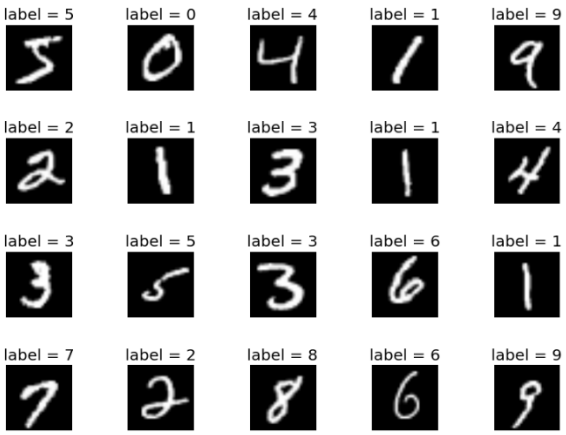
Making Your Neural Network Say “I Don’t Know” — Bayesian NNs using Pyro and PyTorch

Paras Chopra
Nov 27, 2018 · 17 min read

Building an image classifier has become the new “hello world”. Remember the day when you first came across Python and your *print “hello world”* felt magical? I got the same feeling a couple months back when I followed the PyTorch official tutorial and built myself a simple classifier that worked pretty well.



is the new

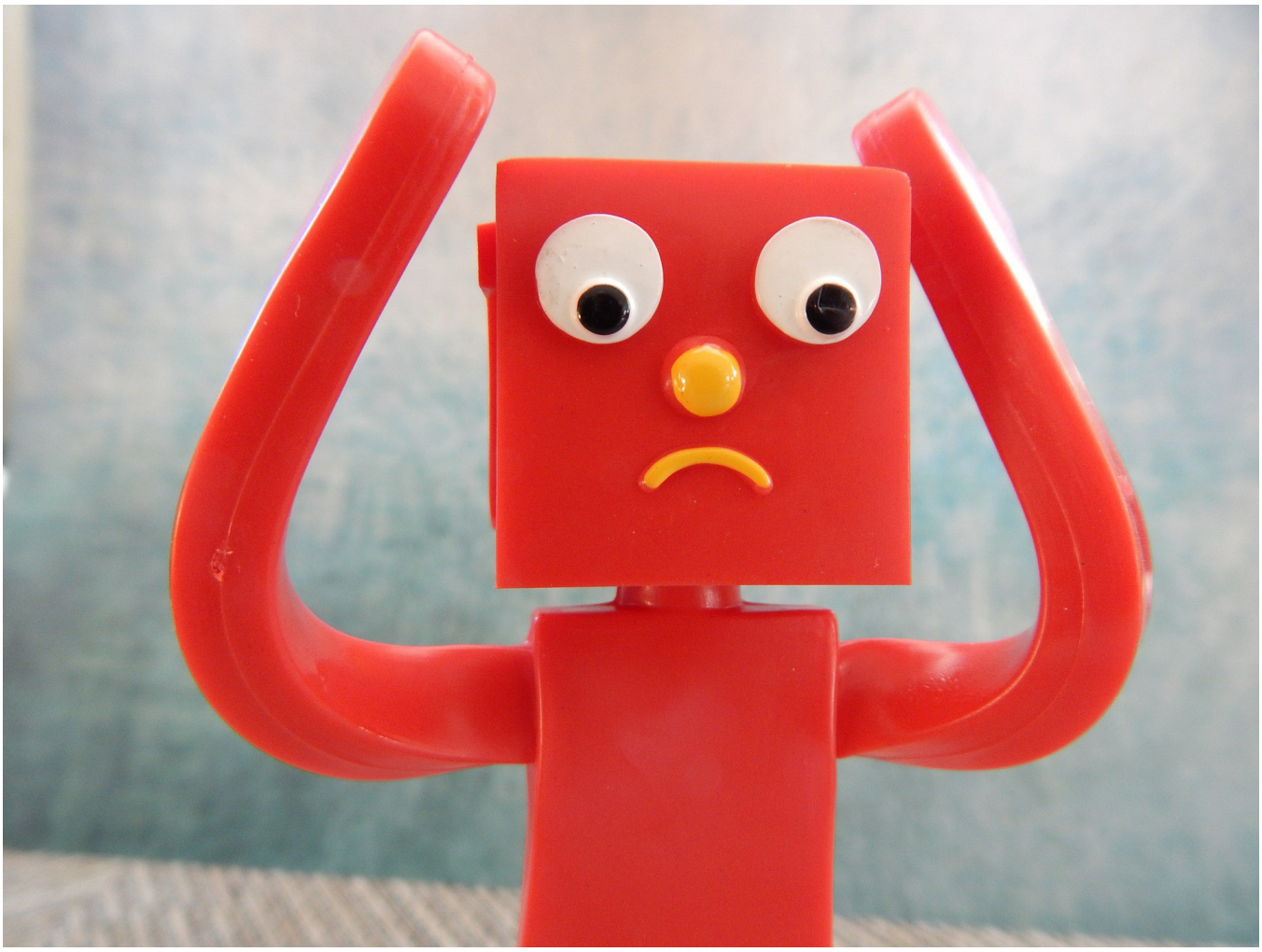
```
>>> print("Hello World!")
Hello World!
```

I was astounded by the accuracy of my simple classifier. If I recall correctly, on the MNIST handwritten digits dataset, it was north of 98% on the test set. (As a side note, this shows how far we’ve come along when a highly accurate image classifier can be built within hours. The ML community — and yes, that includes you— is awesome because of such liberal sharing of knowledge and tools)

Despite the high accuracy of the classifier **one issue kept nagging me:**

> The neural network would spit out a category even if I gave it images completely unrelated to what it has been trained on.

You know the drill. Train a cat vs dog classifier, throw an image of a person and the network would either classify it as a cat or as a dog. (Perhaps — if the network has some sense of humor — happy people as dogs and unhappy ones as cats).



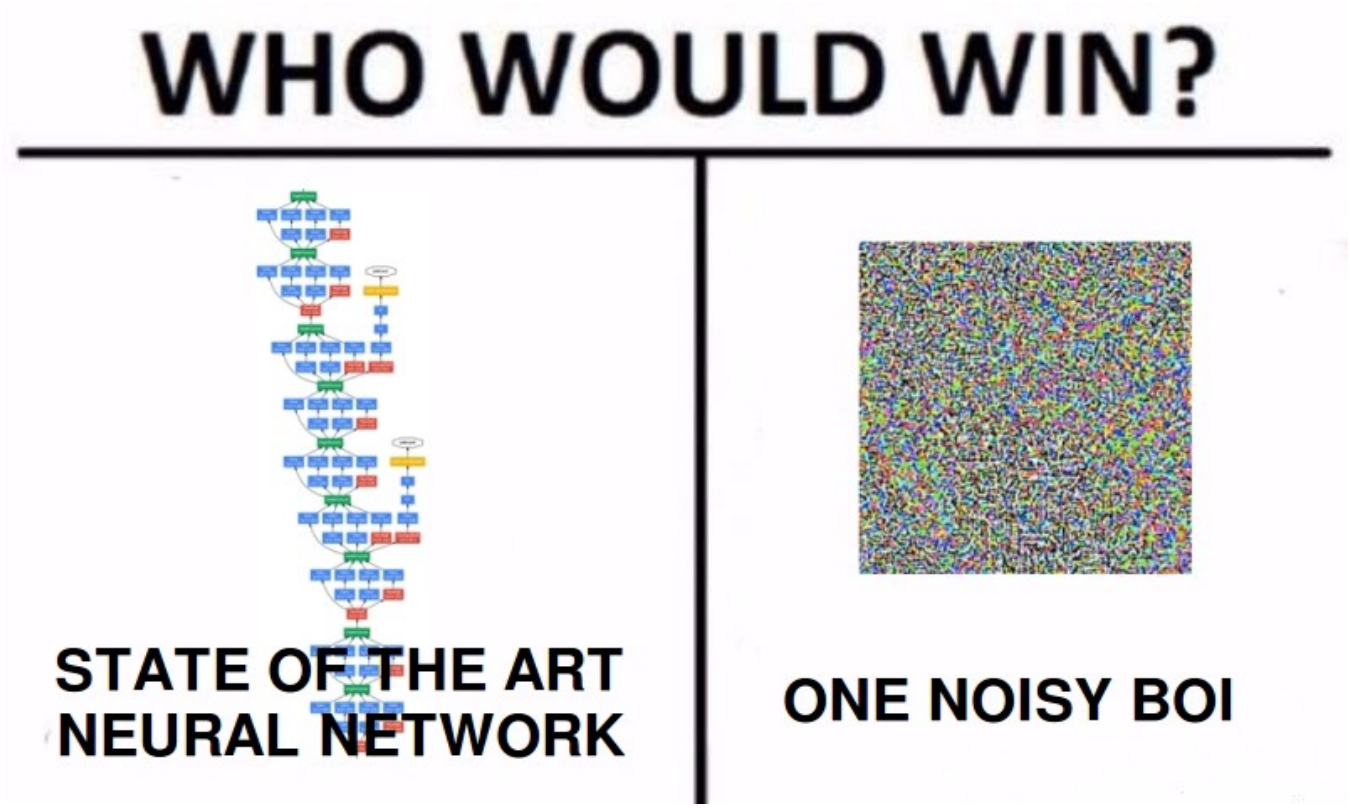
Build your model to throw up its (metaphorical) hands when it’s not sure (Photo via Pixabay)

I knew that my expectations from the classifier were unrealistic. It behaved exactly how it was programmed. If I interpreted the final layer (softmax) output as probabilities, there will always be a category with the maximum value for any image that’s given as an input. The network simply didn’t know the concept of throwing its hands and saying: “this looks like something I’m not trained for.”

But that’s exactly what I wanted my neural network to do.

. . .

In almost all real world problems, **what you want is not just a result but you also need knowledge of confidence / certainty in that result**. If you’re making self-driving car, you want to not just detect pedestrians but also express how confident you are that the object is a pedestrian and not a traffic cone. Similarly, if you are writing a bot that trades on the stock market, you want it to recognize when situation goes out of its comfort zone, so it can stop acting and not go bankrupt. A big part of intelligence is not acting when one is uncertain. So it’s surprising that for many ML projects, expressing uncertainity isn’t what’s aimed for.



Probably one noisy boi (via Tricking Neural Networks: Create your own Adversarial Examples)

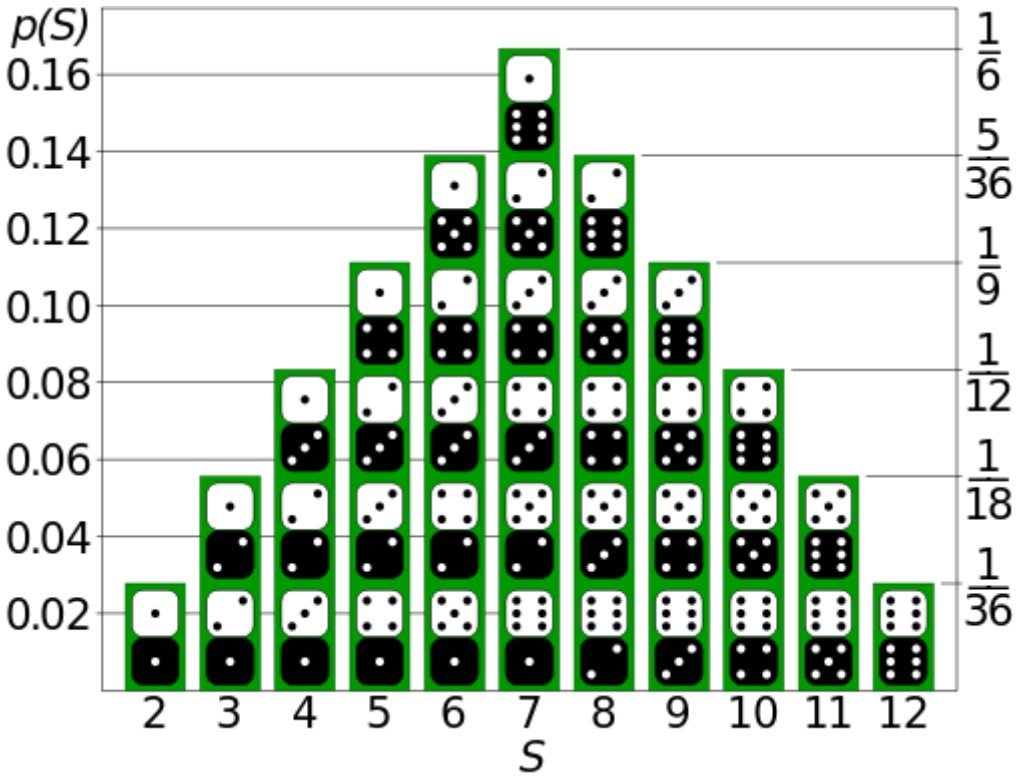
I wanted to explore this direction by building an MNIST classifier which can express (un)certainty of the input image being a particular digit. Such a classifier will have a high accuracy when you show it digits but refuse to classify when you throw unrelated images at it. **My final classifier had accuracy of ~97% on MNIST and it refused to classify white noise and the majority of unrelated (non-MNIST) images**. You can access the code here and may want to follow the Jupyter notebook contained in the repo along with this tutorial.

How bayesian neural networks work

I will not introduce the full extent of Bayesian analysis here, but I'll provide enough context for you to understand and then tinker with the code.

The key idea is pretty simple: in the Bayesian worldview, **everything has a probability distribution attached to it**, including model parameters (weights and biases in NNs). In programming languages, we have variables that can take a specific value and every-time you access the variable, you get the same value. In contrast to that, in the bayesian world, we have similar entities that are called **random variables** that give a different value every time you access it. So if X is a random variable representing the normal distribution, every time you access X, it'll have a different value.

This process of getting a new value from a random variable is called **sampling**. What value comes out depends on the random variable's associated probability distribution. The wider the probability distribution associated with a random variable, the more uncertainty there is regarding its value because it could then take any value as per the (wide) probability distribution.



If your random variable is the sum of digits of two dice throws, at each throw you'll get a value whose probability depends on the distribution above. This means the most likely sum that you can get is 7, and least likely is 2 and 12. (From Wikipedia)

In a traditional neural networks you have fixed weights and biases that determine how an input is transformed into an output. In a bayesian neural network, all weights and biases have a probability distribution attached to them. **To classify an image, you do**

multiple runs (forward passes) of the network, each time with a new set of sampled weights and biases. Instead of a single set of output values what you get is multiple sets, one for each of the multiple runs. The set of output values represent a probability distribution on output values and hence you can find out confidence and uncertainty in *each* of the outputs. As you will see, if the input image is something the network has never seen, for all output classes, the uncertainty will be high which you should interpret the network saying: "I really don't know what this image is about".

Writing your first Bayesian Neural Network in Pyro and PyTorch

The code assumes familiarity with basic ideas of probabilistic programming and PyTorch. In case you're new to either of these, I recommend following resources:

- Bayesian Methods for Hackers to learn the basics of Bayesian modeling and probabilistic programming
- Deep Learning with PyTorch: A 60 minute Blitz. Specifically, the tutorial on training a classifier.

PyTorch has a companion library called Pyro that gives the functionality to do probabilistic programming on neural networks written in PyTorch. This "automatic" conversion of NNs into bayesian counterparts has two steps:

- First, it helps in assigning probability distributions to all weights and biases in the network, hence converting them into random variables
- Second, it helps in using the training data to **infer** those probability distributions so that you can use it to classify images

Inference is the most difficult step of the entire process. It's based on the famous Bayes theorem that you may have seen before.

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

The deceptively simple equation that rules the world

Going into nitty-gritties of this equation is out of the scope of this tutorial but I'll try giving you intuition of what's happening. Assume *A* is the initial probability distributions of weights and biases (known as **priors,** usually some standard distribution like normal or uniform random) and *B* is the training data (input/output pairs of images/labels).

The key idea of the Bayes theorem which you should remember is that we want to use data to find out the updated distributions of weights and biases *P(A | B)* (**posterior**). Just like using initially randomly assigned weights and biases of a network, the initial distributions of parameters (priors) will give us wrong results. Only after using data to get updated distributions of parameters can we use the network to classify images.

The probability distributions of weights and biases are updated via the Bayes theorem taking into account their initial values *P(A)* and **likelihood** of those initial distributions to describe the input data *P (B|A)* (it's read as probability of B given A). The updated distributions of weights *P(A | B)* (**posterior**) depends on which one has a stronger pull — the prior or the likelihood. (If you're curious about the P(B) term, it'll get clear later in this tutorial).

I know that the paragraph above may make strict Bayesians cry in horror. I know the definitions are imprecise. But this tutorial isn't to introduce the full glory of bayesian ways of looking at the data. There are entire books and courses on it and I can't do justice to it in one tutorial. This tutorial is about practical implementation of a Bayesian neural network. I scratched my head for days diving into Pyro tutorials and trying to convert one of their examples into a classifier. I finally found a brief tutorial on IBM Watson's website on using Pyro on MNIST. My code is based on that tutorial but I extend it to on non-MNIST and white-noise data to see if bayesian neural networks can really say "I don't know" when presented with an input they have not seen before.

Even though I'll try explaining the basics of Pyro, you will get a lot of value from this tutorial if you go through their first three tutorials — part I, part II and part III.

Ready? Let's get straight to the code

```
class NN(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(NN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
```

```
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        output = self.fc1(x)
        output = F.relu(output)
        output = self.out(output)
        return output

train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('mnist-data/', train=True, download=True,
                       transform=transforms.Compose([transforms.ToTensor(),])),
        batch_size=128, shuffle=True)

test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('mnist-data/', train=False, transform=transforms.Compose([transf
                       ),
        batch_size=128, shuffle=True)

net = NN(28*28, 1024, 10)
```

net.py hosted with ❤ by GitHub view raw

After importing PyTorch, Pyro and other standard libraries (like matplotlib and numpy), we define a standard feedforward neural network of one hidden layer of 1024 units. We also load MNIST data.

```
def model(x_data, y_data):

    fc1w_prior = Normal(loc=torch.zeros_like(net.fc1.weight), scale=torch.ones_like(net
    fc1b_prior = Normal(loc=torch.zeros_like(net.fc1.bias), scale=torch.ones_like(net.f

    outw_prior = Normal(loc=torch.zeros_like(net.out.weight), scale=torch.ones_like(net
    outb_prior = Normal(loc=torch.zeros_like(net.out.bias), scale=torch.ones_like(net.o

    priors = {'fc1.weight': fc1w_prior, 'fc1.bias': fc1b_prior,  'out.weight': outw_pri

    # lift module parameters to random variables sampled from the priors
    lifted_module = pyro.random_module("module", net, priors)
    # sample a regressor (which also samples w and b)
    lifted_reg_model = lifted_module()

    lhat = log_softmax(lifted_reg_model(x_data))

    pyro.sample("obs", Categorical(logits=lhat), obs=y_data)
```

bnn-model.py hosted with ❤ by GitHub view raw

In Pyro, the *model()* function defines how the output data is generated. In our classifier, the 10 output values corresponding to each digit are generated when we run the neural network (initialised in the *net* variable above) with a flattened 28*28 pixel image. Within *model()*, the function *pyro.random_module()* converts parameters of our neural network (weights and biases) into random variables that have the initial (prior) probability distribution given by *fc1w_prior*, *fc1b_prior*, *outw_prior* and *outb_prior* (in our case, as you can see, we’re initialising these with a normal distribution). Finally, through pyro.sample(), we tell Pyro that the output of this network is categorical in nature (i.e. it can either be 0, 1, 2, and so on.)

```
def guide(x_data, y_data):

    # First layer weight distribution priors
    fc1w_mu = torch.randn_like(net.fc1.weight)
    fc1w_sigma = torch.randn_like(net.fc1.weight)
    fc1w_mu_param = pyro.param("fc1w_mu", fc1w_mu)
    fc1w_sigma_param = softplus(pyro.param("fc1w_sigma", fc1w_sigma))
    fc1w_prior = Normal(loc=fc1w_mu_param, scale=fc1w_sigma_param)
    # First layer bias distribution priors
    fc1b_mu = torch.randn_like(net.fc1.bias)
    fc1b_sigma = torch.randn_like(net.fc1.bias)
    fc1b_mu_param = pyro.param("fc1b_mu", fc1b_mu)
    fc1b_sigma_param = softplus(pyro.param("fc1b_sigma", fc1b_sigma))
    fc1b_prior = Normal(loc=fc1b_mu_param, scale=fc1b_sigma_param)
    # Output layer weight distribution priors
    outw_mu = torch.randn_like(net.out.weight)
    outw_sigma = torch.randn_like(net.out.weight)
    outw_mu_param = pyro.param("outw_mu", outw_mu)
    outw_sigma_param = softplus(pyro.param("outw_sigma", outw_sigma))
    outw_prior = Normal(loc=outw_mu_param, scale=outw_sigma_param).independent(1)
    # Output layer bias distribution priors
    outb_mu = torch.randn_like(net.out.bias)
    outb_sigma = torch.randn_like(net.out.bias)
    outb_mu_param = pyro.param("outb_mu", outb_mu)
    outb_sigma_param = softplus(pyro.param("outb_sigma", outb_sigma))
    outb_prior = Normal(loc=outb_mu_param, scale=outb_sigma_param)
    priors = {'fc1.weight': fc1w_prior, 'fc1.bias': fc1b_prior, 'out.weight': outw_pric

    lifted_module = pyro.random_module("module", net, priors)

    return lifted_module()
```

bnn-guide.py hosted with ❤ by GitHub view raw

Understanding this part — represented by the *guide()* function — was the trickiest thing for me. For quite some time, I didn't understand why was it needed, especially because it looked very much like the *model()* function. Explaining it will be hard, but I'll try. (If you aren't able to understand, my explanation I recommend Pyro tutorials or links below that I've provided on the topic).

Take a look at the Bayes equation again:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

In the *model()* function, we have defined *P(A)* — the priors on weights and biases. The *P(B|A)* part of the equation is represented by the neural network because given the parameters (weights and biases), we can do multiple runs on image, label pairs and find out the corresponding probability distribution of training data. Before training, initially since priors on weights and priors are all the same (all are normal distribution), the likelihood of getting a high probability for the correct label for a given image will be low.

In fact, **inference** is the process of learning probability distributions for weights and biases that maximize the **likelihood** of getting a high probability for the correct image, label pairs.

This process of inference is represented by *P(A |B)* which is the **posterior** probability of parameters *A* given the input/output pairs (*B*). I wrote earlier that inference is difficult. That's because of the term you see in the denominator *P(B)*. This term is called **evidence** and it is simply THE probability of observing the data (input/output pairs) under all possible parameter values, weighted by their respective probabilities.

$$P(B) = \sum_j P(B \mid A_j) P(A_j),$$

Calculating this sum is hard because of three reasons:

- Hypothetically, values of parameters *Aj* could range from -infinity to +infinity
- For *each* value of *Aj* in that range, you have to run the model to find the likelihood of generating the input, output pairs you observe (the total dataset could be in millions of pairs)
- There could be not one but many such parameters (j >> 1). In fact, for a neural network of our size, we have ~8million parameters (number of weights = 1024*28*28*10).

The type of enumeration approach for posterior that I describe above is not practical for anything but very trivial models. Instead of this grid-like enumeration, what if we could do random sampling? In fact, sampling based approaches are widely used and they go by the name Monte-Carlo methods. Particularly, Metropolis-Hastings is a popular algorithm for Monte-Carlo sampling. (It's included in Pyro and most of the other probabilistic programming languages).

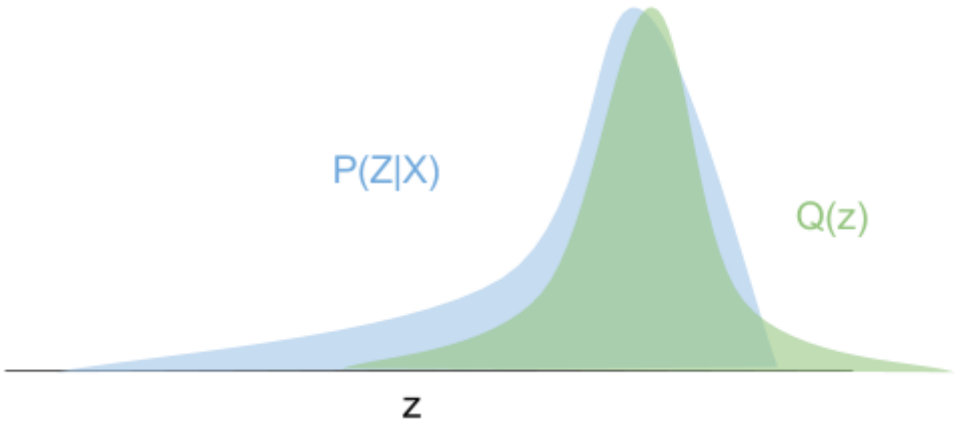
Unfortunately, for complex bayesian models such as a neural network with 8 million parameters, Monte-Carlo methods are still slow to converge and may take weeks to discover the full posterior.

Thankfully, there's an increasingly popular method called Variational Bayes that seems perfect for finding posteriors for neural network parameters, even for large datasets. To understand the intuition behind this technique, I highly recommend watching the following video (up to the first 40 minutes).

Tutorial Session: Variational Bayes and Beyond: Bayesian Inference for Bi...

The gist of Variational Bayes methods is that since we can't exactly compute the posterior, we can find the closest probability distribution to it that is "well-behaved". By "well-behaved", I mean a distribution (like a normal or an exponential distribution) that can be represented by a small set of parameters like mean or variance. So, after random initialization of those parameters in a "well-behaved" distribution, you can do gradient descent and modify parameters of the distribution (like mean or variance) a little bit each time to see if the resulting distribution is closer to the posterior that you want to calculate. (If you're thinking how do we know if resulting distribution is closer to the posterior if posterior is exactly what we want to calculate, you've understood the idea. The answer is that, surprisingly, we don't need the exact posterior to find closeness between it and the other "well-behaved" distribution. Watch the video above to understand the measure of closeness that we actually optimize for: Evidence Lower Bound or ELBO. I also found this series of posts useful on the topic).

To understand Variational Bayes intuitively, see the diagram below:



Via A Beginner's Guide to Variational Methods: Mean-Field Approximation

The blue curve is the true posterior that you'll get if you do that long (enumerative) calculation we talked about before. This curve can take any arbitrary shape because it's a result of enumerative calculation. In contrast to that, because it's a well behaved distribution like a normal distribution, the green curve's entire shape can be described one parameter Z. What Variational Bayes methods do is to then use gradient descent methods to change the value of Z parameter from initial randomly initialised value to the value whose resultant distribution best approximates the true posterior. At the end of optimization, the green curve isn't exactly like the blue curve but it's pretty similar. And we can safely use the approximating green curve instead of unknown true blue

curve for making predictions. (If all this is hard to understand, I recommend watching the video above.)

Now this is where the guide function comes in. It helps us initialize a well-behaved distribution that later we can optimize to approximate the true posterior. Take a look at it again:

```
def guide(x_data, y_data):

    # First layer weight distribution priors
    fc1w_mu = torch.randn_like(net.fc1.weight)
    fc1w_sigma = torch.randn_like(net.fc1.weight)
    fc1w_mu_param = pyro.param("fc1w_mu", fc1w_mu)
    fc1w_sigma_param = softplus(pyro.param("fc1w_sigma", fc1w_sigma))
    fc1w_prior = Normal(loc=fc1w_mu_param, scale=fc1w_sigma_param)
    # First layer bias distribution priors
    fc1b_mu = torch.randn_like(net.fc1.bias)
    fc1b_sigma = torch.randn_like(net.fc1.bias)
    fc1b_mu_param = pyro.param("fc1b_mu", fc1b_mu)
    fc1b_sigma_param = softplus(pyro.param("fc1b_sigma", fc1b_sigma))
    fc1b_prior = Normal(loc=fc1b_mu_param, scale=fc1b_sigma_param)
    # Output layer weight distribution priors
    outw_mu = torch.randn_like(net.out.weight)
    outw_sigma = torch.randn_like(net.out.weight)
    outw_mu_param = pyro.param("outw_mu", outw_mu)
    outw_sigma_param = softplus(pyro.param("outw_sigma", outw_sigma))
    outw_prior = Normal(loc=outw_mu_param, scale=outw_sigma_param).independent(1)
    # Output layer bias distribution priors
    outb_mu = torch.randn_like(net.out.bias)
    outb_sigma = torch.randn_like(net.out.bias)
    outb_mu_param = pyro.param("outb_mu", outb_mu)
    outb_sigma_param = softplus(pyro.param("outb_sigma", outb_sigma))
    outb_prior = Normal(loc=outb_mu_param, scale=outb_sigma_param)
    priors = {'fc1.weight': fc1w_prior, 'fc1.bias': fc1b_prior, 'out.weight': outw_pric

    lifted_module = pyro.random_module("module", net, priors)

    return lifted_module()
```

bnn-guide.py hosted with ❤️ by GitHub view raw

This *guide()* function describes the Z parameters (like mean and variance of weights and biases) that can be changed to see if resultant distribution closely approximates the posterior that comes out of *model()*. Now, in our case the *model()* looks very similar to

guide() but that need not always be the case. In theory, the *model()* function could be much more complicated than the *guide()* function.

With *model()* and *guide()* functions figured out, we're ready to do inference. First, let's tell Pyro which optimizer to use for doing variational inference.

You'll notice that we're using the Adam optimizer from PyTorch (to know more about it and other optimization algorithms, here's a fantastic series). The loss function that we're using for optimization is ELBO (this is like using Mean Squared Error or Cross Entropy loss when training a non-bayesian neural network via backpropagation).

Let's write the optimization loop.

```
num_iterations = 5
loss = 0

for j in range(num_iterations):
    loss = 0
    for batch_id, data in enumerate(train_loader):
        # calculate the loss and take a gradient step
        loss += svi.step(data[0].view(-1,28*28), data[1])
    normalizer_train = len(train_loader.dataset)
    total_epoch_loss_train = loss / normalizer_train

    print("Epoch ", j, " Loss ", total_epoch_loss_train)
```

optim-loop.py hosted with ❤ by GitHub view raw

You'd notice that this loop is pretty much how we train a standard neural network. There are multiple epochs / iterations (in this case it's 5). And in each iteration, we go through a mini-batch of data (input/output pairs of images, labels). One more benefit of variational inference is that we do not have to feed in the entire dataset in one go (which could be in millions). Since an optimizer takes many thousands of steps to find the best value of parameters of guide function, at each step we can feed it the a separate mini-batch of data. This speeds up inference tremendously.

Once the loss seems to be stabilizing / converging to a value, we can stop the optimization and see how accurate our bayesian neural network is. Here's the code for doing that.

```
num_samples = 10
```

```
def predict(x):
    sampled_models = [guide(None, None) for _ in range(num_samples)]
    yhats = [model(x).data for model in sampled_models]
    mean = torch.mean(torch.stack(yhats), 0)
    return np.argmax(mean.numpy(), axis=1)

print('Prediction when network is forced to predict')
correct = 0
total = 0
for j, data in enumerate(test_loader):
    images, labels = data
    predicted = predict(images.view(-1,28*28))
    total += labels.size(0)
    correct += (predicted == labels).sum().item()
print("accuracy: %d %%" % (100 * correct / total))
```

bnn-predict.py hosted with ❤ by GitHub view raw

First thing to notice in the *predict()* function is that we're using the learned *guide()* function (and not the *model()* function) to do predictions. This is because for *model()*, all we know is priors for weights and not the posterior. But for *guide()* after optimization iterations, the distribution given by the parameter values approximate the true posterior and so we can use it for predictions.

Second thing to notice is that for each prediction, we're sampling a new set of weights and parameters 10 times (given by *num_samples*). This effectively means that we're sampling a new neural network 10 times for making one prediction. As you will see later, this is what enables us to give uncertainities on outputs. In the case above, to make a prediction, we're averaging final layer output values of the 10 sampled nets for the given input and taking the max activation value as the predicted digit. Doing that, we see that **our net is accurate 89% of times on the test set**. But note that in this case, we're forcing our net to make a prediction in each case. We haven't used the magic of Bayes theorem to enable our net to say: "I refuse to make a prediction here".

That is exactly we will do next using the code below.

```
prob = np.percentile(histo_exp, 50) #sampling median probability

if(prob>0.2): #select if network thinks this sample is 20% chance of this being a label
  highlight = True #possibly an answer
```

bnn-prob.py hosted with ❤ by GitHub view raw

I won't go into the full code of estimating uncertainty (which you can see in the notebook). Essentially, what we're doing is this:

- For an input image, take 100 samples of neural networks to get 100 different output values from the last layer
- Convert those outputs (which are logsoftmaxed) into probabilities by exponentiating them
- Now, given the input image, for *each* digit we have 100 probability values
- We take median (50th percentile) of these 100 probability values as the threshold probability for each digit
- If the threshold probability is greater than 0.2, we select the digit as a classification output from the network

In other words, we want the neural network to output a digit as a recommendation if out of multiple samples of probability, the median probability for that digit is at least 0.2. This means that for some inputs, the network can output two digits as classification output while for others it can output no digits (which is exactly what we want if we give it non-digit images).

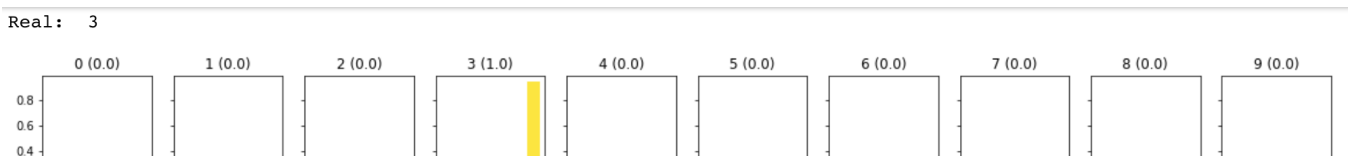
Results on the MNIST dataset

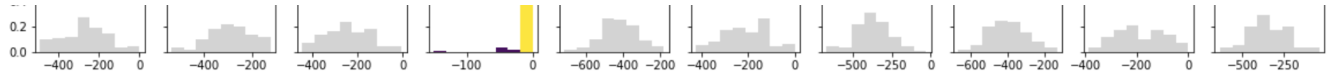
When I ran the network on the entire MNIST test set of 10,000 images, I got these results:

- **Percentage of images which the network refused to classify: 12.5%** (1250 out of 10,000)
- **Accuracy on the remaining 8750 "accepted" images: 96%**

Note that this 96% accuracy when we gave the network a chance to refuse classification is much higher than the 88% accuracy when it was forced to classify.

To visualize what's happening under the hood. I plotted 100 random images from the MNIST test batch. For most of the 100 images, the network classified accurately.



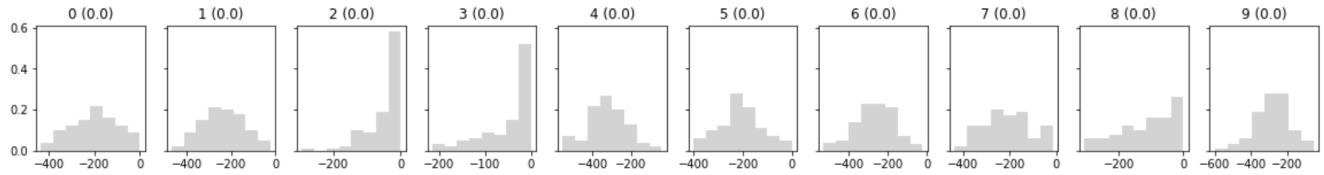


```
Correct
```

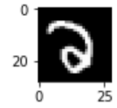
What the plot above shows is that the real label for input image was 3, and for each of the 10 digits, a histogram of log-probabilities is shown. For the label 3, the median log-probability was actually close to 0 which means the probability of this image being 3 is close to 1 ($\exp(0) = 1$). That is why it's highlighted in yellow. Since the label that network selected is same as the real label, it shows "Correct". You can also see what the input image actually looked like.

In my multiple runs with 100 images, the accuracy of network when it made predictions was 94–96%. **The network regularly chose to not make predictions on 10–15% images** and it was fun to look at some of the images where network said: "I'm not really sure".

```
Real:  2
```



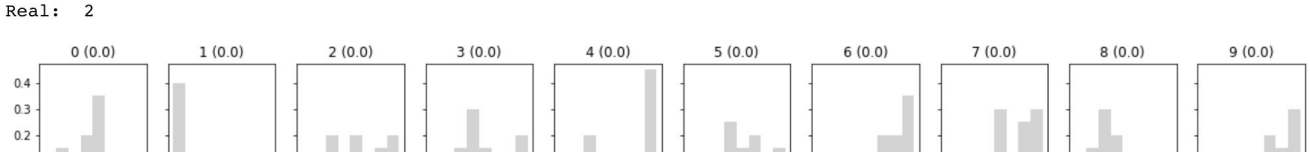
```
Undecided.
```

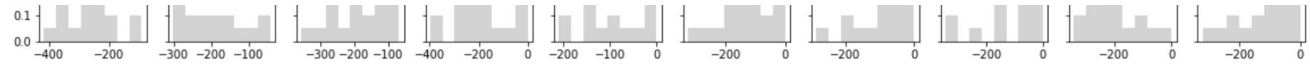


It's hard for even me to tell that the digit is a "2". You can see from histograms that the network had a high uncertainty both for 2 and 3 labels . For such cases where network is undecided, the distribution of log-probabilities is wide for *all* labels while in the case of the accurate classification of "3" in the plot above, you'd notice that the distribution for the digit 3 was narrow while for all other digits it was wide (which meant the network was pretty sure it was 3).

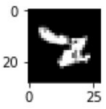
Another case where the network was undecided.

```
Real:  2
```





```
Undecided. Plotting image.
```



You see the image is all messed up. A traditional neural network might have spitted out something but our bayesian network refuses to say anything.

Results on randomly generated images

To see how the network does when it is fed pure white noise, I generated 100 random images.

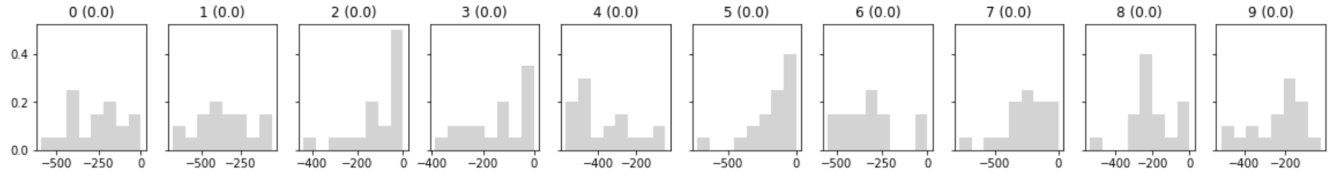
```
# generate random data

images_random = torch.rand(100,28,28)
labels_random = torch.randint(0,10, (100,))
```

random-images.py hosted with ❤️ by GitHub view raw

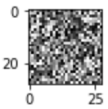
When these images were given as an input, **the network refused to make predictions on 95% of them**.

This is how a typical randomly generated image looked like:

```
Real:  5.0
```



```
Undecided. Plotting image.
```

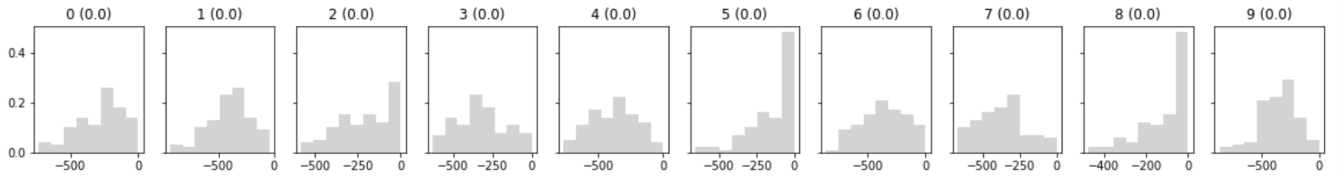


Results on the not-MNIST dataset

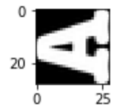
I went one step further and downloaded the not-MNIST dataset which is a dataset of alphabets rather than digits. It looks like this:

For the not-MNIST test set, **the network refused to classify ~80% of images** (363 out of a total of 459 in the test set).

An example of not-MNIST image is shown below.




```
Undecided.
```



It’s great to see our network give good accuracy on what it was trained on (MNSIT) while not getting fooled by a dataset that was custom designed to fool it (not-MNIST).

Conclusion and how to make our Bayesian network even better

The state of the art results on MNIST dataset have 99.8% accuracy. So our ~96% accuracy (when we want to make a prediction) is a far cry from that.

There are four ways to get better accuracy:

- We used a very simple model: single layer neural network with 1024 neurons. If we use a more advanced convolutional network, I’m sure we can improve our accuracy.
- If we keep running our optimization for much longer, we can improve our accuracy

- If we sample more data points (rather than 100) per image, results could improve
- If we make our acceptance criteria from median probability to be minimum 0.2 to perhaps 10th percentile probability to be minimum 0.5, our network will reject a lot more images but on accepted ones, it may have a higher accuracy

Overall, I'm very happy with the results. I hope you have fun playing with the code :)

Feel free to comment on this post with your questions and I'll try my best to answer them. If you are able to improve the code, send a pull request to me on github. And in case, you use the basic code on a new data set or problem, please email me at paras1987 <at> gmail <dot> com and I'd love to hear from you.

Thanks Nirant Kasliwal, Divyanshu Kalra and S. Adithya for reviewing the draft and giving helpful suggestions.

PS: I've recently made a 20 minute video on what makes deep learning so effective. Go watch it now!

Liked this tutorial? Check out my other tutorials too:

- One neural network, many uses. Build image search, image captioning, similar words and similar images using a single model
- Making deep neural networks paint to understand how they work. Generate abstract art in 100 lines of PyTorch code and explore how neural networks work
- Generating New Ideas for Machine Learning Projects Through Machine Learning. Generating style-specific text from a small corpus of 2.5k sentences using a pre-trained language model. Code in PyTorch
- Reinforcement learning without gradients: evolving agents using Genetic Algorithms. Implementing Deep Neuroevolution in PyTorch to evolve an agent for CartPole [code + tutorial]

I tweet about deep learning and AI. Follow me at https://twitter.com/paraschopra