BAYESIAN NEURAL NETWORK

LEANNE DONG

1. Review of Bayesian Inference

1.1. **Bayesian vs Frequentist.**

1.1.1. *Frequentist.*

- Focus is on **the parameter**, which is assumed to be a fixed constant
- Confidence intervals are read in terms of **repeated sampling**

"95% of similar sized intervals from repeated chance that θ exists within the interval"

1.1.2. *Bayesian.*

- Focus is on **subjective probability**, taking into account a priori predictions
- Credible intervals read in terms of **subjective uncertainty**

"There is a 95% chance that θ exists within the interval"

1.2. **Bayes Theorem.**
Bayes Theorem updates our Prior intuition about the paramemters with information we gather from sample. And then the question becomes how much do we weight our prior intuition vs how much do we weight the information we get from our sample.

$$[\theta|\text{data}] = \frac{[\text{data}|\theta] * [\theta]}{[\text{data}]}$$

Here, $[\theta|\text{data}]$ is the *posterior* distribution. $[\text{data}|\theta]$ is the likelihood. $[\theta]$ is the prior. $[\text{data}]$ is the normalising constant to make sure our Posterior sum to one.

Since our data is independent to θ, equation above can be represented as $[\theta|\text{data}] \propto [\text{data}|\theta][\theta]$

1.3. **Visual Example.**

1.4. **Bayesian Inference for a normal mean.**
Consider a single observation is taken from a normal distribution with mean θ (and known σ)

$$[y|\theta] = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\left(\frac{1}{2\sigma^2}\right)(y-\theta)^2\right\}$$

Recall the Bayesian relationship:

$$[\theta|y] \propto [y|\theta[\theta]$$

Consider a **prior** of the following form:

$$[\theta] \propto \exp\left\{-\left(\frac{1}{2\tau^2}\right)(y-\theta)^2\right\} \qquad \theta \sim N(\mu_0, \tau_0^2)$$

The posterior will therefore be:

$$[\theta|y] \propto \exp\left\{-\frac{1}{2}\left(\frac{(y-\theta)^2}{\sigma^2} + \frac{(\theta-\mu_0)^2}{\tau_0^2}\right)\right\}$$

Date: February 15, 2019.

$$[\theta|y] \propto \exp\left\{-\frac{(\theta-\mu_1)^2}{2\tau_1^2}\right\}$$

where

$$\mu_1 = \frac{(\mu_0/\tau_0^2)+(y/\sigma^2)}{(1/\tau_0^2)+(1/\sigma^2)}, \quad \frac{1}{\tau_1^2} = \frac{1}{\tau_0^2}+\frac{1}{\sigma^2}$$

1.5. **Conjugate priors.** Let us reconsider the Bayesian equation:

$$[\theta|y] \propto [y|\theta][\theta]$$

We often choose convenient parametric forms for our priors, such that the posterior remains manageable.

Where the prior's parametric form is retained through to the posterior, it is called a conjugate prior.

Likelihood	Conjugate prior
Normal	Normal
Binomial/Bernoulli	Beta
Exponential/Poisson	Gamma
Uniform	Pareto
Multinomial	Dirichlet

The beta prior Consider the binomial likelihood function:

$$[y|\theta] \propto \theta^y(1-\theta)^{n-y}$$

Now consider a prior of the form:

$$[\theta] = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}$$

where

$$B(\alpha,\beta) = \int_0^1 \theta^{\alpha-1}(1-\theta)^{\beta-1}d\theta$$

The posterior becomes

$$[\theta|y] \propto \theta^{y+\alpha-1}(1-\theta)^{n-y+\beta-1}$$

So, the posterior is still in the same form as our prior. Essentially, the prior distribution is essentially analagous to having

- $\alpha - 1$ additional individuals with disease
- $\beta - 1$ additional individuals without the disease

1.6. **Credible intervals.** This is the Bayesian equivalence of confidance interval.

2. Markov Chain Monte Carlo

We are going to cover the Markov chain Monte Carlo which is kind of a silver bullet of probabilistic programming, in a sense it allows you to train or to do imprints on almost any model without too much trouble. We will discuss some extensions of this technique. So how to make it faster in some particular problems, by exposing the structure of the problems, and we'll also discuss some limitations, like the speed of the method.

- MCMC - Silver bullet of probabilistic modelling
- Learn how exploit specifics of your problem to speed up MCMC

- Understand the limitations

2.1. **Monte Carlo.** So, let's start the discussion of Markov chain Monte Carlo with the Monte Carlo part. Monte Carlo was originated in the Manhattan Project which gave us the atomic bomb and it was a quick and dirty answer to the question, what to do until the mathematicians arrives. The idea of Monte Carlo simulation is to, so if you have a complicated model of some problem and you want to predict something about this model, instead of thinking carefully and, you know, writing down equations and solving them, you may just simulate your model many times and then see, for example, the average of the simulation as some kind of smooth prediction of what will happen. So, probably the reason why this thing was invented in Manhattan Project is because it's one of the first really huge scientific projects where we already had lots of means to approximate integrals and to do these kind of simulations. And second of all, we already had computers in some reasonably useful state, where we can actually assimilate stuff, not mentally but automatically. And the Monte Carlo algorithm turned out to be pretty efficient, and it made it to the list of top 10 algorithms of 20th century. So the name Monte Carlo was given by the name of the city Monte Carlo which is famous for its casinos, and probably because, you know, everything in Manhattan Project had to have its code name. So let's see a small example of what I'm talking about. So, what is Monte Carlo applied to some really simple problem. Say you want to approximate the value of π, and you know that π is the area of the unit circle, a circle with radius one. So we can, kind of, use this information to approximate it. And the idea here is that if you consider a rectangle from 0 to 1 on both axes, then this quarter of a circle which appears in this rectangle, has exactly the area pi divided by four. It's just one-fourth of the circle. And its area equals to some integral which basically says at each point whether this point is in the circle or not. So if we integrate this thing along this rectangle, we'll get the area. So how can we compute this integral? It's hard. But we can throw lots and lots of points on this unit rectangle and just see the fraction of those points that are put in the circle, in this quarter of the circle, and use this as an approximation to what the integral really is. So, you can see that this maybe not the best method to approximate the value of π because here, for example, you spend like 3000 samples. So you assemble 3000 points and the pi was approximate as 3.1, which is not the best accuracy you can get with this much computations.

$$\begin{aligned}\frac{\pi}{4} &= \mathbb{E}(x^2 + y^2 \leq 1)\\ &\approx \frac{1}{M}\sum_{s=1}^{M}(x_s^2 + y_s^2 \leq 1)\end{aligned}$$

$$x_s, y_s \sim \mathcal{U}(0, 1)$$

But anyway, the method is really simple. It's like two lines of code in almost any programming language. And this is the general theme of Monte Carlo simulations. So, Monte Carlo usually gives you something which is really easy to program, which is really universally applicable to lots of and lots of problems, which is very scalable to parallelization. So, if you have like 100 computers, then you can really easily parallelize this thing to be 100 times more efficient which is not the case for many other algorithms which are much harder to parallelize. And sometimes this Monte Carlo method can be slow compared to other alternatives. So, usually, if you sit down carefully for a weekend, like write down lots of equations and think, then you may come up with a better way to do, to solve your problem than to do just Monte Carlo simulation. Sometimes not, but sometimes it's just a quick and dirty approach which gives you the first approximation. But anyway, it's very useful. So, to be a little bit more general about this family of techniques, let's say that we want to approximate some expected value of some function f, with respect to some distribution $[x]$. And we will use, we will sample a few points for some end points from this distribution $[x]$, and use the average of size as the approximation. So, here, we are kind of using sampled average instead of the expected value. And this thing has lots of nice various co-properties like it's unbiased, meaning that if you sample enough points, you can get closer and closer to the actual true answer, at least with high probability. And anyway, we're interested in knowing how fast you will converge to that. So, this is Monte Carlo.

$$\mathbb{E}_{[x]} f(x) = \frac{1}{M} \sum_{s=1}^{M} f(x_s)$$

$$x_s \sim p(x)$$

And you may ask a question like, why do we care? Why do we need to approximate expected values? Well, in probabilistic programming, there are probably a few reasons to do that. Let's consider two of them. So first of all, if you wanted to do full Bayesian inference, like we covered a little bit in week one, then, instead of having some pragmatic model and finding the best failure of parameters, you may just want to tweak your parameters as latent variables. And then for a new object X, if you want to predict it's label, for example, and you have some training data set x train and y train. You may want to just integrate out the latent variable like this.

- Full Bayesian inference

$$\begin{aligned} &[y|x, Y_{\text{train}}, X_{\text{train}}] \\ = &\int [y|x, w][w|Y_{\text{train}}, X_{\text{train}}]dw \end{aligned}$$

So, imagine that, for example, x is image and you want to predict like whether it's an image of a cat or dog. And then $[y|x]$ and w maybe for example a neural network. With weights w and which takes as inputs this image x and outputs your distribution over labels y. And then, instead of using this just one neural network with some kind of optimal or best set of weights w you're considering all possible neural networks with this architecture. So, for each possible value for weight's w, you consider this neural network. You pass your image through this neural network. You write down the answers the $[y]$ and then you average all these predictions with some weights which are given by the Pasteur's division the weights $[w]$ given the training data set. So it's like an infinitely large ensemble of neural networks. Where the weights of the ensemble are given by the Pasteur's distribution $[w]$ given the training data set. And this $[w|Y_{\text{train}}, X_{\text{train}}]$ gives you some idea about how well these weights will behave as a weight for your neural network. So this value will be higher for reasonable weights and lower for the weights which doesn't make sense. So this way, you're doing full base in inference. You're having a few benefits of probabilistic modeling like you can, for example, predict uncertainty in your predictions. And well basically, instead of just fixed value of weights, you have a distribution over weights. And this thing is, of course, intractable. So if you have a complicated function like in neural network here, you can compute this integral analytically. So you have to do some approximation. And one of the way to do it is to just say that this thing is just an expected value of this neural network. So we have a neural network output and the computing the expected value of this neural network output with respect to some distributional weights. And so, we can approximate this thing with Monte Carlo. If we can sample from $[w|Y_{\text{train}}, X_{\text{train}}]$. So this is one example, where the Monte Carlo can be useful to approximate expected value in the probabilistic modeling. And you may ask how we can find this posterior distribution of the weights $[w]$ given the training data set?

$$[w|Y_{\text{train}}, X_{\text{train}}] = \frac{[Y_{\text{train}}|X_{\text{train}}, w][w]}{Z}$$

Well the posterior distribution is proportional to the joint distribution. So likelihood P of Y train given X train and W times the prior and W and divided by some normalization constant. And so the numerator here is easy because you defined your model yourself. You could have, for example, defined P of W the prior to be normal and the likelihood is just your output of your neural network. So you can compute this thing for any values of Y, X, and W. But then, the denominator, the normalization constant Z is much harder. It contains something to go on site so you can't ever compute it. So it's kind of a general situation where you know your distribution from which you want to sample but only up to normalization constant. Right? And other example of these kind of expected values approximation can be useful in probabilistic modeling is the M-step of the expectation maximization algorithm which we covered in week two. So, on the E step of expectation maximization, we found the posterior distribution latent variable T given the data set and parameters. And on the M-step, we want to maximize the expected value of algorithm of the joint probability. With respect to the parameters theta and the expected value with respect to this posterior distribution Q.

$$\max_{\theta} \mathbb{E}_q \log[X, T|\theta]$$

So, here again, if the posterior distribution Q is too hard to work with, and we can't integrate this thing analytically, we can do a few things. We can, first of all, we can approximate Q to be some to lay in some simple class like factorize distribution. And that's what we did in the previous week, week three about variational inference. So we're approximating the posterior and then computing these integral analytically for the simplified posterior. Or we can use the exact posterior but we can sample some data set of samples from this set of latent variable values and then approximate this expected value with just average with respect to these samples and then maximize this approximation instead of the true expected values. So these are two examples of where you can need expected value in probabilistic modeling and how Monte Carlo simulation can help you here. And in the following videos, we are going to use just, we're going to use the formulation of this problem is as like we want to learn how to sample from a complicated probabilistic distribution which we know up to normalization constant. And we will always write like $[x]$ meaning that x is the parameters you want to sample. So here we had, in the first example of full Bayesin interference we had $[x|\text{data}]$. And in this second example we have $[\text{latent variable}|\text{data}]$. But in all cases, the full ingredients, we will just write down, you know, $[x]$ meaning that it sample distribution we want to sample from.

3. Markov Chain

Too elementary

4. Gibbs Sampling

$$p(x_1, x_2, x_3) = \frac{\widehat{p}(x_1, x_2, x_3)}{Z}$$

Start with (x_1^0, x_2^0, x_3^0), e.g. $(0, 0, 0)$

For $k = 0, 1, \cdots$

$$x_1^{k+1} \sim p(x_1|x_2 = x_2^k, x_3 = x_3^k)$$

$$x_2^{k+1} \sim p(x_2 | x_1 = x_1^{k+1}, x_3 = x_3^{k})$$

$$x_3^{k+1} \sim p(x_3 | x_1 = x_1^{k+1}, x_2 = x_2^{k+1})$$

Proposition 4.1.

$$\sum_{x,y,z} [x, y, z \to x', y', z'][x, y, z] \overset{?}{=} [x', y', z']$$

$$\begin{aligned}
&\sum_{x,y,z} [x'|y = y, z = z][y'|x = x', z = z][z'|x = x', y = y'] * [x, y, z] \\
&= [z'|x', y'] \sum_{y,z} ([x'|y, z][y'|x', z] \underbrace{\sum_{x} [x, y, z]}_{[y,z]}) \\
&= [z'|x', y'] \sum_{z} [y'|x', z] \underbrace{\sum_{y} [x', y, z]}_{[x',z]}
\end{aligned}$$

So let's prove that the Gibbs sampling over the three sub-steps, considered as one big step, indeed provides you a Markov chain that converged to the desired distribution p. So what we want to prove is that the distribution of the new point, x', y', and z', equals, so we want to prove that it equals, to the one step for the Gibbs sampling, starting from the distribution p, and then doing one step of the sampling. So here we'll have marginalizing out the previous step, x, y, and z. And the transition probability q, probability to go from x, y, z, to x', y', z'. And times the joint distribution. To prove the equality, Let us look at the right hand side of this expression. So we have sum with respect to the previous point of the transition probability. So how probable is it, under the Gibbs sampling, to move from x, y, z, to x', y', z'? Well, we have to first of all move from x to x', so that will be least conditional probability, $y = y$, and $z = z$. Then we have to assume that we move to x', and then what is the probability that we move from y to y prime? So it'll be y', given that we have already moved to x', and started with z. And finally, the third conditional probability, that we move to z prime, given that we already moved to x' y'. So this is the transition probability, this overall thing, and times the starting probability, $[x, y, z]$. So times $[x, y, z]$. And we want to prove that this thing equals to the $[x', y', z']$, because in this case, we will prove that this p is stationary for our Markov chain. And then, it means that the Markov chain converged to this distribution, p. So first of all, it's not that this part, $[z']$, it doesn't depend on x, y or z at all. So we can move it outside the sum. It'll be $[z|x', y']$ times sum of these two terms. And then, times $[x, y, z]$. And note here that, actually, the only part that depends on x is this one. So it's $[x, y, z]$, which basically means that we can push the summation aside. We can write here the sum with respect to only y and z, and write the summations with respect to x here. So it is summation with respect to all values of x, from 1 to the cardinality of x. And also, by the way, note that if our variables x, y, and z would be continuous, we will have integrations instead of sums. But the overall logic of the proof will be exactly the same. So, we have this expression. These two are equal to each other. And you can note that this part, we have just marginalized out x. So we end up with $[y, z]$, the marginal distribution $[y, z]$. Great, now we can multiply this part by $[x'|y, z]$ to get a joint distribution. So this thing will equal to, Again, this part times sum, we'll have...

Let's start with this term, $[y'|x', z]$, times the product of these two joints, which is the joint distribution. And note that the only part that depends on y is this joint distribution. So we can move this summation again, with respect to y, inside and right down here, summation only with respect to z, and here the summation with respect to y. So it will be $[x', y, z]$, which is a product of these two terms, okay? And again, note that when we marginalize out y here, we end of with $[x', z]$. And again, we can multiply these two terms together to get a joint distribution. So finally, we have something like this. It's, again, the first term, $[z'|x', y']$, times summation with respect to z of $[y'|x', z]$, times the joint distribution. So it's just the full joint distribution of these three terms, y', x', z.

And these parts, since we marginalized out z, again, it equals to $[y', z']$, by the definition of the marginalization. So it's like a rule of sum of probabilities. And finally, by multiplying these two terms together, we end up with what we wanted. So this thing equals to $[z', x', y']$. So we're done here. We have just proven that if you start with distribution p on the step x, y, z, and then make one step of this Gibbs sampling procedure, one coordinates at a time, we'll end up with the same distribution p, $[x', y', z']$. Which basically means that this distribution beam's stationary for the Gibbs sampling. And that's what will converge to this distribution from any starting point. And basically, it means that if we use Gibbs sampling, it, indeed, will eventually start to produce us samples from the distribution p.

4.1. **Gibbs Sampling.** Here we skip some examples but discuss simply some pros and cons of Gibbs Sampling.

Pros:

- Reduce multidimensional sampling to sequence of samplings
- A few lines of code

Cons:

- Highly correlated samples
- Slow convergence (mixing)
- Not parallel

5. Metropolis-Hasting

As we see earlier, building a MC via Gibbs sampling produces highly correlated samples and it converges somewhat slowly to the distribution. Both of these downsides source from the property that Gibbs is doing kind of local and small steps in sample space. In this section, we will discuss some other scheme to build MC, which instead of producing just one predefined MC, will give us a whole family of MC. And one can choose the one that has the desired properties like, it converges faster and maybe it produces less correlated samples. This family of techniques is called Metropolis-Hastings and the idea is to apply the rejection sampling idea, two Markov chains.

```
For $k = 1, 2, \cdots$
    • Sample $x'$ from a wrong $Q\ (x^k \to x')$
    • Accept $x'$ with probability $A(x^k \to x')$
    • Otherwise stay at $x^k$
          $x^{k+1} = x^k$
```

$$T(x \to x') = Q(x \to x')A(x \to x') \quad \text{for all} \quad x \neq x'$$

$$T(x \to x) = Q(x \to x)A(x \to x) \sum_{x' \neq x} Q(x \to x')(1 - A(x \to x')) \quad \text{for all} \quad x \neq x'$$

How to choose $A : \pi(x') = \sum_x \pi(x)T(x \to x')$

Detail Balance

Proposition 5.1.

$$\textit{If} \quad \pi(x)T(x \to x') = \pi(x')T(x' \to x)$$

$$\textit{Then} \quad \pi(x') = \sum_x \pi(x)T(x \to x')$$

Proof.

$$\begin{aligned} \sum_x \pi(x)T(x \to x') &= \sum_x \pi(x')T(x' \to x) \\ &= \pi(x') \underbrace{\sum_x T(x' \to x)}_{1} \\ &= \pi(x') \end{aligned}$$

□

The following summarise Metropolis-Hasting algorithm.

For $k = 1, 2, \cdots$
- Sample x' from a wrong Q $(x^k \to x')$
- Accept x' with probability $A(x^k \to x')$
- Otherwise stay at x^k

$$x^{k+1} = x^k$$

$$T(x \to x') = Q(x \to x')A(x \to x') \quad \text{for all} \quad x \neq x'$$

$$T(x' \to x') = Q(x' \to x')$$

How to choose $A : \pi(x') = \sum_x \pi(x)T(x \to x')$

5.1. Choosing the critic.

$$\pi(x) \underbrace{Q(x \to x')A(x \to x')}_{T(x \to x')} = \pi(x') \underbrace{Q(x' \to x)A(x' \to x)}_{T(x' \to x)}$$

$$\frac{A(x \to x')}{A(x' \to x)} = \frac{\pi(x')Q(x' \to x)}{\pi(x)Q(x \to x')} = \rho < 1$$

$$A(x \to x') = \rho$$

$$A(x' \to x) = 1$$

So we can set

$$A(x \to x') = \min\left\{1, \frac{\pi(x')Q(x' \to x)}{\pi(x)Q(x \to x')}\right\}$$

To summerise the Metropolis Hastings approach to build a Markov Chain.
For $k = 1, 2, \cdots$

- sample x' from a wrong Q $(x^k \to x')$
- Accept proposal x' with probability $A(x^k \to x')$
- Otherwise stay at x^k

$$x^{k+1} = x^k$$

$$A(x \to x') = \min\left\{1, \frac{\hat{\pi}(x')Q(x' \to x)}{\hat{\pi}(x)Q(x \to x')}\right\}$$

We should choose Q to be strictly positive

$$Q(x \to x') > 0$$

Here are some Opposing forces

- Q should spread out, to improve mixing and reduce correlation
- But then acceptance probability is often low.

5.2. **Example of Metropolis-Hastings.** Here we sample from

$$Q(x \to x') = \mathcal{N}(x, 1)$$

$$A(x \to x') = \min\left(1, \frac{\pi(x')Q(x' \to x)}{\pi(x)Q(x \to x')}\right) = \min\left(1, \frac{\pi(x')}{\pi(x)}\right)$$

$$A(x \to x') = \min\left(1, \frac{0.27}{0.07}\right) = \min(1, 3.87)$$

In summary, rejection sampling applied to MCs. Here are some Pros and Cons
Pros

- You can choose among family of MCs
- Works for unnormalized densities
- Easy to implement

Cons

- Samples are still correlated
- Have to choose among family of MCs

5.3. **MCMC summary. Pros**

- Easy to implement
- Easy to parallelize
- Unbiased estimates (wait more, more accuracy)

Cons

- Usually slower than alternatives

6. MCMC for LDA

7. Bayesian Neural Network

Let's discuss how can we apply Markov chain Monte Carlo to Bayesian Neural Networks. Essentially, BNN is your usual neural network, and it has weights on each h, right? So each connection has some weights which would train during basically fitting our neural network into data. Basing neural networks instead of weights, they have distributions and weights. So we treat w, the weights, as a latent variable, and then to do predictions, we marginalize w out. And this way, instead of just hard set failure for W11 like three, we'll have a distribution on w in posterior distribution which we'll use to obtain the predictions. And so, to make a prediction for new data object x or and though using the training data set of objects X_{train} and Y_{train}, we do the following. We say that this thing equals to integral where we marginalize our w. So we consider all possible values for the weights w, and we average the predictions with the respect to them. So here you have p of y given x and w, is your usual neural network output. So, you have your image x, for example, and you pass it through your neural network with parameters w. And then you record these predictions. And you do that for all possible values for the parameters w. So there are infinitely many values for W, and for each of them you pass your image through the corresponding neural network, and write down the prediction. And then you average all these predictions with weights, where weights are the posterior distribution on w, which basically says us, how probable is that these particular w was according to the training data set. So, you have kind of an infinitely large ensemble of neural networks with all possible weights, and with basically importance being proportional to the posterior distribution w. And this is full base in inference applied in neural networks, and this way we can get some benefits from probablistic programming in neural networks. So again, estimate uncertainty, we may tune some hyperparameters naturally and stuff like that. And so we may notice here that this prediction, this integral, equals to an expected value of your output from your neural network, with respect to the posterior distribution w. So, basically it's an expected output of your neural network with weights defined by the posterior.

$$\begin{aligned}
&[y|x, Y_{\text{train}}, X_{\text{train}}] \\
&= \int [y|x, w][w|Y_{\text{train}, X_{\text{train}}}]dw \\
&= \mathbb{E}_{[w|Y_{\text{train}}, X_{\text{train}}]}[y|x, w]
\end{aligned}$$

And so to solve this problem, let's use your favorite Markov chain Monte Carlo procedure. So let's approximate this expected value with sampling, for example with Gibbs sampling.

$$[w|Y_{\text{train}}, X_{\text{train}}] \sim \{\text{Gibbs}\}$$

And if require a few samples from the posterior distribution w, we can use that Ws, that weights of neural network and then, if we have like, for example, 10 samples. For each sample is a neural network, is a weights for some network. And then for new image, we can just pass it through all this 10 neural networks, and then average their predictions to get approximation of the full weight in inference with an integral. And how can we sample from the posterior? Well, we know it after normalization counts, as usually.

$$[w|Y_{\text{train}}, X_{\text{train}}] = \frac{[Y_{\text{train}}|X_{\text{train}}, w][w]}{Z}$$

So, here this posterior distribution W is proportional to the likelihood, so basically the prediction of a neural network on the training data set with parameters W times the prior $[w]$ which you can define as you wish, for example, just a standard normal distribution. And you have to divide by normalization constant, which you've done now. But it's okay because Gibbs sampling doesn't care, right? So it's a valid approach, but I think the problem here is that Gibbs sampling or Metropolis-Hastings sampling for that matter, it depends on the whole data set to make its steps, right? We discussed at the end of the previous video, that sometimes Gibbs sampling is okay with using mini-batches to make moves, but sometimes it's not. And as far as I know, in Bayesian neural networks, it's not a good idea to use Gibbs sampling with the mini-batches. So, we'll have to do something else. If we don't want to, you know, when we ran our Bayesian neural network on large data set, we don't want to spend time proportional to the size of the whole large data set or at each duration of training. Want to avoid that. So let's see what else can we do. And here comes the really nice idea of something called, Langevin Monte Carlo. So it forces false. Say, we want to sample from the posterior distribution $[w]$ given some data. So train in data X_{train} and Y_{train}. Let's start from some initial value for the base w, and then in iterations do updates like this. So here, we update our w to be our previous w, plus epsilon, which is kind of learning create, times gradient of our logarithm of the posterior, plus some random noise.

$$\begin{aligned} w^{k+1} &= w^k + \varepsilon \nabla \log[w^k|D] + \eta^k, \\ &= w^k + \varepsilon \nabla \left(\log[w^k] + \sum_{i=1}^{N} \log[y_i|x_i, w^k] \right) + \eta^k \end{aligned}$$

So the first part of this expression is actually a usual gradient ascent applied to train the weights of your neural network. And you can see it here clearly. So if you look at your posterior p of w given data, it will be proportional to logarithm of prior, plus logarithm of the condition distribution, p of y given x and w. And you can write it as follows by using the purpose of logarithm that, like logarithm of multiplication is sum of logarithms. And you should also have a normalization constant, here is that. But is a constant with respect to our optimization problem so we don't care about it, right? And on practice this first term, the prior, if you took a logarithm of a standard to normal distribution for example, it just gets some constant times the Euclidean norm of your weights w. So it's your usual weight decay which people oftenly use in neural networks. And the second term is, usual cross entropy. Usual objective that people use to train neural networks. So this particular update is actually a gradient descent or ascent with step size epsilon applied to your neural network to find the best possible values for parameters. But on each iteration, you add some Gaussian noise with variants being epsilon. So proportional to your learning crate. And if you do that, and if you choose your learning crate to be infinitely small, you can prove that this procedure will eventually generate your sample from the desired distribution, p of w given data. So basically, if you omit the noise, you will just have the usual gradient ascent. And if you use infinitely small learning crate, then you will definitely goes to just the local maximum around the current point, right? But if you add the noise in each iteration, theoretically you can end up in any point in the parameter space, like any point. But of course, with more probability, you will end up somewhere around the local maximum. If you're doing that, you will actually a sample from a posterior distribution. So you will end up in points with high probability of more often than in points with low probability. On practice, you will never use infinitely small learning crate, of course. But one thing you can do about it is to correct this scheme with Metropolis-Hastings. So you can say that theoretically, I should use infinitely small learning crate. I use not infinitely small but like .1, so I have to correct, I'm standing from the wrong distribution. And I can do Metropolis-Hastings correction to reject some of the moves and then to guarantee that I will sample from the current distribution. But, since we want to do some large scale optimization here and to work with mini-batches, we will not use this

Metropolis-Hastings corrections because it's not scalable, and we'll just use small learning crate and hope for the best. So this way, we will not actually derive samples from the true posterior distribution w, but will be close enough if your learning crate is small enough, is close enough to the infinitely small, right? So the overall scheme is false.

- Initialize weight w^0
- Do say 100 iterations with usual SGD, but add Gaussian noise $\eta^k \sim N(0, 2\varepsilon I)$ to each update
- After 100,00 epochs decide that MC converged and start collecting weights values
- For a new object predict compute average prediction of CNNs with weights $w^{100}, w^{101}, \cdot, w^{200}$.

We initialized some weights of our neural network, then we do a few iterations or epochs of your favorite SGD. But on each iteration, you add some noise, some Gaussian noise with a variance being equal to the learning crate, to your update. And notice here also that you can't change learning rate at all, at any stage of your symbolic or you will also break the properties of this Langevin Monte Carlo idea. And then after doing a few iterations like hundred of them, you may say that, okay, I believe that now I have already converged. So, let's collect the full learning samples and use them as actual samples from the posterior distribution. That's the usual idea of Monte Carlo. And then finally, for a new point you can just diverge the predictions of your hundred slightly different neural networks on these new objects to get the prediction for your object. But this is really expensive, right? So there is this really nice and cool idea that we can use a separate neural network that will approximate the behavior of these in sample. So we are simultaneously training these Bayesian neural network. And simultaneously with that, we're using its behavior to train a student neural network that will try to mimic the behavior of this Bayesian neural network in the usual one. And so it has quite a few details there on how to do it efficiently, but it's really cool. So if you're interested in these kind of things, check it out.