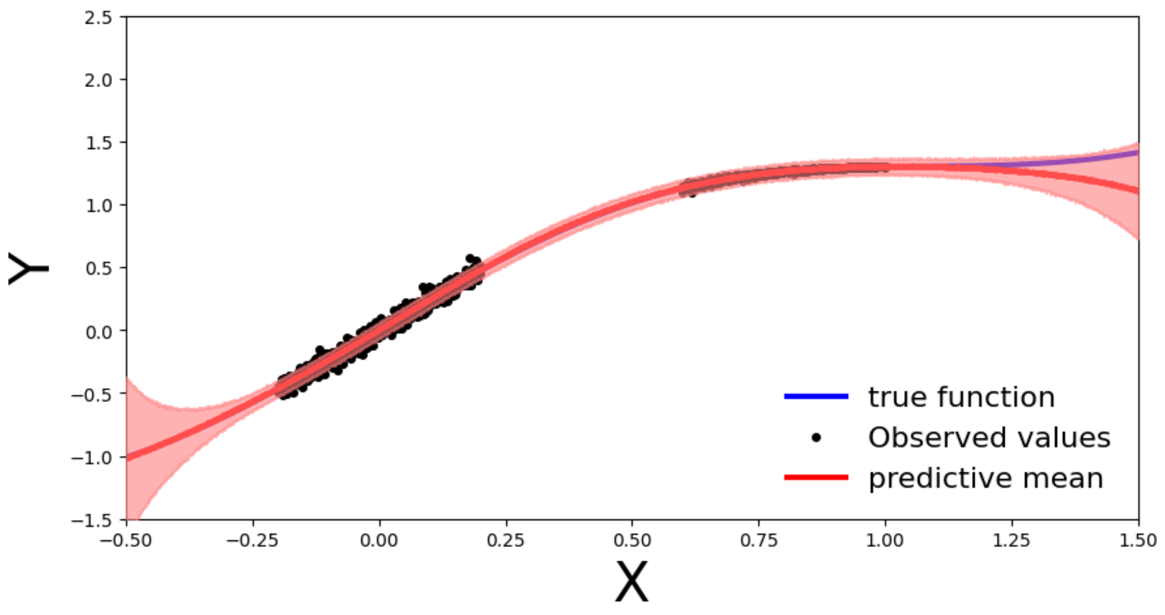
$$y = x + sin(\frac{\pi}{2}x) + sin(\pi x)$$

- Publicly available dataset(from kaggle):
  - Experience - Salary Dataset [link]
    - **Input:** Experience(in months)
    - **Output:** Salary(in thousands of dollars)
  - Admission Prediction dataset [link]
    - **Input:** Collection of features (GRE Score, TOEFL Score, University Rating, SOP, LOR, CGPA, Research)
    - **Output:** Chance of Admit (percentage)

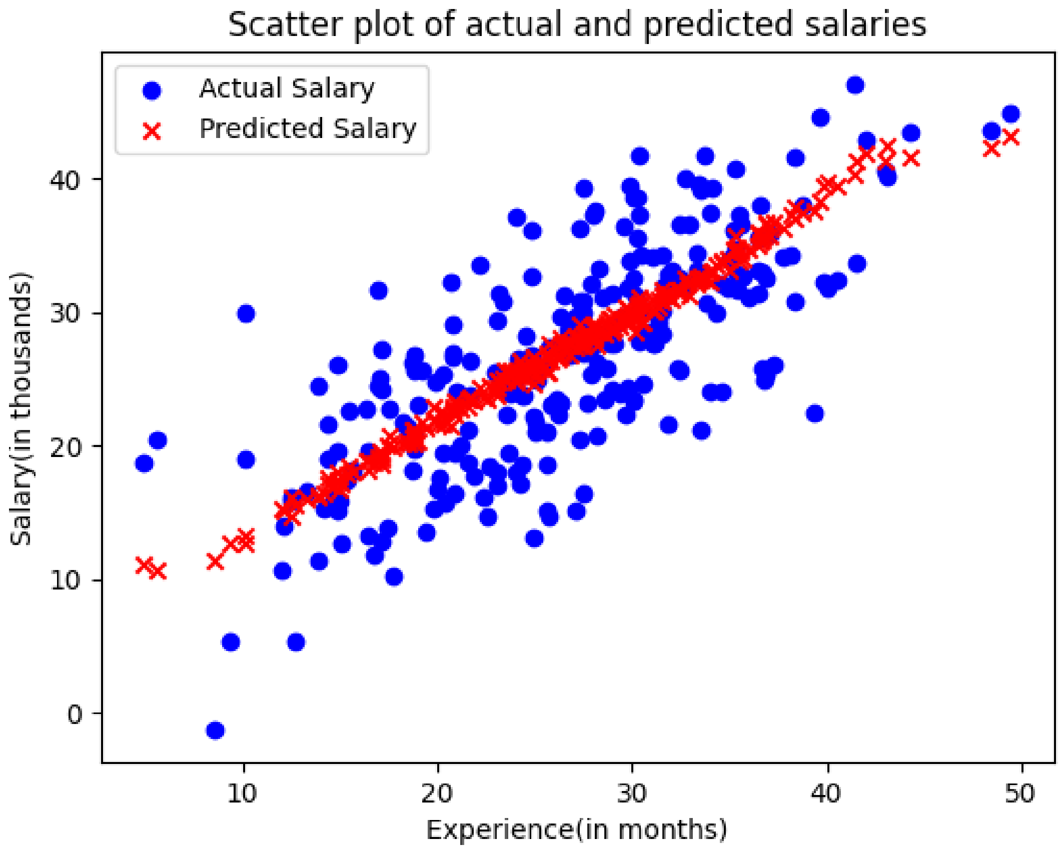
§6.2.3 Results

Results on different dataset are as follows:

1. **Synthetically created dataset:** The model training time was 17 minutes 20 seconds with an acceptance probability of 0.959.
   We ran the model on the test dataset and found the following results:



2. **Experience-Salary Dataset:** We trained the model on 75% of the data and tested it on 25% of the data (using test-train split). It took 19 minutes and 1 second to train the data with an acceptance probability of 0.85.
We ran the model on the test dataset and it gave the mean squared error as 31.11. The plot generated is as follows:



3. **Admission Prediction dataset:** Similar to the previous case, we trained on 75% of the data and it took 17 minutes and 29 seconds with an acceptance probability of 0.796.
We ran it on the test dataset and it gave the mean squared error to be 0.033.

§6.3 Experiment 3[Notebook link]

In this experiment, we trained a Bayesian Neural Network using three datasets, one synthetically generated and two publicly available datasets, using variational inference.

§6.3.1 Tools and Libraries

Same as experiment 1

§6.3.2 Methodology

We created a deep bayesian neural network similar to the BNN created in experiment 2. All the initial parameters are same. The BNN is trained using three datasets as mentioned above (similar to experiment 2).

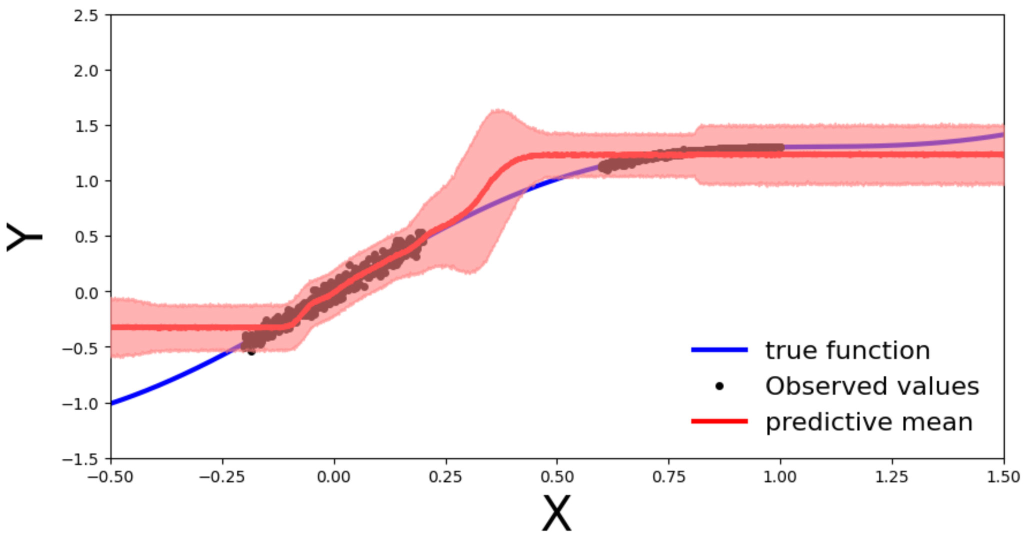
The only difference is that here data is processed using stochastic variational inference or SVI. AutoDiagonalNormal is used as the guide and Adam optimiser is used for optimisation.

```
x_train = torch.from_numpy(x_train.to_numpy()).float()
y_train = torch.from_numpy(y_train.to_numpy()).float()

model1 = BNN()
pyro.set_rng_seed(int(time.time()))
mean_field_guide1 = AutoDiagonalNormal(model1)
optimizer1 = pyro.optim.Adam({"lr": 0.01})

svi = SVI(model1, mean_field_guide1, optimizer1, loss=Trace_ELBO())
pyro.clear_param_store()

num_epochs = 10000
progress_bar = trange(num_epochs)

for epoch in progress_bar:
    loss = svi.step(x_train, y_train)
    progress_bar.set_postfix(loss=f"{loss / x_train.shape[0]:.3f}")
```

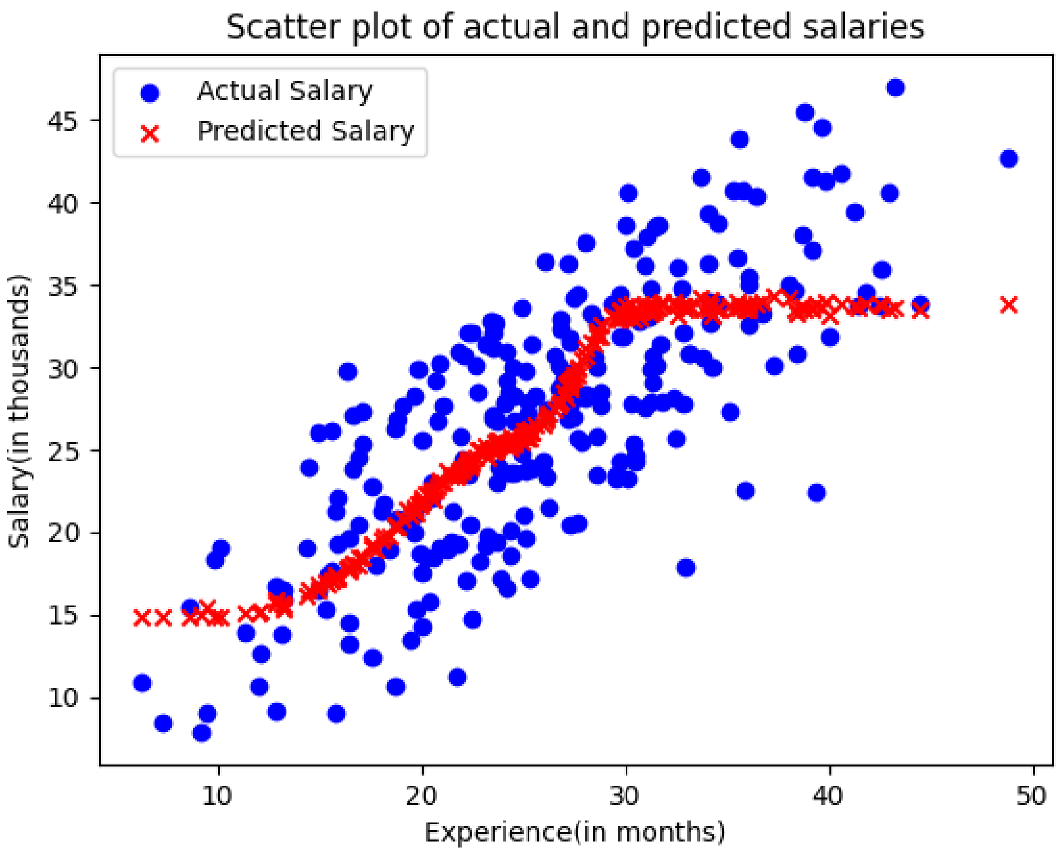
§6.3.3 Results

Results on different dataset is as follows:

1. **Synthetically created dataset:** The model training time was 8 minutes 6 seconds. We ran the model on the test dataset and found the following results:



2. **Experience-Salary Dataset:** We trained the model on 75% of the data and tested it on 25% of the data (using test-train split). It took 3 minutes and 40 seconds to train the data.
   We ran the model on the test dataset and it gave the mean squared error as 28.77. The plot generated is as follows:



3. **Admission Prediction dataset:** Similar to the previous case, we trained on 75% of the data and it took 3 minutes and 14 seconds.
   We ran it on the test dataset and it gave the mean squared error to be 0.021.

§6.4 Experiment 4[Notebook link]

In this experiment, we trained a basic Bayesian Neural Network using iris dataset and compared the accuracy over various epoch counts.

§6.4.1 Tools and Libraries

- **Numpy:** It is a library providing comprehensive mathematical functions and tools. Used for different utilities like `numpy.array` etc.
- **Matplotlib:** It is a plotting library in python. Used for plotting plots like scatter plots, line curves etc.
- **PyTorch:** PyTorch is a machine learning framework. It is used for specific purposes like defining layers in neural networks (`nn.Linear`), tensors etc.
- **Torchbnn** [5]**:** It is a derived library from torch which has specific utilities for Bayesian neural networks.
- **SKLearn:** It is used here for importing the dataset used (iris in this case).

§6.4.2 Methodology

We used the torchbnn library to build a custom neural network having Bayes layers. The initial μ value is 0 and initial σ value is 0.1. Below is the code for the same.

```
class BayesianNet(T.nn.Module):
  def __init__(self):
    super(BayesianNet, self).__init__()
    self.hid1 = bnn.BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=4,
                                      out_features=100)
    self.oupt = bnn.BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=100,
                                      out_features=3)

  def forward(self, x):
    z = T.relu(self.hid1(x))
    z = self.oupt(z)
    return z
```

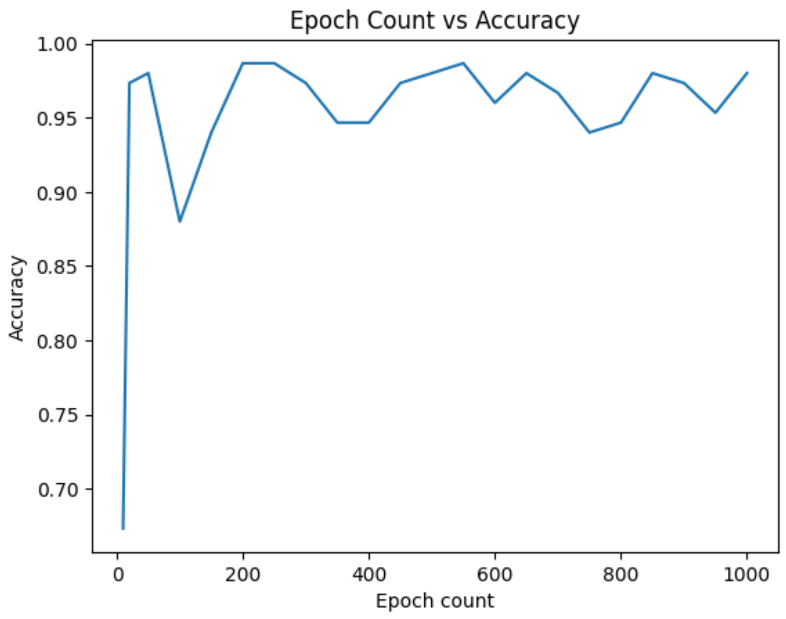
Various losses are also used here like CrossEntropyLoss and KLDivergence Loss which are used for backpropagation and parameter updation.

```
ce_loss = T.nn.CrossEntropyLoss()
kl_loss = bnn.BKLLoss(reduction='mean', last_layer_only=False)
optimizer = T.optim.Adam(net.parameters(), lr=0.01)
```

```
X = batch['predictors']
Y = batch['species']
optimizer.zero_grad()
oupt = net(X)

cel = ce_loss(oupt, Y)
kll = kl_loss(net)
tot_loss = cel + (0.10 * kll)

epoch_loss += tot_loss.item()
tot_loss.backward()
optimizer.step()
```

§6.4.3 Results

Here are the final results where accuracy is plotted against no. of epochs.



We can clearly see that the accuracy is the highest for number of epochs = 250.

§7 Conclusion and Future work

From our work, we conclude that BNNs are able to quantify the uncertainty found in point estimate neural networks. The error probabilities are also not significantly high. One remarkable point is as follows:

> **Remark 7.1.** The time to train the neural network with MCMC (NUTS kernel) is higher than that of stochastic variational inference(SVI). This is because of the following reasons:
>
> - BURN-IN time is involved in MCMC(such that the markov chain is properly mixed)
> - Also MCMC samples points at regular intervals to make them *iids*. This accounts for the higher time difference.

In our experiments we have considered only MCMC and SVI, however in a more practical and scalablity point of view, both fail. Here comes solutions like Bayes by Backprop which are more practical. Another solution is to train in mini-batches. For this, we need Stochastic Gradient Langevin Dynamics(SGLD). Thus our future work is as follows:

- We will focus on different priors like gamma, laplacian, cauchy distributions, etc.
- We will experiment with Bayes by backprop as well as Stochastic Gradient Langevin Dynamics(SGLD).

References

[1] Jakob Gawlikowski et al. *A Survey of Uncertainty in Deep Neural Networks*. 2022. arXiv: 2107.03342 [cs.LG].

[2] W. K. Hastings. "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". In: *Biometrika* 57.1 (1970), pp. 97–109. ISSN: 00063444. URL: http://www.jstor.org/stable/2334940 (visited on 11/29/2023).

[3] Laurent Valentin Jospin et al. "Hands-On Bayesian Neural Networks—A Tutorial for Deep Learning Users". In: *IEEE Computational Intelligence Magazine* 17.2 (2022), pp. 29–48. DOI: 10.1109/MCI.2022.3155327.

[4] Solomon Kullback and Richard A Leibler. "On information and sufficiency". In: *The annals of mathematical statistics* 22.1 (1951), pp. 79–86.

[5] Sungyoon Lee, Hoki Kim, and Jaewook Lee. "Graddiv: Adversarial robustness of randomized neural networks via gradient diversity regularization". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

[6] Reza Shokri et al. "Membership Inference Attacks Against Machine Learning Models". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 3–18. DOI: 10.1109/SP.2017.41.