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. [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:

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

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 14, 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. 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 crate 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.