# DT8122 Probabilistic AI Assignment

Shweta Tiwari

July 2019

## 1 Introduction

In applied machine learning, deep learning tools have gained huge attention. However, such tools are not good enough to capture model uncertainty in classification and regression task. Bayesian models on the other hand offer a mathematical grounded framework to reason about model uncertainty. Standard deep learning tools for regression and classification do not capture model uncertainty.

Model uncertainty arises due to imperfections and idealizations made in physical model formulations for load and resistance, as well as in the choices of probability distribution types for the representation of uncertainties. Often, it is not possible with very few expectations to make highly accurate predictions about the magnitude of the typical structure responses due to loadings, even when governing input quantities are known. This additional source of uncertainty is termed model uncertainty, and occurs as a result of simplifying assumptions, unknown boundary conditions, and the unknown effects and interactions of other variables that are not included.

## 2 Evaluation metrics and methods

### 2.1 Prediction interval

In statistical analysis and machine learning, especially with respect to the regression analysis, a typical confidence interval is measured by prediction interval(PI) [1]. It is a range of values that predicts the value of a new observation based on existing model. In other words, prediction interval is where you expect a future value to fall in.

The PI is defined as in equation 1, and a python snippet to calculate it is also shown in Figure 1.

$$P\left(\hat{y}_{Li} \leq y_i \leq \hat{y}_{Ui}\right) \geq \gamma, \tag{1}$$

where $[\hat{y}_{Li}, \hat{y}_{Ui}]$ is called a level $\gamma$ prediction interval.

[1] `https://en.wikipedia.org/wiki/Prediction_interval`

```python
import scipy
def pred_interval(y_true, y_pred, quantile):  # z = significance level of 95%, which is the Gaussian critical valu
    z = scipy.stats.norm.ppf(quantile)
    sum_err = np.sum((y_true - y_pred) ** 2)
    stdev = np.sqrt(1/(len(y_true) - 2) * sum_err)
    interval = z * stdev
    return interval
```

Figure 1: Method to calculate prediction interval

## 2.2 Prediction Interval Coverage Probability

The prediction interval coverage probability (PICP) [3] can be calculated using the prediction interval. Consider a vector, $k$, of length $n$ that represents whether each data point has been captured by the estimated PIs, with each element $k_i \in \{0, 1\}$ is given by (a python snippet to calculate the PICP is shown in Figure 2),

$$k_i \in \begin{cases} 1, & \text{if } \hat{y}_{Li} \leq y_i \leq \hat{y}_{Ui} \\ 0, & \text{otherwise} \end{cases}$$

$$c = \sum_{i=1}^{n} k_i, \tag{2}$$

$$PICP = \frac{c}{n}. \tag{3}$$

```python
def picp(y_true, y_lower, y_upper):
    x = []
    for y, lo, up in zip(y_true, y_lower, y_upper):
        condition = ((lo < y) & (y < up))
        if condition:
            value = 1
        else:
            value = 0
        x.append(value)
    result = np.sum(x) / float(len(x))
    return result
```

Figure 2: Prediction interval coverage probability(PICP)

## 2.3 Mean Prediction Interval Width (MPIW)

MPIW [3] is calculated as following in equation 4 and an implementation is shown in 3:

$$MPIW = \frac{1}{n} \sum_{i=1}^{n} \hat{y}_{Ui} - \hat{y}_{Li}. \tag{4}$$

## 2.4 Root Mean Square Erorr (RMSE)

RMSE is calculated as shown in Figure 4:

```python
def mpiw(y_lower, y_upper):
    result = (y_upper - y_lower).mean()
    return result
```

Figure 3: Mean Prediction Interval Width (MPIW)

```python
RMSE = np.sqrt(np.mean((y_pred - y_test.values)**2))
```

Figure 4: Root Mean Square Error (RMSE)

# 3 Models

## 3.1 Dropout as a Bayesian Approximation

The network uncertainty is a quantitative metric revealing the network confidence in its prediction. Dropout is a well-established procedure to regularize a neural network and limit overfitting. [2] proposes a simple approach to quantify the neural network uncertainty, employing dropout during both training and testing

A neural network, with dropout enabled during testing, generates a different output every forward pass for the same input. The paper mathematically shows that these multiple passes are equivalent to Monte-Carlo sampling. Thus, the first and second moment (mean and variance) provide the network's output and uncertainty, respectively.

High variance/standard deviation indicates high network uncertainty and vice versa. A quantitative uncertainty measure is valuable especially if further decisions are based on the network output. The theoretical framework employs a dropout layer before every weight layer as a Bayesian inference approximation. The dropout rate is a hyper-parameter that needs to be tuned. A small dropout rate eliminates the Monte-Carlo sampling utility. A big dropout rate can lead to divergence or at least require more iterations to converge.

Structure of neural network is shown in Figure 5. Main training loop for the dropout network is shown in Figure 6, and the prediction generator is shown in Figure 7

## 3.2 Bayes by Backprop

The goal of classic multi layer perceptron(MLP) was to find an optimal point estimation for the weights. Simple feedforward neural networks are prone to overfitting. Often, these networks are incapable to assess the uncertainty in the training data and so make over confident decisions about the correct class, prediction or action when applied to supervised or reinforcement learning problems. This drawback motivates the application of Bayesian learning to neural networks, introducing probability distributions over the weights.

Following are the motivations for modeling uncertainty of weights:

```
class My_Network(nn.Module):
    def __init__(self, hidden_layers, dropout_rate, activation_function):
        super(My_Network, self).__init__()
        self.model = nn.Sequential()
        #Adds a child module to the current module. The module can be accessed as an attribute using the given name
        self.model.add_module('input', nn.Linear(9, hidden_layers[0]))
        self.model.add_module('relu0', nn.ReLU())
        for i in range(len(hidden_layers)-1):
            self.model.add_module('dropout'+str(i+1), nn.Dropout(p=dropout_rate))# pytorch's inbuilt droupout
            self.model.add_module('hidden'+str(i+1), nn.Linear(hidden_layers[i], hidden_layers[i+1]))
            self.model.add_module('relu'+str(i+1), nn.ReLU())
        self.model.add_module('dropout'+str(i+2), nn.Dropout(p=dropout_rate))
        self.model.add_module('final', nn.Linear(hidden_layers[i+1], 1))

    def forward(self, x):
        return self.model(x)
```

Figure 5: Network Structure for Dropout

```
class Regressor:
    def __init__(self, hidden_layers=[1024, 1024, 1024], dropout_rate=0.2, activation_function='relu', \
                 epoch=1000, lr=0.0001, weight_decay=1e-6):
        self.model = My_Network(hidden_layers=hidden_layers, dropout_rate=dropout_rate, activation_function=activat
        self.model.cpu()
        self.criterion = nn.MSELoss().cpu()
        self.epoch=epoch
        self.lr=lr
        self.weight_decay=weight_decay
        self.optimizer = optim.Adam(self.model.parameters(), lr=lr, weight_decay=weight_decay)

    def fit(self, X_train, y_train, verbose=True):
        X = Variable(X_train)
        y = Variable(y_train)
        print(self.model)
        self.model.train()
        start_time = time.time()
        for epoch_i in range(self.epoch):
            epoch_i += 1
            epoch_loss = 0
            self.optimizer.zero_grad()
            net_out = self.model(X)
            loss = self.criterion(net_out, y)
            loss.backward()
            self.optimizer.step()
            epoch_loss += float(loss.data)
            epoch_loss /= len(X)
            if verbose:
                print(f'Epoch {epoch_i} | loss: {epoch_loss:.4f}')
        running_time = time.time() - start_time
        print('Running time in seconds:', running_time)
        return self
```

Figure 6: Dropout network training

```
'''
    y_pred = prediction by dropout model
    y_hat_mc = prediction by dropout model using monte carlo sampling
'''
def predict_reg(model, X, T=1000):
    X = Variable(torch.from_numpy(X).type(torch.FloatTensor))
    model = model.train()
    y_hat_mc = np.array([model(X).data.cpu().numpy() for _ in range(T)]).squeeze()
    model = model.eval() # only retrieved pre-trained model
    y_pred = model(X).data.cpu().numpy()
    model = model.train()# to set a flag( do not take pre-trained model)
    return y_pred, y_hat_mc
```

Figure 7: Generate predictions

- Regularisation via a compression cost on the weights
- Richer representations and predictions from model averaging
- Exploration in simple reinforcement learning problems such as contextual bandits.

[1] derives a variational approximation to the true posterior. This algorithm makes network more honest with respect to their overall uncertainty also automatically leads to requlariza tion, therefore eliminates the need of using dropout in this model.

A simple prior is the Gaussian distribution. We can define the Gaussian distribution and our Gaussian prior as shown in Figure 8.

```
log2pi = np.log(2.0 * np.pi)
def log_gaussian(x, mu, sigma):
    return float(-0.5 * log2pi - np.log(np.abs(sigma))) - (x - mu)**2 / (2 * sigma**2)

def log_gaussian_logsigma(x, mu, logsigma):
    return float(-0.5 * log2pi) - logsigma - (x - mu)**2 / (2 * torch.exp(logsigma)**2)
```

Figure 8: Log Gaussian

Instead of a single Gaussian, we use mixture priors. A Gaussian distribution has been chosen for defining variational posterior on the weights is centered on the mean vector. An implementation of mixture priors and variational posteriors is shown in Figure 9, and its corresponding network structure for Bayes by BackProp is given in Figure 10. An implementation of the combined loss function is shown in Figure 11

```
class MLPLayer(nn.Module):
    def __init__(self, n_input, n_output, sigma_prior):
        super(MLPLayer, self).__init__()
        self.n_input = n_input
        self.n_output = n_output
        self.sigma_prior = sigma_prior
        self.W_mu = nn.Parameter(torch.Tensor(n_input, n_output).normal_(0, 0.01))
        self.W_logsigma = nn.Parameter(torch.Tensor(n_input, n_output).normal_(0, 0.01))
        self.b_mu = nn.Parameter(torch.Tensor(n_output).uniform_(-0.01, 0.01))
        self.b_logsigma = nn.Parameter(torch.Tensor(n_output).uniform_(-0.01, 0.01))
        self.lpw = 0
        self.lqw = 0

    def forward(self, X, infer=False):
        if infer:
            output = torch.mm(X, self.W_mu) + self.b_mu.expand(X.size()[0], self.n_output)
            return output

        epsilon_W, epsilon_b = self.get_random()
        W = self.W_mu + torch.log(1 + torch.exp(self.W_logsigma)) * epsilon_W
        b = self.b_mu + torch.log(1 + torch.exp(self.b_logsigma)) * epsilon_b
        output = torch.mm(X, W) + b.expand(X.size()[0], self.n_output)
        self.lpw = log_gaussian(W, 0, self.sigma_prior).sum() + log_gaussian(b, 0, self.sigma_prior).sum() #
        self.lqw = log_gaussian_logsigma(W, self.W_mu, self.W_logsigma).sum() + log_gaussian_logsigma(b, self.b_mu,
        return output

    def get_random(self):
        return Variable(torch.Tensor(self.n_input, self.n_output).normal_(0, self.sigma_prior).cpu()), Variable(to
```

Figure 9: Mixture priors and variational posteriors

```
class MLP(nn.Module):
    def __init__(self, n_input, sigma_prior):
        super(MLP, self).__init__()
        self.l1 = MLPLayer(n_input, 500, sigma_prior)
        self.l1_relu = nn.ReLU()
        self.l2 = MLPLayer(500, 500, sigma_prior)
        self.l2_relu = nn.ReLU()
        self.l3 = MLPLayer(500, 1, sigma_prior)

```

Figure 10: Network structure for Bayes by BackProp

```
def forward_pass_samples(X, y):
    s_log_pw, s_log_qw, s_log_likelihood = 0., 0., 0.
    for _ in range(n_samples):
        output = net(X)
        sample_log_pw, sample_log_qw = net.get_lpw_lqw()
        sample_log_likelihood = log_gaussian(y, output, sigma_prior).sum()
        s_log_pw += sample_log_pw
        s_log_qw += sample_log_qw
        s_log_likelihood += sample_log_likelihood

    return s_log_pw/n_samples, s_log_qw/n_samples, s_log_likelihood/n_samples

def criterion(l_pw, l_qw, l_likelihood):
    return ((1./n_batches) * (l_qw - l_pw) - l_likelihood).sum() / float(batch_size)
```

Figure 11: Combined Loss

# 4 Evaluation and Results

I used power dataset from the [2] with the following features that are hourly average ambient variables:

- Temperature (T) in the range 1.81C and 37.11C
- Ambient Pressure (AP) in the range 992.89-1033.30 milibar
- Relative Humidity (RH) in the range 25.56
- Exhaust Vacuum (V) in the range 25.36-81.56 cm Hg
- Net hourly electrical energy output (EP) 420.26-495.76 MW

The complete dataset that I used has dimensions (9568,5), out of which the 10% (957,4) was used for the testing and the remaining 90% was used for training (the last column in the complete dataset is target).

For the first method [2] that I implemented, has following parameters:

- hidden layers = [1024, 1024, 1024]
- dropout-rate = 0.2
- activation = relu
- epoch = 100

- learning-rate = 0.0001
- weight-decay = 1e-6

For the second method [1], I used following parameters:

- layers = [1024, 1024, 1024]
- activation = relu
- epoch = 10
- learning-rate = 0.001
- batch-size = 100
- n-samples = 3

Generated point predictions and prediction intervals with true target values for the two methods are shown in Figure 12 and Figure 13, respectively.

For the first method, the evaluation metrics (RMSE, PICP and MIPW) are:

- RMSE = 27.923414
- PICP = 0.907
- MIPW = 91.96

For the second method, the evaluation metrics (RMSE, PICP and MIPW) are:

- RMSE = 31.28
- PICP = 0.993
- MIPW = 103.01

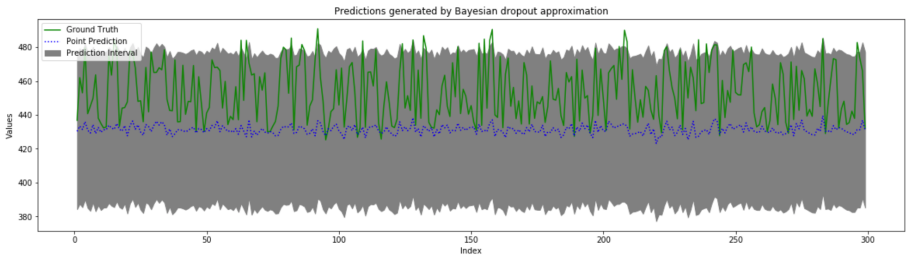


Figure 12: Predictions: Dropout as a Bayesian Approximation

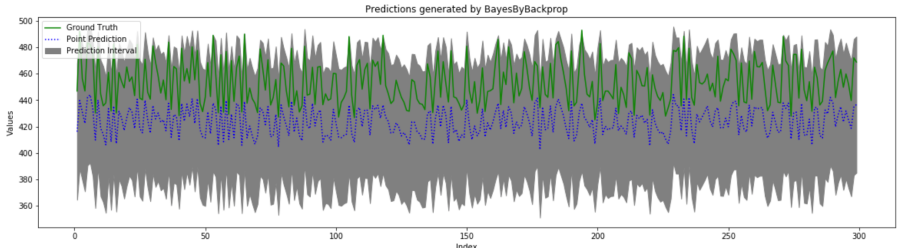


Figure 13: Predictions: Dropout as a Bayesian

# References

[1] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural network. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1613–1622, Lille, France, 07–09 Jul 2015. PMLR.

[2] Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1050–1059, New York, New York, USA, 20–22 Jun 2016. PMLR.

[3] Tim Pearce, Alexandra Brintrup, Mohamed Zaki, and Andy Neely. High-quality prediction intervals for deep learning: A distribution-free, ensembled approach. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4075–4084, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.