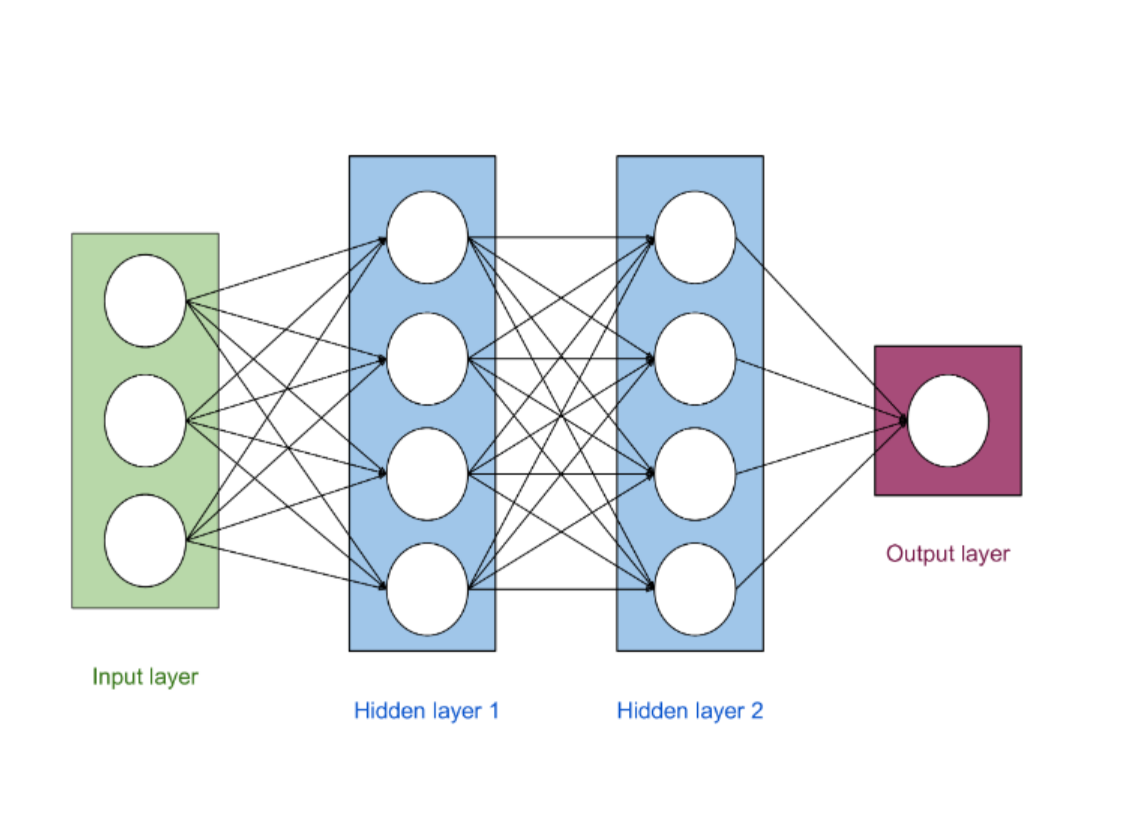
medium.com

Bayesian Neural Network Series Post 2: Background Knowledge

Kumar Shridhar

17-21 minutes



This post is the second post in an **eight-post series of Bayesian Convolutional Networks**. The posts will be structured as follows:

1. **Need** for Bayesian Neural Networks
2. **Background knowledge** needed to understand Bayesian Networks better
3. Some **recent work** in the field of Bayesian Neural Networks

4. Bayesian Convolutional Networks **using Variational Inference**
5. Build your own **Bayesian Convolutional Network in PyTorch**
6. **Uncertainty estimation** in a Bayesian Neural Network
7. **Model Pruning** in a Bayesian Neural Network
8. **Applications** in other areas (Super Resolution, GANs and so on..)

It is highly recommended to read the first post about the need for Bayesian Networks before proceeding further. Also, check out the post 'Why the World needs a Bayesian perspective'.

Let's start this post by breaking **Bayesian Neural Networks** into **Bayesian** and **Neural Networks**.

Bayesian inference forms an important part of statistics and probabilistic machine learning. It is based on **Bayes' theorem** given by famous statistician Thomas Bayes. In Bayesian Inference, the hypothesis probability is updated as more evidence or information becomes available.

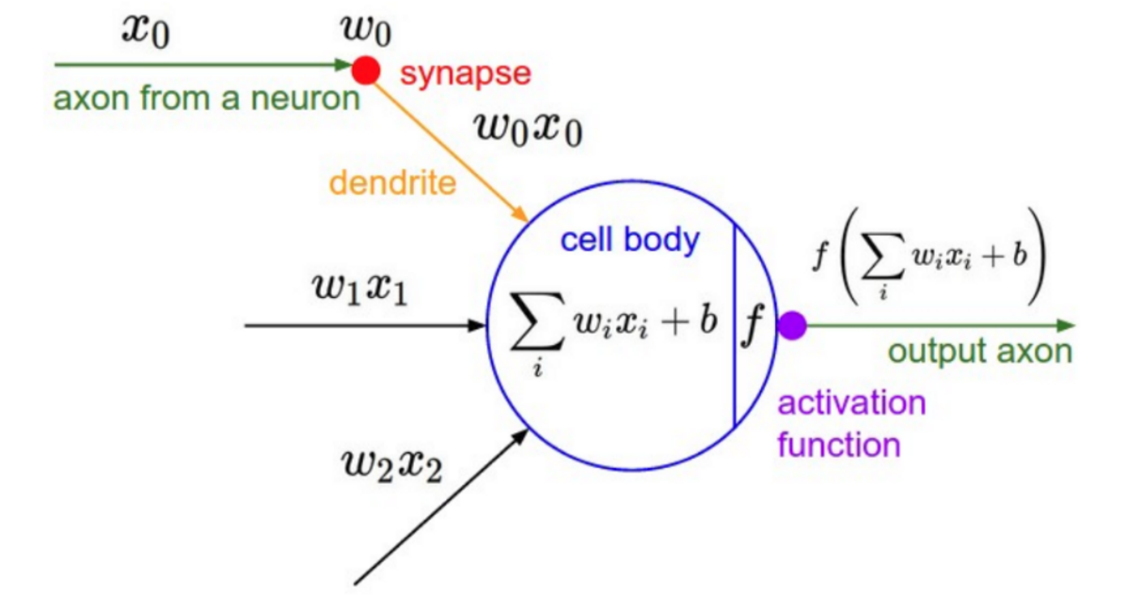
Neural Network, on the other hand, can be thought of an end to end systems or set of algorithms that mimic the human brain (not everyone believes in it but it was the foundation) and tries to learn complex representation within the dataset to provide an output.

Neural Networks

There are very good tutorials already on Neural Networks. I will try to brief the neural networks analogy with the brain and will spend more time explaining the Probabilistic Machine Learning segments that we will work on in future.

Brain Analogies

A **perceptron** is conceived as a mathematical model of how the neurons function in our brain by a famous psychologist **Rosenblatt**. According to Rosenblatt, a neuron takes a set of binary inputs (nearby neurons), multiplies each input by a continuous-valued weight (the synapse strength to each nearby neuron), and thresholds the sum of these weighted inputs to output a 1 if the sum is big enough and otherwise a 0 (the same way neurons either fire or do not fire).



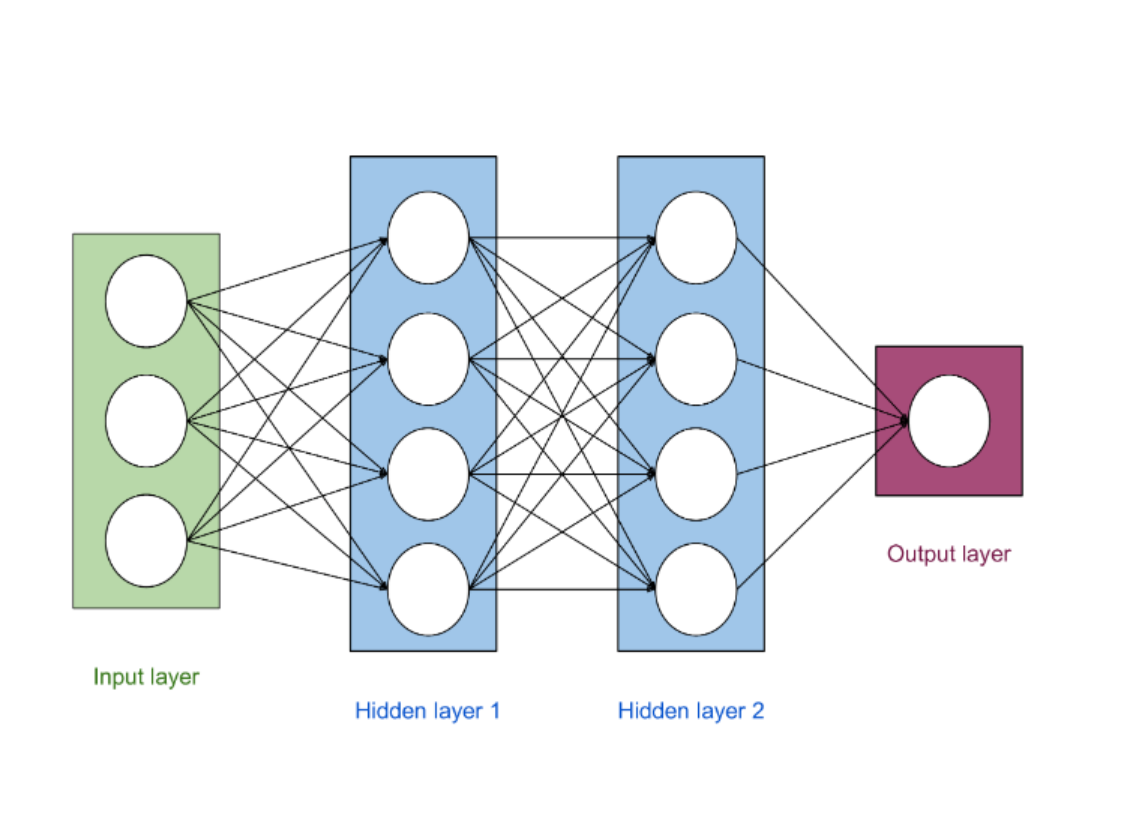
Biologically inspired Neural Network (Source)

Artificial Neural Network

Inspired by the biological nervous system, the structure of an **Artificial Neural Network (ANN)** was developed to process information similar to how brain process information. A large number of highly interconnected processing elements (neurons) working together makes a Neural Network solve complex problems. Just like humans learn by example, so does a Neural Network. Learning in biological systems involves adjustments to the synaptic connections which is similar to weight updates in a

Neural Network.

A Neural Network consists of three layers: **input layer** to feed the data to the model to learn representation, a **hidden layer** that learns the representation and the **output layer** that outputs the results or predictions. Neural Networks can be thought of an end to end system that finds patterns in data which are too complex to be recognized by a human to teach to a machine.



Neural Network with two hidden layers

Convolutional Neural Network

Hubel and **Wiesel** in their hierarchy model mentioned a neural network to have a hierarchy structure in the visual cortex. LGB (lateral geniculate body) forms the simple cells that form the complex cells which form the lower order hypercomplex cells that finally form the higher-order hypercomplex cells.
Also, the network between the lower order hypercomplex cells and the higher-order hypercomplex cells are structurally similar

to the network between simple cells and the complex cells. In this hierarchy, a cell in a higher stage generally has a tendency to respond selectively to a more complicated feature of the stimulus pattern, and the cell at the lower stage responds to simpler features. Also, higher stage cells possess a larger receptive field and are more insensitive to the shift in the position of the stimulus pattern.

Similar to a hierarchy model, a neural network starting layers learns simpler features like edges and corners and subsequent layers learn complex features like colours, textures and so on. Also, higher neural units possess a larger receptive field which builds over the initial layers. However, unlike in multilayer perceptron where all neurons from one layer are connected with all the neurons in the next layer, weight sharing is the main idea behind a convolutional neural network.
Example: instead of each neuron having a different weight for each pixel of the *input image (28*28 weights)*, the neurons only have a small set of *weights (5*5)* that is applied to a whole bunch of small subsets of the image of the same size. Layers past the first layer work in a similar way by taking in the 'local' features found in the previously hidden layer rather than pixel images, and successively see larger portions of the image since they are combining information about increasingly larger subsets of the image. Finally, the final layer makes the correct prediction for the output class.

The reason for why this is helpful is intuitive if not mathematically clear: without such constraints, the network would have to learn the same simple things (such as detecting edges, corners, etc) a whole bunch of times for each portion of the image. But with the constraint there, only one neuron would need to learn each simple feature, and with far fewer weights overall, it could do so much faster! Moreover, since the pixel-exact locations of such features do not matter the neuron could basically skip neighbouring subsets of the image — subsampling, now known as a type of pooling — when applying the weights, further reducing the training time. The addition of these two types of layers — **convolutional** and **pooling** layers — are the primary distinctions of Convolutional Neural Nets (CNNs/ConvNets) from plain old neural nets.

Probabilistic Machine Learning

To explain **Probabilistic Machine Learning** in simple words, let's break the term into **Probabilistic** and **Machine Learning**.

Machine Learning simply is to develop some algorithms that perform certain tasks given some data. It can range from finding a pattern in unstructured data to classifying emails, from language understanding to autonomous cars. Based on the observed data, some inference is made by a machine learning method. A model is trained which learns some patterns and assumptions from the observed data (training data) and makes inference on the unobserved data (test data). Since every inference comes with prediction confidence, a conclusion is derived. However, the model can be uncertain about the prediction due to numerous reasons: input data noise, sensory noise, measurement errors, non-optimal hyper-parameters settings and so on.

The **probabilistic** model in Machine Learning states that all forms of uncertainty cannot be a true value but more like a probability value and uses probability theory to answer everything. Probabilistic distributions are used to model the learning, uncertainty, and unobserved states. A prior probability distribution is defined before observing the data, the learning happens and the distribution transforms into posterior distributions once the data is observed. Using probability theory to learn from data forms the base for Bayesian Learning.

Uncertainties play a major role in Bayesian learning and let's take a detailed look into the type of uncertainties:

Uncertainties in Bayesian Learning

Uncertainties in a network is a measure of how certain the model is with its prediction. In Bayesian modelling, there are two main types of uncertainties: **Aleatoric** uncertainty and **Epistemic** uncertainty.

Aleatoric uncertainty measures the noise inherent in the observations. This type of uncertainty is present in the data collection method like the sensor noise or motion noise which is uniform along with the dataset. The uncertainty cannot be reduced even if more data is collected.

Epistemic uncertainty represents the uncertainty caused by the model itself. This uncertainty can be reduced given more data and is often referred to as model uncertainty. Aleatoric uncertainty can further be categorized into **homoscedastic** uncertainty, the uncertainty which stays constant for different inputs, and **heteroscedastic** uncertainty which depends on the inputs to the model, with some inputs potentially having more noisy outputs than others. **Heteroscedastic** uncertainty is in particular important so that model prevents from outputting very confident decisions.

Uncertainties can be estimated by placing a probability distributions over either the model parameters or model outputs. **Epistemic** uncertainty is modelled by placing a prior distribution over a model's weights and then trying to capture how much these weights vary given some data. **Aleatoric** uncertainty, on the other hand, is modelled by placing a distribution over the output of the model.

Now, we have a fair idea of probabilistic machine learning fundamentals, Bayesian learning, and Neural Networks. Combining Bayesian approaches to Neural Networks looks like a lucrative idea but in practice, it is hard to train a Bayesian Neural Network. The most popular approach to train a Neural Network is **backpropagation** and we use Bayes by Backprop to train the Bayesian Neural Networks. Let's take a look into the methods in details.

Backpropagation

Backpropagation in a Neural Networks was proposed by *Rumelhart* in 1986 and it is the most commonly used method for training neural networks. Backpropagation is a technique to compute the gradient of the loss in terms of the network weights. It operates in two phases: **firstly**, the input features through the network propagates in the forward direction to compute the function output and thereby the loss associated with the parameters. **Secondly**, the derivatives of the training loss with respect to the weights
are propagated back from the output layer towards the input layers.
These computed derivatives are further used to update the weights of the network. This is a continuous process and updating of the weight occurs continuously over every iteration.

Despite the popularity of backpropagation, there are many hyperparameters in backpropagation based stochastic optimization that requires specific tuning, e.g., learning rate, momentum, weight decay, etc. The time required for finding the optimal values is proportional to the data size. For a network trained with backpropagation, only point estimates of the weights are achieved in the network. As a result, these networks make overconfident predictions and do not account for uncertainty in the parameters. Lack of uncertainty measure makes the network prone to overfitting and a need for regularization.

A Bayesian approach to Neural Networks provides the shortcomings with the backpropagation approach as Bayesian methods naturally account for uncertainty in parameter estimates and can propagate this uncertainty into predictions. Also, averaging over parameter values instead of just choosing single point estimates makes the model robust to overfitting.

Several approaches have been proposed in the past for learning in Bayesian Networks: *Laplace approximation*, *MC Dropout*, and *Variational Inference*. We used Bayes by Backprop for our work and is explained next.

Bayes by Backprop

Bayes by Backprop was introduced was *Blundell*, et al. (cite) for learning a probability distribution on the weights of a neural network. The whole of this approach can be summarized as follows:

> *Instead of training a single network, the proposed method trains an ensemble of networks, where each network has its weights drawn from a shared, learned probability distribution. Unlike other ensemble methods, the method typically only doubles the number of parameters yet trains an infinite ensemble using unbiased Monte Carlo estimates of the gradients.*

In general, exact Bayesian inference on the weights of a neural network is intractable as the number of parameters is very large and the functional form of a neural network does not lend itself to exact integration. Instead, we take a **variational approximation** rather than a Monte Carlo scheme to find the **approximate Bayesian posterior distribution**.

OK! It makes sense. So let's dive in a bit more, as this method forms the base of our method that will be explained in the later blogs. Let us begin with understanding why a distribution gets intractable and the need for approximating it. Let us begin with Bayes theorem:

$$P(\theta|x) = \frac{P(x|\theta)P(\theta)}{P(x)}$$

Bayes theorem

From **Bayes theorem**, as stated above, we try to find the probability of our model parameters **θ** given some data **x**. This is known as **posterior** and we want to compute it. Now the numerator has **P(θ)** which is our **prior** (estimation before seeing the data) and **P(x|θ)** which is the **likelihood** and shows the data distribution. Both of these values are easy to compute. The denominator **P(x)** is the **evidence** and it shows if the data **x** is generated from the model. Now, this makes things tricky. We can only compute this by integrating over all possible model values:

$$P(x) = \int P(x,\theta)d\theta)$$

As we can see, this makes the whole solution **intractable** and the only way to solve it is to **approximate** it. So we will use a **variational inference** to approximate the functional form.

> There are other methods that can be used to approximate the integrals and the popular ones are Markov Chain Monte Carlo and Monte Carlo Dropout.

Variational Inference

Assume we have any density function and we want to estimate it. We start with choosing a distribution (can be Gaussian as it is the most popular one) and we keep on changing it until we reach the very close to the desired function, our posterior in this case. We want to reach as close as possible to the true distribution which is intractable and we can do that by **minimizing the Kullback-Liebler (KL) divergence** between the two.

So, we have a function **P(w|D)** (our posterior from above) and we want to approximate it with another distribution **q(w|D)** with some **variational parameters θ**.

> Note that the symbols have changed here to keep it consistent with blog Probabilistic Deep Learning: Bayes by Backprop that explains the topic very well.

KL divergence makes the problem an optimization problem and can be minimized as:

$$\theta_{opt} = \arg\min_{\theta} \mathrm{KL}\Big[q_\theta(w|\mathcal{D})||p(w|\mathcal{D})\Big]$$

A well explained diagram showing how to approximate the intractable posterior. Source

But this is not the end. If we solve the KL divergence, it gives an **intractable** equation again due to the presence of an integral function:

$$\begin{aligned}\theta^{opt} &= \underset{\theta}{\arg\min}\ \mathrm{KL}\left[q_\theta(w|\mathcal{D}) \| p(w|\mathcal{D})\right] \\ &= \underset{\theta}{\arg\min}\ \mathrm{KL}\left[q_\theta(w|\mathcal{D}) \| p(w)\right] \\ &- \mathbb{E}_{q(w|\theta)}[\log p(\mathcal{D}|w)] + \log p(\mathcal{D})\end{aligned}$$

$$\mathrm{KL}\left[q_\theta(w|\mathcal{D}) \| p(w)\right] = \int q_\theta(w|\mathcal{D}) \log \frac{q_\theta(w|\mathcal{D})}{p(w)} dw.$$

Source

Now we cannot approximate an approximated function already (Technically we can but it will again need to be as close as to the original function and hence KL divergence to find the distance and again the integration problem). So we can **sample** from the approximated function **q(w|D)** as it is much easier to sample weights from an **approximated function q(w|D)** than an intractable **true posterior function p(w|D)**. Upon doing so we reach to a tractable function as stated below:

$$\mathscr{F}(\mathscr{D},\theta) \approx \sum_{i=1}^{n} \log q_\theta(w^{(i)}|\mathscr{D}) - \log p(w^{(i)}) - \log p(\mathscr{D}|w^{(i)})$$

These sampled weights **w**, are used in the backpropagation of the neural network to learn the **posterior distribution**.

Now, as we can see it is possible to train the Bayesian Neural Network by **Bayes by Backprop** approach and Bayesian Neural Network incorporates regularization automatically. We will learn in the next blogs the Bayesian Convolutional Neural Networks using variational inference approach. We use two operations in convolutions (more details in coming blogs or read here) and hence the number of parameters doubles up for a Bayesian CNN compared to a point estimate based CNN. Hence, to reduce the network parameters, we pruned the network architecture and let's see how it can be done.

Model Weights Pruning

Model pruning reduces the sparsity in a deep neural network's various connection matrices, thereby reducing the number of valued parameters in the model. The whole idea of model pruning is to reduce the number of parameters without much loss in the accuracy of the model. This reduces the use of a large parameterized model with regularization and promotes the use of dense connected smaller models. Some recent work suggests that the network can achieve a sizable reduction in model size, yet achieving comparable accuracy.
Model pruning possesses several advantages in terms of reduction in computational cost, inference time and in energy-efficiency. The resulting pruned model typically has sparse connection matrices. An efficient inference using these sparse models requires purpose-built hardware capable of loading

sparse matrices and/or performing sparse matrix-vector operations. However, the overall memory usage is reduced with the new pruned model.

There are several ways of achieving the pruned model, the most popular one is to map the low contributing weights to zero and reducing the number of overall non-zero valued weights. This can be achieved by training a large sparse model and pruning it further which makes it comparable to training a small dense model.

Assigning weights zero to most features and non-zero weights to only important features can be formalized by applying the L_0 **(L-zero) norm**, as it applies a constant penalty to all non-zero weights.

L_0 norm can be thought of a feature selector norm that only assigns non-zero values to features that are important.

However, the **L_0 norm** is **non-convex** and hence, **non-differentiable** that makes it **an NP-hard problem** and can be only efficiently solved when *P = NP*.

The alternative to L_0 norm is the **L_1 norm**, which is equal to the sum of the absolute weight values. **L_1 norm** is **convex** and hence **differentiable** and can be used as an **approximation to L_0 norm.** L_1 norm works as a sparsity-inducing regularizer by making a large number of coefficients equal to zero, working as a great feature selector.

This blog was just to provide background knowledge of the terms and the concepts that will be used in the future blogs and if I missed something, please let me know.

If you want to read things in advance, check the thesis work, or the paper.

Implementation in PyTorch is available here.

Feel free to comment on any ideas, feedback or find me here.

For more blogs on the related topic, check out: