Deep Learning : A Bayesian Perspective

Jonathan Fhima
Department of Applied Mathematics, ENS
Master MVA
jonathan.fhima@dauphine.eu

John Levy
Department of Applied Mathematics, ENS
Master MVA
johnlevy125@gmail.com

Abstract

This paper reviews several research papers on the topic of Bayesian applied to Deep Learning and is main focused on Deep Learning: a Bayesian perspective [1]. First we will present the fundamentals of Deep Learning. Then we will speak about how regularization problems can be seen as Bayesian problems and how Bayesian theory can helps to improve the performance of classical Deep Learning solution. After that we will present some code implementation we have done. To conclude we will try to give some points of improvement from there.

1 Deep Learning

1.1 Introduction

Deep Learning has become one of the most popular area for machine learning in the recent years. Indeed, thanks to the processing power and the amount of data that are now available, Deep Learning techniques are now commonly used and performs state-of-the-art results for classification or prediction problems. Deep Learning is a branch of Machine learning which relies on Neural Network. We are going to present the main idea behind Neural Network architecture and why it works so well.

1.2 Deep Learning Theory

The goal of Deep Learning is to find the right output Y given a high dimensional input, that is to say, find a function F such that Y = F(X), where $X = (X_1, ..., X_p)$. For instance, in classification problem, we need to learn F such that $F : X \to Y$ where $Y \in \{1, ...K\}$. The Deep Learning approach uses univariate activation functions to decompose a high dimensional X. Indeed, Kolmogorov-Arnold showed that any multivariate continuous function can be represented with composition of univariate functions. With this result, we are now able to represent our Neural Network :

$$\begin{aligned}
Z^{(1)} &= f^{(1)}(W^{(0)}X + b^{(0)}) \\
Z^{(2)} &= f^{(2)}(W^{(1)}X + b^{(1)}) \\
&\dots \\
Z^{(L)} &= f^{(L)}(W^{(L-1)}X + b^{(L-1)}) \\
\hat{Y}(X) &= W^{(L)}X + b^{(L)}
\end{aligned}$$

where, $W^{(l)}$ are weights matrices, $b^{(l)}$ are the bias, and $\hat{Y}(X)$ the predictor (i.e the output). Now it is clear, that $\hat{Y}(X) = (f_1^{W_1,b_1} \circ ... \circ f_L^{W_L,b_L})$ where $(f_1, ..., f_L)$ are univariate function defined by $f_l^{W_l,b_l} = f_l(\sum_{j=1}^{N} W_{i,j} z_j + b_l)$. Hence, the mapping function F is modeled with the superposition of univariate semi affine functions. Another useful result about Deep Learning is the universal approximation theorem. This theorem states that any continuous function can be approximated, under

Preprint. Under review.

mild conditions, as closely as wanted by a two-layer network. As the number of neurons goes to infinity, any continuous function F can be approximated by some neural network $\hat{F}$ because each component $f(W_{(j)}^T x - b_j)$ behaves like a basis function and functions in a suitable space admits a basis expansion.

2 Bayesian Deep Learning and Dimensionality reduction

2.1 Regularization

One of the main problem in Deep Learning is the choice of the complexity of the model. Indeed, if the model becomes complex, it will have a lower bias but a higher variance, but if the model becomes to simple, it will have a low variance but a high bias. This problem is known as the variance-bias trade-off. The goal is to find the model with the optimal trade-off between variance and bias.
After the Neural Networks passes its inputs all the way to its outputs, the network evaluates how good the prediction is by computing the loss functions given by the negative log likelihood :
$L(Y, \hat{Y}^{(W,b)}(X)) = -\log p(Y|\hat{Y}^{(W,b)}(X))$ with (W,b) the parameters of our Neural Network.
Now, we are going to introduce Regularization techniques in a Bayesian perspective. This techniques discourages our network to learn a more complex or flexible model, so as to avoid the risk of overfitting. As we mentioned above, the Network needs to compute its loss functions, and in order to control the bias-variance trade-off, we add a regularization term to the loss.

The new problem is now : $L_\lambda(Y, \hat{Y}^{(W,b)}(X)) = -\log p(Y|\hat{Y}^{(W,b)}(X)) - \log p(\phi((W,b))|\lambda)$ with λ the penalization constant term. It turns out that the regularization term can be viewed as a prior distribution over the parameters of our Neural Network. So by taking the Bayesian point of view : everything is a distributions. We now define θ as the distributions over the parameters of our Neural Network, and we define the model by :

-Prior beliefs on the parameters : $p(\theta)$
-Posterior beliefs on the parameters : $p(\theta|\{X,Y\})$
-Likelihood : $p(Y|\theta,X)$

As the bayesian rule states that : $p(\theta|\{X,Y\}) \propto p(\theta)p(Y|\theta,X)$, finding the maximum A Posteriori (MAP) is defined by : $\max_\theta \log p(\theta|\{X,Y\} = \max_\theta \underbrace{\log p(\theta)}_{Regularization} + \underbrace{\log p(Y|\theta,X)}_{Loss}$
and this can be seen as our new loss that we want to optimize.
For example, we show that L^2 Regularization, also known as Ridge Regularization, is defined by putting a Normal Distribution on the prior. First, we define $p(Y|\theta,X) = \mathcal{N}(X\theta, \sigma^2 I)$ and $p(\theta) = \mathcal{N}(0, \sigma^2)$. Now, let's compute the posterior :

$$\begin{aligned}
\log p(\theta|\{X,Y\}) &\propto \log p(y|\theta,X) + \log p(\theta) \\
&\propto \frac{||X\theta - Y||^2}{2\sigma^2} - \frac{||\theta||^2}{2\sigma^2} \\
&= \frac{(X\theta - Y)^T(X\theta - Y)}{2\sigma^2} - \frac{\theta^T\theta}{2\sigma^2} \\
&= -\frac{\theta^T X^T X\theta - 2\theta^T X^T Y}{2\sigma^2} - \frac{\theta^T\theta}{2\sigma^2} \\
&= -\frac{\theta^T X^T Y}{\sigma^2} - \theta^T \underbrace{(\frac{X^T X}{2\sigma^2} + \frac{I}{2\sigma^2})}_{\Sigma^{-1}}\theta \\
&= -\frac{1}{2}(\theta - \frac{\Sigma X^T Y}{\sigma^2})^T \Sigma^{-1}(\theta - \frac{\Sigma X^T Y}{\sigma^2})
\end{aligned}$$

We have proven that $p(\theta|\{X,Y\}) = \mathcal{N}(\theta|\frac{\Sigma X^T Y}{\sigma^2}, \Sigma)$.
In particular : $\underbrace{\theta_B}_{\theta_{MEP}} = \frac{\Sigma X^T Y}{\sigma^2} = \underbrace{(X^T X + \lambda I)^{-1} X^T Y}_{Ridge}$ with λ the penalization constant term.

2.2 Dropout for Model and Variable Selection

Dropout is a popular regularization technique. The main idea is to inject noise to the weights at each iteration during training. In this part we will focus on Bernoulli noise. With Bernoulli noise, a certain number of weights of our networks will be dropped randomly (according to the probability of dropping a neuron), and will enable our network to learn a fraction of the weights at each training iteration. First, we will show that optimize the model with Dropout is in fact minimizing a regularized network Loss.
For simplicity of the proof, we consider a single layer linear unit in a network, the results hold for multi-layer network with activation functions.

The output $\hat{y}$ is a weighted sum of the inputs, that is to say $\hat{y} = \sum_{i=1}^{n} w_i x_i$.
We define the two following least square loss :

$$L_N = \frac{1}{2}(y - \sum_{i=1}^{n} w_i^{'} x_i)^2 \qquad\qquad L_D = \frac{1}{2}(y - \sum_{i=1}^{n} \delta_i w_i x_i)^2$$

L_N is the loss for a regular network, and L_D for a dropout network with $\delta_i \sim Bern(p)$. As the back-propagation in our training uses a gradient descent, we will first look at the gradient of the dropout network :

$$\frac{\partial L_D}{\partial w_i} = -y\delta_i x_i + w_i \delta_i^2 x_i^2 + \sum_{j=1,j\neq i}^{n} w_j \delta_j \delta_i x_i x_j$$

In the equation of the loss function for regular network, we have defined $w^{'}$, now we suppose that $w^{'} = p * w$. So, by taking the derivative of the loss function for this regular network, we get :

$$\frac{\partial L_N}{\partial w_i} = -y p_i x_i + w_i p_i^2 x_i^2 + \sum_{j=1,j\neq i}^{n} w_j p_j p_i x_i x_j$$

And by taking the expectation of the above equation for the gradient of the dropout loss, we get :

$$\begin{aligned} \mathrm{E}[\frac{\partial E_D}{\partial w_i}] &= -y p_i x_i + w_i p_i^2 x_i^2 + w_i \mathrm{Var}(\delta_i) x_i^2 + \sum_{j=1,j\neq i}^{n} w_j p_j p_i x_i x_j \\ &= \frac{\partial E_N}{\partial w_i} + w_i \mathrm{Var}(\delta_i) x_i^2 \\ &= \frac{\partial E_N}{\partial w_i} + w_i p_i (1-p_i) x_i^2 \end{aligned}$$

Hence, we have that minimizing the dropout loss is equivalent to minimizing a regularized network of the form :

$$L_R \quad = \quad \underbrace{\frac{1}{2}(y - \sum_{i=1}^{n} p_i w_i x_i)^2}_{Loss} \quad + \quad \underbrace{\sum_{i=1}^{n} p_i (1-p_i) w_i^2 x_i^2}_{Penalization}$$

This concludes the proof that Dropout can be viewed as minimizing the usual loss with a penalization term. In fact, it is simply a Bayes ridge regression with a g-prior as an objective function.
Dropout can also select the right choice of neurons in a layer. Indeed, by dropping units rather than the input layer, we can establish which probability p gives the best results.

To go further in our experimentation, we also studied Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning [3]. As above, we are again interested in a Bernoulli Noise. For each W_i, we define q to be the product of the mean weight matrix M_i and the diagonal matrix composed by $z_{i,j} \sim Bern(p)$. Hence, $q_{M_i}(W_i) = M_i \cdot diag([z_{i,j}])$. Sampling the diagonal element z from a Bernoulli is identical to randomly setting columns of M to 0 which is identical to randomly setting units of the network to 0. At the end, dropout can be approximated by averaging the weights of the network, and by doing so, we will be able to catch the uncertainty of our model. To convince one that by using dropout, we are catching the uncertainty in the model, we have implemented a Neural Network with Dropout, the results are presented in the section 4 Experiments.

2.3 Dimension Reduction with Bayesian

In this section, we present a Bayesian model selection approach to perform dimensionnality reduction. The goal of dimensionnality reduction is to find the lowest dimensions space to project our data and in the same time, keep as much information as we can. To this end, we study a probabilistic principal component analysis model (PPCA) based on a normal-gamma prior distribution. Specifically, PCA consists in transforming the variables to a new set of variables called principal components which are orthogonal and ordered such that the variation present in the original variables decreases as we move down in the order. To choose the projection space, a common practice is to consider the eigenvalues of the sample covariance matrix, but no solution has been widely accepted for choosing how many PCs should be computed. Most of the methods for finding the number of PCs are based on asymptotic assumptions. A natural non-asymptotic method is provided by exact Bayesian model selection. We present here a Bayesian method for selecting the number of PCs that relies on a prior structure based on the PPCA model. Let's introduce the matrix of data $X = (x_1, ..., x_n)^T$ which is an $(n \times p)$ matrix.
The goal is to project X on a d-dimensional subspace while retaining as much variance as possible, and we defined the PPCA model $\mathscr{M}_d$ by :

$$X_i = Wy_i + \varepsilon_i$$

where $y_i \sim \mathscr{N}(0, I_d)$ is a low-dimensional Gaussian latent vector, W is a $p \times d$ parameter matrix called the loading matrix and $\varepsilon_i \sim \mathscr{N}(0, \sigma^2 I_p)$ is a Gaussian noise term. This model is equivalent to PCA model as the principal component of X can be retrieved using the maximul likelihood estimator W_{ML}. The main advantage of study PPCA is that, as a probabilistic model, standards techniques are available. Now, we present the PPCA model under a Bayesian structure. Indeed, we consider a Gaussian prior distribution on the loading matrix W with $w_{j,k} \sim \mathscr{N}(0, \phi^{-1})$ for $j \in \{1, ..., p\}$ and $k \in \{1, ..., d\}$ a gamma prior distribution over the noise variance $\sigma^2 \sim Gamma(a, b)$ with strictly positive parameter a, b, and ϕ. We are also considering that $\forall d \in \{1, ..., p\}$ $p(\mathscr{M}_d) \propto 1$.
The posterior probabilities of models can be written as, $\forall d \in \{1, ...p\}$:

$$p(\mathscr{M}_d | X, a, b, \phi) \propto p(X | a, b, \phi, \mathscr{M}_d) p(\mathscr{M}_d)$$

where

$$p(X|a,b,\phi,\mathscr{M}_d) = \prod_{i=1}^{n} \int_{\mathbb{R}^{d \times p} \times \mathbb{R}^+} p(x_i | W, \sigma, \mathscr{M}_d) p(W|\phi) p(\sigma | a, b) dW d\sigma$$

is the marginal log-likelihood of the data under condition of independence.
Moreover, it has been proved that this marginal log-likelihood can be approached thanks to the choice of the priors. As a result, the marginal log-likelihood of the model, when $b = \frac{\phi}{2}$ is given by :

$\log p(X|a,\phi,\mathscr{M}_d) = \sum_{i=1}^{n} \log p(x_i|a,\phi,\mathscr{M}_d)$ where the posterior law of x_i is :

$$x_i \sim GAL_p(2\phi^{-1} I_p, 0, a + d/2)$$

with GAL the Generalized Gauss-Laplace law.

2.4 Why use Bayesian Neural Network

In classical deep learning, we want to learn what are the best weights for our model, to generate an accurate prediction $\hat{Y}(X)$ given an input X, in term of minimisation of a loss function. Even if this kind of methods works pretty well with several tasks, like for instance classification for image recognition, it doesn't allow you to catch uncertainty from your model.
For some tasks it doesn't matter because you think that your model have an accuracy as good that you could rely on it. But imagine that you are working on a project where generate a bad prediction is almost lethal, then being able to know how certain you are about your prediction is crucial.

2.5 Bayesian Framework for Deep Learning

To be able to generate a Bayesian Neural Network, we will first introduce a Bayesian framework. From now we will write $D = (X_{train}, Y_{train})$ the training data set where X_{train} are the inputs and

Y_{train} the values to predict. Let write X^* a new observation and Y^* the unknown label of this new observations.
Given the data training D, the goal of a Bayesian Neural Network is to build a model $p(\hat{Y}|X^*, X_{train}, Y_{train}, \theta)$ and to find the parameter θ which maximise this value. To do so, we have to:
-Define a prior distribution on θ : $p(\theta)$
-Find the posterior thanks to the training: $p(\theta|D) = p(\theta|X_{train}, Y_{train}) = \frac{p(\theta, Y_{train}|X_{train})}{p(Y_{train}|X_{train})} =$ $\boxed{\frac{p(Y_{train}|\theta, X_{train})p(\theta)}{\int p(Y_{train}|\theta, X_{train})p(\theta)d\theta}}$

-Predict the label $\hat{Y}$ of a new input X^* using total probabilities: $\boxed{p(\hat{Y}|X^*, X_{train}, Y_{train}) = \int p(\hat{Y}|X^*, \theta)p(\theta|X_{train}, Y_{train})d\theta}$
The main difficulty is that the two boxed equations are intractable because it is not possible to integrate for all the possible values of θ. Even if we knew all his possible values, it will not be computationally efficient. We will see in the section B. Bayesian Inference for Deep Learning how to solve these problems.

2.6 Bayesian Inference for Deep Learning

As we say in the previous section. Bayesian framework, our first problem is that the posterior law of θ : $p(\theta|X_{train}, Y_{train})$ is intractable. To solve this problem, we will try to find an other function $q(\theta|\phi)$ which approximate this posterior. To do so, we will use the Kullback-Leibler divergence. Indeed, a good candidate for $q(\theta|\phi)$, will be the function which minimize the Kullback-Leibler divergence with the posterior law, with respect to ϕ :

$$\begin{aligned}
KL(q||p) &= \int q(\theta|D,\phi)\log(\frac{q(\theta|D,\phi)}{p(\theta|D)})d\theta \\
&= \int q(\theta|D,\phi)\log(\frac{q(\theta|D,\phi)p(D)}{p(\theta,D)})d\theta \\
&= \int q(\theta|D,\phi)\log(p(D))d\theta - \int q(\theta|D,\phi)\log(\frac{p(\theta,D)}{q(\theta|D,\phi)})d\theta \\
&= \log p(D) - \boxed{\int \frac{q(\theta|D,\phi)\log(p(Y_{train}|X_{train},\theta)p(\theta)}{q(\theta|D,\phi)}}
\end{aligned}$$

The left term of the last equation doesn't depend on ϕ and the problem becomes the maximization of the boxed formula, which is called the ELBO, with respect to ϕ.
The resulting maximization problem is solved using gradient descent. To do so, it is more convenient to write the ELBO as :

$$ELBO(\phi) = \int q(\theta|D,\phi)\log p(Y_{train}|X_{train},\theta)d\theta - \int q(\theta|D,\phi)\log(\frac{q(\theta|D,\phi)}{p(\theta)})d\theta$$

The gradient of the first term : $\nabla_\phi \int q(\theta|D,\phi)\log p(Y_{train}|X_{train},\theta)d\theta = \nabla_\phi E_q \log p(Y_{train}|X_{train},\theta)$ is not an expectation. So we can't calculate it using Monte Carlo methods. We want to represent it as an expectation of some random variable to use Monte Carlo methods.
There are two standard methods to do it :
1- The log derivative trick : we use the fact that $\nabla_x f(x) = f(x)\nabla_x log f(x)$
Indeed we will be able to calculate:

$$\begin{aligned}
\nabla_\phi \int q(\theta|D,\phi)\log(p(Y_{train}|X_{train},\theta))d\theta &= \int \nabla_\phi q(\theta|D,\phi)\log(p(Y_{train}|X_{train},\theta))d\theta \\
&= \int \frac{q(\theta|D,\phi)}{q(\theta|D,\phi)}\nabla_\phi q(\theta|D,\phi) log(p(Y_{train}|X_{train},\theta))d\theta \\
&= \int q(\theta|D,\phi)\nabla_\phi \log(q(\theta|D,\phi))\log(p(Y_{train}|X_{train},\theta)) \\
&= E_q[\nabla_\phi log(q(\theta|D,\phi))\log(p(Y_{train},X_{train}))]
\end{aligned}$$

Now the gradient is an expectation and could be calculated thanks to Monte Carlo methods if we choose a family of function for $q(\theta|D,\phi)$ which we know the derivative. We can use the same trick for the right part of the ELBO.

2- The reparametrization trick by representing θ as a value of a deterministic function $\theta = g(\varepsilon, x, \phi)$, where $\varepsilon \sim r(\varepsilon)$ does not depend on ϕ. The derivative is given by :

$$\begin{aligned}\nabla_\phi E_q[\log(p(Y_{train}|X_{train},\theta)] &= \int r(\varepsilon)\nabla_\phi \log(p(Y_{train}|X_{train},g(\varepsilon,x,\phi)))d\varepsilon \\ &= E_\varepsilon[\nabla_g \log(p(Y_{train}|X_{train},g(\varepsilon,x,\phi)))\nabla_\phi g(\varepsilon,x,\phi)]\end{aligned}$$

The reparametrization is trivial when $q(\theta|D,\psi) = \mathcal{N}(\theta|\mu(D,\phi),\Sigma(D,\phi))$, and $\theta = \mu(D,\phi) + \varepsilon\Sigma(D,\phi)$ with $\varepsilon \sim \mathcal{N}(0,I)$.

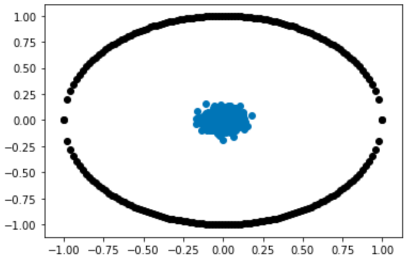
3 Finding Good Bayes Predictor

The main reason about why deep learning works is because it is a flexible way of splitting the input space into sub spaces. This is also what does tree models or kernel based model. It visibly build a kernel and when you give it an input the model will try a neighbours and the weight which will be use for these neighbour. Then it does a weighted average of these neighbours and it becomes your prediction.

3.1 High dimensional space

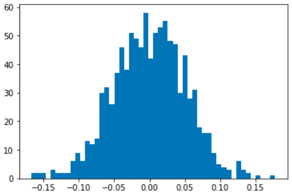
The main reason why we need more flexible methods is that in high dimensional space, there is the concentration of measure which will perturbed the way to find neighbours.
A simple experience I have implemented is to sample uniformly some points on the unit ball of R^{400}. And represent the projection of these point on the vector space generated by the two units vectors $e_1 = (1,0,.....0)$ and $e_2 = (0,1,0,.....0)$. You can see in the figure below that all the point are concentrated near the origin.

```
<matplotlib.collections.PathCollection at 0x7f3b4a5fd9b0>
```



An other simple experiment is to plot the histogram of the projection on the same points on the vector space generated by the unit vector e_1. And you can see in the figure below that, according to Maxwell theorem, the histogram converge to the distribution of a normal law centered in zero. And the more the dimension of the space of sampling is high, the more the variance of this normal law is small.

```
<a list of 50 Patch objects>)
```

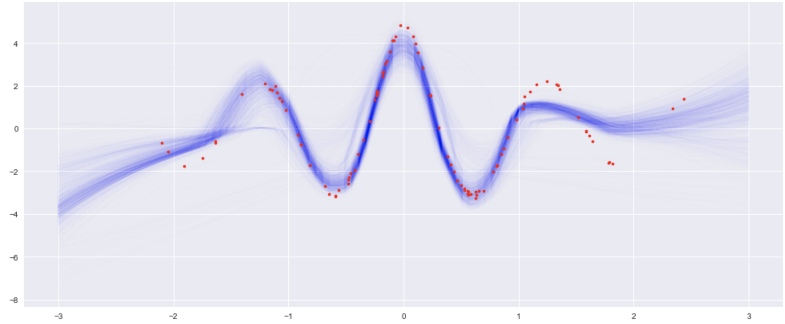


So if we want to use a kernel base technique for this kind of point, your prediction will fails because the new point you want to predict the label will not have any neighbours.
Deep learning allows to solve this problem.

4 Experiments

4.1 Dropout to catch uncertainty implementation

In order to have a better understanding on how dropout can catch uncertainty, we implement a neural network that follow the "Dropout as a Bayesian Approximation" paper.



The main idea is we can look at the uncertainty, if you condition on any particular input then you are going to be able to characterize both what is the mean and what is the variance of that input, for example if we are predicting at the very left, we are gonna see quite high variance because we are relatively uncertain, but near to 0, we have a lot of agreement between your samples from the approximate posterior, so it lead to a higher confidence.

4.2 Bayesian Deep Neural Network to catch uncertainty implementation

4.2.1 Introduction

In this section, we have implemented a Bayesian Deep Neural Network and trained it on the MNIST dataset.
The structure of the network is very simple :
- A fully connected layer of input size 28x28, output size 1024 with Relu activation function.
-An other fully connected layer of input size 50 and output size 10. With this kind of networks, without using Bayesian framework, we have an accuracy of 89% on the test dataset.

4.2.2 Bayesian Framework

We use the module Pyro to modelise our Bayesian Framework.
First we had to define the model:
We put a multivariate Normal prior with 0 means and Identity covariance matrix for the all the weights of our two layers. Then during the different epochs, the model will have to lift network parameters to random variable sample from the prior.
Then we define a guide, which role will be to be able to approximate the posterior of the observations generated by the model. It is the equivalent of what we called variationnal inference in the section 2.

4.2.3 Result without using uncertainty

When we don't take into account uncertainty, our accuracy is the same than with the non Bayesian model. To do our estimation, we will sample several iid networks (thanks to our posterior law on the weights) here 10, and say that the prediction of the network is the prediction which has been output by the biggest number of these networks. To do so, we use a softmax, and return the biggest probability with his index. In the next sections, we will call p this probability and r the prediction.

4.2.4 Results taking uncertainty into account

Uncertainty allows us to decide if we want to predict an output for an input or if we don't want to do it if we are not confident enough on the prediction of our network. To do so, we now generate n different networks (here 100) with the same process as before. And will process the outputs of these 100 networks. We have implemented three different way to do it.

4.2.5 First way and second way of using uncertainty

In this section, we will compute the prediction as the same way as above, but furthermore we will also compute the variance of the different outputs from our networks. We will first assume that our prediction is the good one. Then we will modelise our outputs as if they was different values of a sampled Bernoulli law of parameter p, with value 1 if the prediction is the same than r, or 0 otherwise. Having all these statistics, we will be able to calculate the variance of our observations: $V = \frac{\sum_{i=0}^{n}(1_{P_i=r}-r)^2}{100}$
Denoting P_i the prediction of the neural i Once having this variance and the prediction r, We will be able to calculate a 95% confidence interval for our prediction, ie an upper and a lower bound for our probability p, thanks to the formula: $I = [p - 1.96(\frac{variance}{n})^{0.5}; p + 1.96(\frac{variance}{n})^{0.5}]$
Thanks to that we will be able to decide when we want to make our prediction. The first thing we have tried was to be opportunist in face of uncertainty and to decide to make a prediction when the lower bound confidence was above 0,5. Making this limit higher will give you a better accuracy on the predicted images, but you will make a prediction on a smaller number of images. For instance, putting this limit at 0.95 allows us to have an accuracy of 100% on our predictions with our simple model, but our model decide to make a prediction for only 893 over 5000.
The limit you will put depend on the expectations you have on your network. When putting this limit at 0.5, we obtained an accuracy of $96,11\%$ on the prediction we have made (ie 4328 over 5000 test images). Here is the detail of our results:

number of predictions	number of good predictions	number of false predictions	Accuracy when decide to answer
4328	4160	168	96,11%

number of time we didn't make a prediction	number of time it was a good choice	number of time it was a bad choice	Ratio
672	298	374	44%

The second thing we have tried is to use the 5000 first images of the test set (over 10 000) to train a LogisticRegressor from scikit-learn. The output of this regressor is : 1 if the Bayesian Neural Network should predict a value and 0 otherwise. Using this strategy, we obtained an accuracy of 93.18% on the prediction we have made (ie 4782 ove 5000) on the same data set that the previous strategy. Here is the detail of our results:

number of predictions	number of good predictions	number of false predictions	Accuracy when decide to answer
4782	4456	326	93,2%

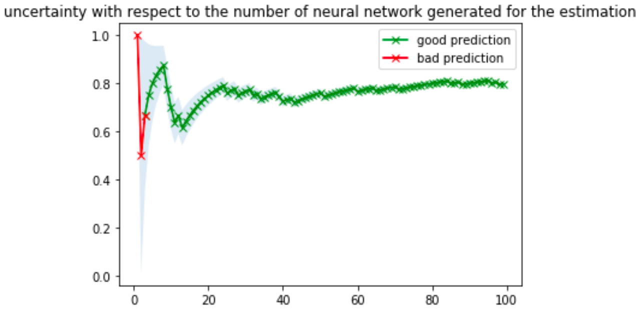
number of time we didn't make a prediction	number of time it was a good choice	number of time it was a bad choice	Ratio
218	114	104	0.52%

I decided to continue the experiments of this section with the second strategy. One of the other interest of being able to not answer is that my network will not make a prediction if the input is completely different from the data used for the training.
Here we tried to challenge our network with a picture of a monkey find on google. We use this picture as input of our 100 Bayesian Neural Networks without taking into account uncertainty and as we could expect the prediction was false, our networks told me that it was an image of the number seven instead of a monkey.
To compare, we use uncertainty with my regressor with the variance and the probabilities give by my networks for this image, and the regressor told me that he would rather not predict any output for the monkey image.

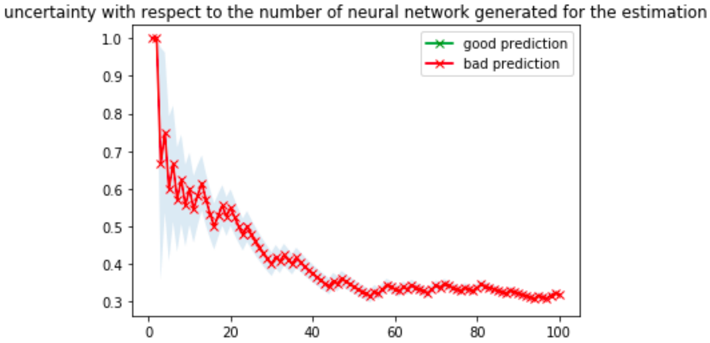
From now we are convinced of the advantage of using Bayesian frameworks to catch uncertainty.

But even if ours networks are able to say "I don't know", we were not convinced. Indeed we would like to evaluate the number n of network necessary to outperform a classical neural network, and also to plot the uncertainty.

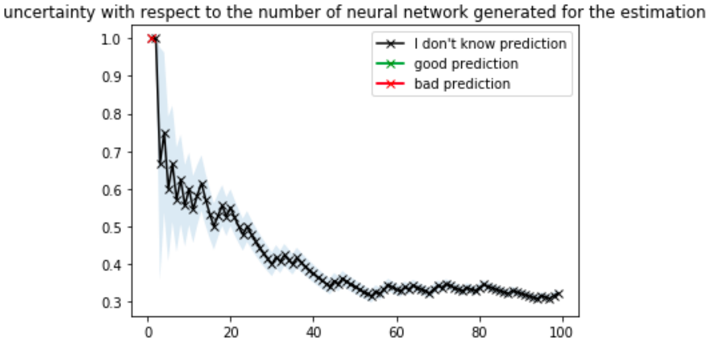
This is why I decided to select an image from the mnist dataset and to plot the predictions of my n networks with respect to n. In this picture, the y axis represent the probability output from my network, the x axis represent the number of network use for the prediction (the number 0 is equivalent of using a non Bayesian neural network). And the blue shape represent the upper and lower bound confidence on the prediction (ie the uncertainty).



As you can see, the Bayesian networks outperform the non Bayesian network even if n is small.
We also decided to plot the same kind of figure for the predictions of my networks for the monkey image.
This image is when we don't take uncertainty into account :



As you could imagine all the networks are wrong and it is normal because they are not trained to recognize a monkey.
To compare with this figure, I plot the same one but taking uncertainty into account :

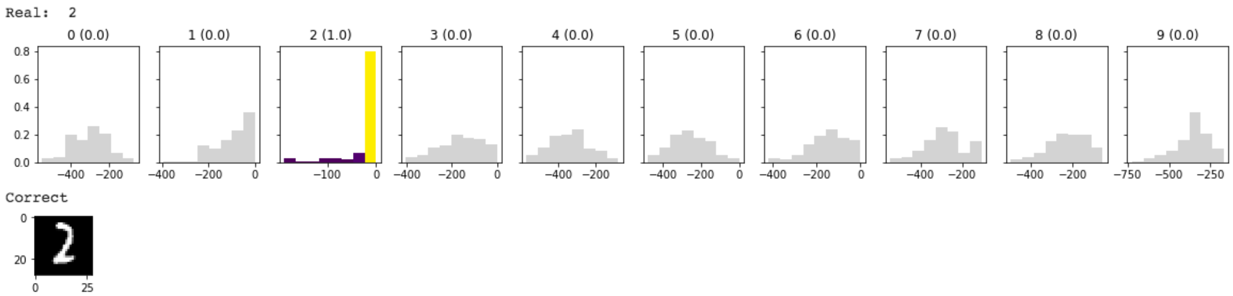


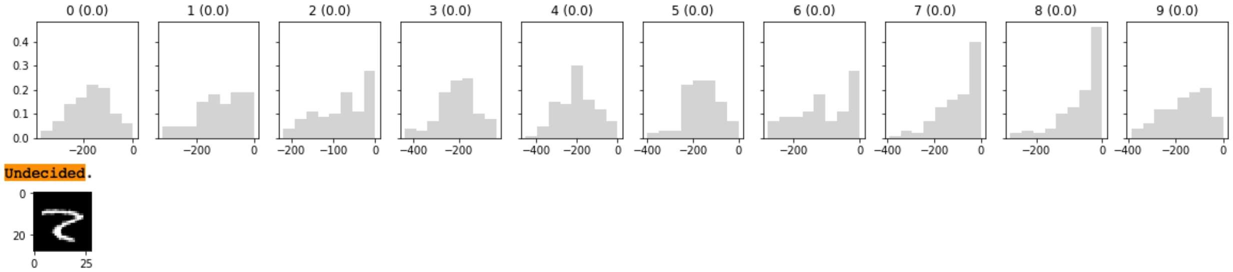
And as we could expect, only the non Bayesian network was not able to take a good decision.

4.2.6 Third way of using uncertainty

This strategy is inspired from the article : "Making your Neural Network say I don't know" [2].
For this strategy, we will save all the outputs from our n networks after applying them a softmax. We consider that the true probability of being in a category of number, is the median of the probability of being in this same category given by all this vectors. Then we will make a prediction only if one

of this probabilities is greater than 0.2. If it is the case we will take the category with the biggest probability as prediction.
In the next pictures, you will see our results when our network have a good prediction and when he decided to not answer.





We evaluate this method for 128 images of the test dataset. The network decided to make a prediction for 118 of these images with an accuracy of 99%.

5 Conclusion

We have worked on several research papers but our reference has always been the paper [1]. Thanks to this work we have deeply understood how Bayesian Neural Networks work and how they can be useful to catch uncertainty. One point of improvement of the article [1] is that it speaks about a lot of different things but not in details. Our work was to understand these key concepts, to try to prove some results which was not originally proved, and to find other sources of information to complete the original paper. The main limit we have found to Bayesian Neural Networks is his computational cost. Indeed we have often evaluated our inputs thanks to several neural networks sampled from our posterior law. This multiple evaluation is of course slower than an evaluation through a single classical neural network.

References

[1] Nicholas Polson, Vadim Sokolov, 2017. Deep Learning, a Bayesian Perspective

[2] Paras Chopra, 2018. Making Your Neural Network Say “I Don’t Know”.

[3] Yarin Gal, Zoubin Ghahramani, 2016. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning.

[4] Charles Bouveyron, Pierre Latouche, Pierre-Alexandre Mattei, 2019. Exact Dimensionality Selection for Bayesian PCA