Neural Networks Project

Image classifier

Realized by: OUHMAID khalid

Supervised by: **Hamd AIT ABDELALI**

Master SDBD: 2019-2020

Table of Contents

Neural Network Definition

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated.

Neural networks help us cluster and classify. You can think of them as a clustering and classification layer on top of the data you store and manage. They help to group unlabeled data according to similarities among the example inputs, and they classify data when they have a labeled dataset to train on. (Neural networks can also extract features that are fed to other algorithms for clustering and classification; so you can think of deep neural networks as components of larger machine-learning applications involving algorithms for reinforcement learning, classification and regression.)

Neural Network Project

I have built a neural network project which is an **image classifier** that works pretty well using **pytotch** and the dataset : **MNIST handwritten digits dataset.**

Understanding the code of the image classifier

```
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([transforms
                       ),
        batch_size=128, shuffle=True)

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

After importing PyTorch, Pyro and other standard libraries (like matplotlib and numpy), I defined 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.fc1.weight))
    fc1b_prior = Normal(loc=torch.zeros_like(net.fc1.bias), scale=torch.ones_like(net.fc1.bias))

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

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

    # 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)
```

in Pyro, the *model()* function defines how the output data is generated. In my 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 my case, we're initialising these with a normal distribution). Finally, through pyro.sample(), i tell Pyro that the output of this network is categorical in nature

```python
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_prior, 'out.b

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

    return lifted_module()
```

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

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), i 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*).

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

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).

```
optim = Adam({"lr": 0.01})
svi = SVI(model, guide, optim, loss=Trace_ELBO())
```

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 my 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.

```
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)
```

We notice that this loop is pretty much how we train a standard neural network. There are multiple epochs (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))
```

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
```

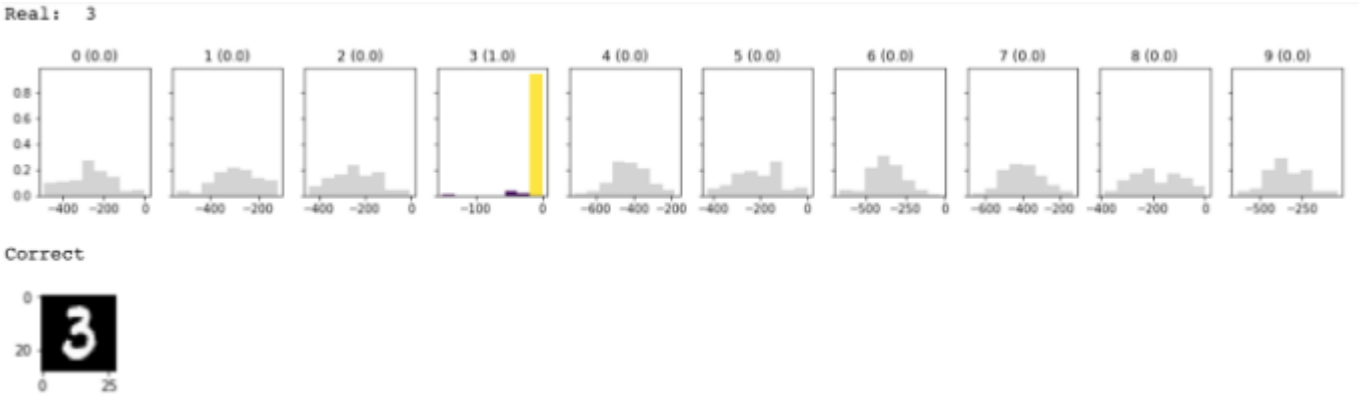
Results on the MNIST dataset

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%**

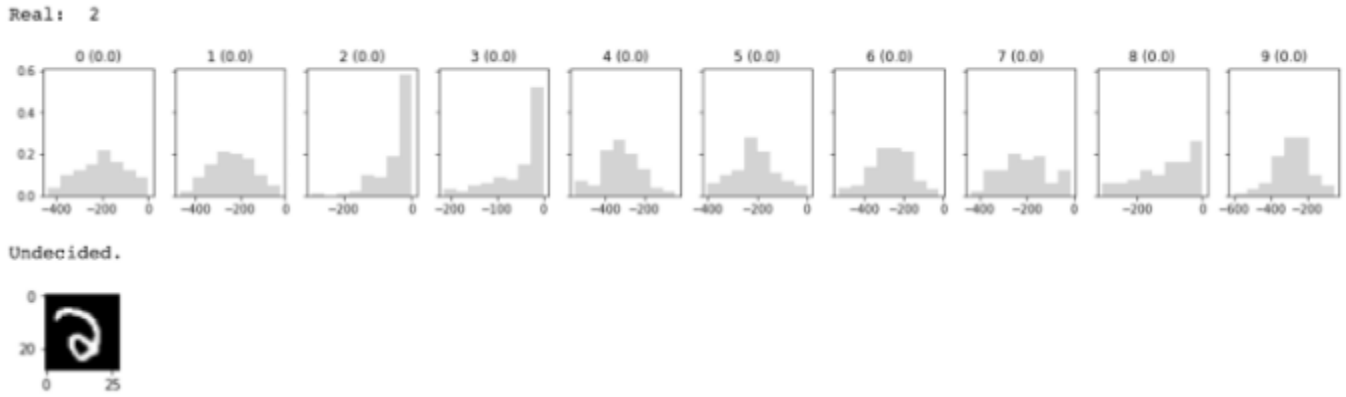
Note that this 96% accuracy when we gave the network a chance to refuse classification is much higher than the 88% accuracy when it was forced to classify.

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.



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.

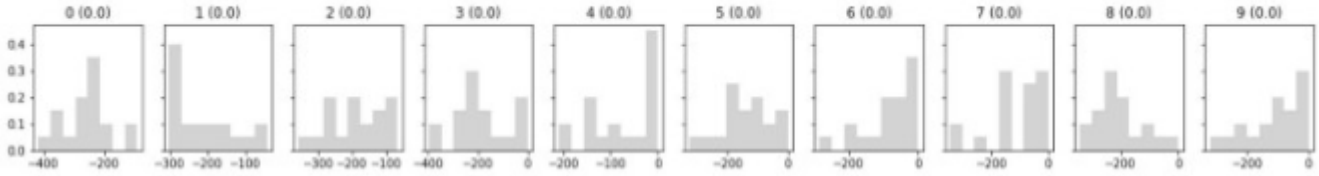
In my multiple runs with 100 images, the accuracy of network when it made predictions was 94–96%. **The network regularly chose to not make predictions on 10–15% images** and it was fun to look at some of the images where network said: "I'm not really sure".



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.
```