homework 6 solutions

Exercise 1

Solution 1

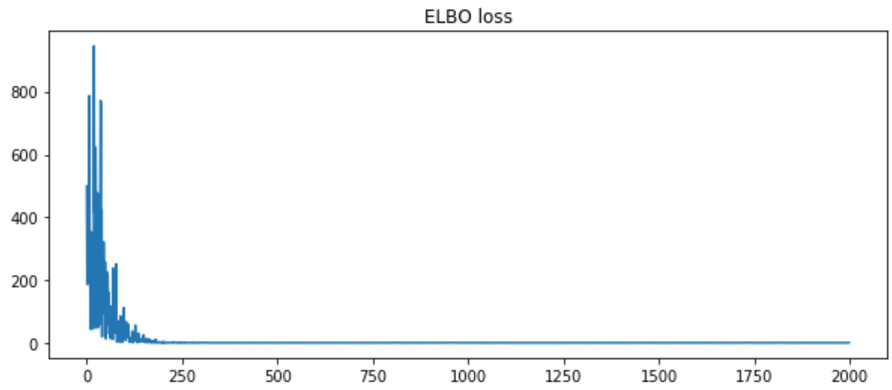
```
import pyro
import torch
import pyro.distributions as dist
import pyro.optim as optim
from pyro.infer import SVI, Trace_ELBO
import pandas as pd
%pylab inline
from pyro.infer import Predictive
import torch.distributions.constraints as constraints
figsize=(10,4)
pyro.set_rng_seed(0)
```

```
normalize = True

smoke = pd.read_csv('~/Desktop/smoke.csv', header=None, names=['age_cat','smoke_cat','popul
ation','deaths'])
if normalize: smoke = (smoke-smoke.min())/(smoke.max()-smoke.min())

from random import sample
indexes = np.array([sample(range(0,9),2) for i in range(4)])
for i,j in enumerate(indexes):
    j+= 9 * i

train_data = smoke.drop(indexes.flatten(),axis=0)
test_data = smoke.iloc[indexes.flatten()]

y_train = torch.tensor(train_data["deaths"].values, dtype=torch.float)
x_train = torch.stack([torch.tensor(train_data[column].values, dtype=torch.float)
                        for column in ['age_cat','smoke_cat','population']], 1)

y_test = torch.tensor(test_data["deaths"].values, dtype=torch.float)
x_test = torch.stack([torch.tensor(test_data[column].values, dtype=torch.float)
                       for column in ['age_cat','smoke_cat','population']], 1)
```

```python
pyro.clear_param_store()

def death_model(x,y):
    n_observations, n_predictors = x.shape

    w = pyro.sample("w", dist.Normal(torch.zeros(n_predictors), 1e-2 * torch.ones(n_predict
ors)))
    b = pyro.sample("b", dist.Normal(0.,1e-2))

    #I suppose the text meant mu=E[y|x]= exp(W*x+b)
    mu = torch.exp((w*x).sum(dim=1) + b)

    with pyro.plate("target", n_observations):
        pyro.sample("y", dist.Poisson(mu), obs=y)

def death_guide(x,y=None):
    n_observations, n_predictors = x.shape

    w_loc = pyro.param("w_loc", torch.rand(n_predictors))
    w_scale = pyro.param("w_scale", torch.rand(n_predictors),
                         constraint=constraints.positive)

    w = pyro.sample("w", dist.Normal(w_loc, w_scale))

    b_loc = pyro.param("b_loc", torch.rand(1))
    b_scale = pyro.param("b_scale", torch.rand(1), constraint=constraints.positive)

    b = pyro.sample("b", dist.Normal(b_loc, b_scale))
```

```python
death_svi = SVI(model=death_model, guide=death_guide,
              optim=optim.ClippedAdam({'lr' : 2e-2}),
              loss=Trace_ELBO())
losses = []
for step in range(2000):
    loss = death_svi.step(x_train, y_train)/len(x_train)
    losses.append(loss)
    if step % 100 == 0:
        print(f"Step {step} : loss = {loss}")

fig, ax = plt.subplots(figsize=figsize)
ax.plot(losses)
ax.set_title("ELBO loss");
```

```
Step 0 : loss = 499.41790612254823
Step 100 : loss = 42.301537820271086
Step 200 : loss = 1.3327459607805525
Step 300 : loss = 1.0865782839911324
Step 400 : loss = 1.346647356237684
Step 500 : loss = 0.960088815007891
Step 600 : loss = 0.9350871358598981
Step 700 : loss = 1.0514477108206068
Step 800 : loss = 1.0344758033752441
Step 900 : loss = 0.960349610873631
Step 1000 : loss = 1.0363010466098785
Step 1100 : loss = 1.0519108389105116
Step 1200 : loss = 0.9867801155362811
Step 1300 : loss = 0.9484681401933942
Step 1400 : loss = 0.9679238285337176
Step 1500 : loss = 0.9663672276905605
Step 1600 : loss = 1.0166945457458496
Step 1700 : loss = 0.9419149841581073
Step 1800 : loss = 0.9255082777568272
Step 1900 : loss = 1.091580901827131
```




```
# w_i and b posterior mean
inferred_w = pyro.get_param_store()["w_loc"]
inferred_b = pyro.get_param_store()["b_loc"]

y_pred = torch.exp(inferred_w * x_test).sum(1) + inferred_b

print("MAE =", torch.nn.L1Loss()(y_test, y_pred).item())
print("MSE =", torch.nn.MSELoss()(y_test, y_pred).item())
```

```
MAE = 2.74535870552063
MSE = 7.591362476348877
```

Solution 2

```
data = pd.read_csv('ex6.csv')
data.head(5)
```

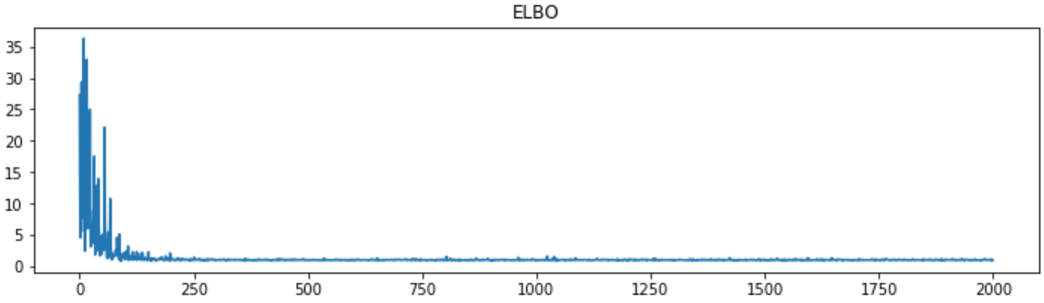
	age	smoke	pop	dead
0	1	1	656	18
1	2	1	359	22
2	3	1	249	19
3	4	1	632	55
4	5	1	1067	117

```
# prepare all data
# I drop the first columns to prevent multicollinearity and for the interpretability
pred=pd.get_dummies(data,columns=["age","smoke"],drop_first=True)
pred.columns=["pop","dead","45-49","50-54","55-59","60-64","65-69",
              "70-74","75-79","80+","cigarPipeOnly","cigarrettePlus","cigarretteOnly"]
pred=(pred-pred.min())/(pred.max()-pred.min()) # to standardize data
# data to torch tensor
dead = torch.tensor(pred["dead"].values, dtype=torch.float)
predictors = torch.stack([torch.tensor(pred[column].values, dtype=torch.float) for column i
n
                          ["45-49","50-54","55-59","60-64","65-69","70-74","75-79","80+",
                           "cigarPipeOnly","cigarrettePlus","cigarretteOnly","pop"]],1)
# 80% of data are use to train and 20% to test
X_train, X_test, dead_train, dead_test = train_test_split(predictors,dead,test_size=0.2,ran
dom_state=2)
pred.head(5)
```

	pop	dead	45-49	50-54	55-59	60-64	65-69	70-74	75-79	80+	cigarPipeOnly	cigarrettePlus	cigarretteOnly
0	0.093719	0.016016	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.043836	0.020020	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.025361	0.017017	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.089688	0.053053	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.162748	0.115115	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
pyro.clear_param_store()

def dead_model(predictors, dead):
    n_observations, n_predictors = predictors.shape
    # sample weights
    w = pyro.sample("w", dist.Normal(torch.zeros(n_predictors),torch.ones(n_predictors)))
    b = pyro.sample("b", dist.LogNormal(0, 1))
    yhat = torch.exp((w*predictors).sum(dim=1) + b)
    # condition on the observations
    with pyro.plate("dead", n_observations):
        pyro.sample("obs", dist.Poisson(yhat), obs=dead)

def dead_guide(predictors, dead=None):
    n_observations, n_predictors = predictors.shape
    w_loc = pyro.param("w_loc", torch.rand(n_predictors))
    w_scale = pyro.param("w_scale", torch.rand(n_predictors), constraint=constraints.positi
ve)
    w = pyro.sample("w", dist.Normal(w_loc, w_scale))
    b_loc = pyro.param("b_loc", torch.rand(1))
    b_scale = pyro.param("b_scale", torch.rand(1), constraint=constraints.positive)
    b = pyro.sample("b", dist.LogNormal(b_loc, b_scale))
```

```
dead_svi = SVI(model=dead_model, guide=dead_guide,optim=optim.ClippedAdam({'lr' : 0.01}),lo
ss=Trace_ELBO())

losses=[]
for step in range(2000):
    loss = dead_svi.step(X_train, dead_train)/len(X_train)
    losses.append(loss)
    if step % 100 == 0:
        print(f"Step {step} : loss = {loss}")

fig, ax=plt.subplots(figsize=(12,3))
ax.plot(losses)
ax.set_title("ELBO");
```




```python
print("Inferred params:", list(pyro.get_param_store().keys()), end="\n\n")

# w_i and b posterior mean
inferred_w = pyro.get_param_store()["w_loc"]
inferred_b = pyro.get_param_store()["b_loc"]

for i,w in enumerate(inferred_w):
    print(f"w_{i} = {w.item():.8f}")
print(f"b = {inferred_b.item():.8f}")
```

```
Inferred params: ['w_loc', 'w_scale', 'b_loc', 'b_scale']

w_0 = -1.06922281
w_1 = -1.09135151
w_2 = -0.68223840
w_3 = -0.34224269
w_4 = -0.47131369
w_5 = -0.66442955
w_6 = -0.85937321
w_7 = -0.94235039
w_8 = -0.91154218
w_9 = -0.50682598
w_10 = -0.47133189
w_11 = -0.42116481
b = -1.71771407
```

Since I dropped the first columns of the dummies (that is I consider them the base classes), their interpretation is included in the intercept b and the other classes are interpreted as an addition or a reduction from the base class in logarithmic case.

For example assuming that class 45-49 is equal to 1 and is a no smoke:
$ln(\mathbb{E}(\mu|x)) = w_0 * pop - 1.06718993 - 1.68892777$

```python
# print latent params quantile information
def summary(samples):
    stats = {}
    for par_name, values in samples.items():
        marginal = pd.DataFrame(values)
        percentiles=[.05, 0.5, 0.95]
        describe = marginal.describe(percentiles).transpose()
        stats[par_name] = describe[["mean", "std", "5%", "50%", "95%"]]
    return stats

# define the posterior predictive
predictive = Predictive(model=dead_model, guide=dead_guide, num_samples=100, return_sites=
("w","b","sigma"))

# get posterior samples on test data
svi_samples = {k: v.detach().numpy() for k, v in predictive(X_test, dead_test).items()}

# show summary statistics
for key, value in summary(svi_samples).items():
    print(f"Sampled parameter = {key}\n\n{value}\n")
```

```
Sampled parameter = w

        mean       std        5%        50%       95%
0   -1.061201  0.634297 -1.954491 -1.065024 -0.089888
1   -1.067627  0.738118 -2.181260 -0.957188  0.146299
2   -0.645354  0.539468 -1.563853 -0.648842  0.164813
3   -0.341803  0.522325 -1.262750 -0.304551  0.438478
4   -0.510206  0.476894 -1.259992 -0.474618  0.304026
5   -0.673575  0.658050 -1.872210 -0.612291  0.328389
6   -0.790928  0.532371 -1.697377 -0.672947 -0.063504
7   -0.865187  0.677436 -2.047732 -0.876014  0.323543
8   -0.996276  0.586803 -1.854990 -0.959474 -0.165680
9   -0.532058  0.449436 -1.244095 -0.481532  0.229961
10  -0.464310  0.591172 -1.457940 -0.415338  0.420104
11  -0.394778  0.577144 -1.484544 -0.345890  0.460817

Sampled parameter = b

       mean       std        5%        50%       95%
0  0.230301  0.135515  0.088434  0.197502  0.518164
```

```
# compute predictions using the inferred paramters
y_pred = (inferred_w * X_test).sum(1) + inferred_b

print("MAE =", torch.nn.L1Loss()(dead_test, y_pred).item())
print("MSE =", torch.nn.MSELoss()(dead_test, y_pred).item())
pd.DataFrame({'test': dead_test.tolist(), 'predict': y_pred.tolist()})
```

```
MAE = 3.030893087387085
MSE = 9.569859504699707
```

Exercise 2

Solution 1

First of all we import the dataset, normalize it and perform the train-test split.

```
# import data set
from sklearn import datasets

iris = datasets.load_iris()
```

```
# convert to torch.tensor
features = torch.stack([torch.tensor(iris.data[:,i]) for i in range(0,4)], dim=1)
labels = torch.tensor(iris.target)
```

```
# normalize data
features_norm = (features - torch.mean(features, dim=0))/ torch.std(features, dim=0)
```

```
# train-test split
x_train, x_test, y_train, y_test = train_test_split(features_norm, labels, random_state=0,
test_size=0.2)
```

Let's set the class "setosa", class 0, as the baseline class.

For the other two classes we assume:

$$w_k \sim \mathcal{N}(0, 1)$$
$$b_k \sim \mathcal{N}(0, 1)$$
$$y = X\, w_k + b_k$$

and we compute the probabilities $P(y|x, w_k)$ through the Softmax function.

```
pyro.clear_param_store()

def log_reg_model(x, y):
    n_observations, n_predictors = x.shape

    # sample weights
    w1 = pyro.sample("w1", dist.Normal(torch.zeros(n_predictors), torch.ones(n_predictor
s)))
    w2 = pyro.sample("w2", dist.Normal(torch.zeros(n_predictors), torch.ones(n_predictor
s)))

    # sample bias term
    b1 = pyro.sample("b1", dist.Normal(0.,1.))
    b2 = pyro.sample("b2", dist.Normal(0.,1.))

    # compute the y
    l0 = torch.zeros(n_observations, dtype= float)
    l1 = (w1*x).sum(dim=1) + b1
    l2 = (w2*x).sum(dim=1) + b2

    # compute the probabilities
    softmax = torch.nn.Softmax(dim=1)
    v = softmax(torch.t(torch.stack([l0,l1,l2])))

    # condition on the observations
    with pyro.plate("data", n_observations):
        y = pyro.sample("y", dist.Categorical(probs=v), obs=y)
```

We define the posterior distributions as

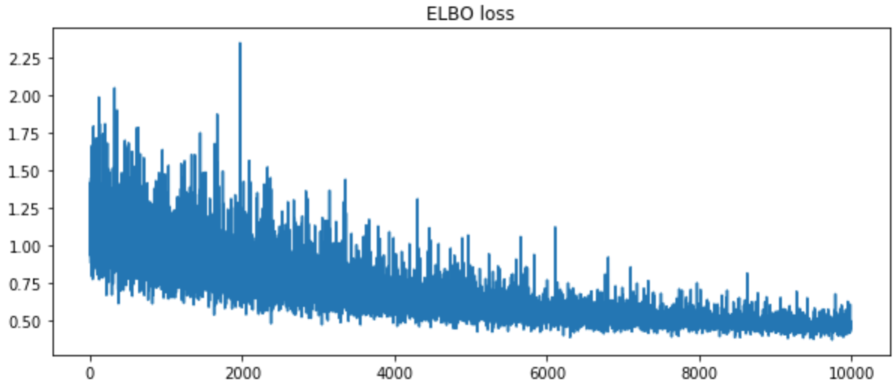
$$w_k \sim Normal$$
$$b_k \sim Normal$$

```
def log_reg_guide(x, y=None):

    n_observations, n_predictors = x.shape

    w1_loc = pyro.param("w1_loc", torch.rand(n_predictors))
    w1_scale = pyro.param("w1_scale", torch.rand(n_predictors), constraint=constraints.posi
tive)
    w1 = pyro.sample("w1", dist.Normal(w1_loc, w1_scale))

    w2_loc = pyro.param("w2_loc", torch.rand(n_predictors))
    w2_scale = pyro.param("w2_scale", torch.rand(n_predictors), constraint=constraints.posi
tive)
    w2 = pyro.sample("w2", dist.Normal(w2_loc, w2_scale))

    b1_loc = pyro.param("b1_loc", torch.rand(1))
    b1_scale = pyro.param("b1_scale", torch.rand(1), constraint=constraints.positive)
    b1 = pyro.sample("b1", dist.Normal(b1_loc, b1_scale))

    b2_loc = pyro.param("b2_loc", torch.rand(1))
    b2_scale = pyro.param("b2_scale", torch.rand(1), constraint=constraints.positive)
    b2 = pyro.sample("b2", dist.Normal(b2_loc, b2_scale))
```

Finally we run SVI inference

```
# perform inference
log_reg_svi = SVI(model=log_reg_model, guide=log_reg_guide,
              optim=optim.ClippedAdam({'lr' : 0.0002}),
              loss=Trace_ELBO())

losses = []
for step in range(10000):
    loss = log_reg_svi.step(x_train, y_train)/len(x_train)
    losses.append(loss)
    if step % 1000 == 0:
        print(f"Step {step} : loss = {loss}")

figsize=(10,4)
fig, ax = plt.subplots(figsize=figsize)
ax.plot(losses)
ax.set_title("ELBO loss");
```

```
Step 0 : loss = 1.0686675025862509
Step 1000 : loss = 0.9453884888979046
Step 2000 : loss = 0.8059751271655742
Step 3000 : loss = 0.602962341752927
Step 4000 : loss = 0.7665670263612
Step 5000 : loss = 0.6072856730448197
Step 6000 : loss = 0.5007131704441923
Step 7000 : loss = 0.5119495044464452
Step 8000 : loss = 0.5264486852661803
Step 9000 : loss = 0.46723967080461315
```



We can now extract the inferred parameters.

```
print("Inferred params:", list(pyro.get_param_store().keys()), end="\n\n")

inferred_w1 = pyro.get_param_store()["w1_loc"]
inferred_w2 = pyro.get_param_store()["w2_loc"]

inferred_b1 = pyro.get_param_store()["b1_loc"]
inferred_b2 = pyro.get_param_store()["b2_loc"]
```

```
Inferred params: ['w1_loc', 'w1_scale', 'w2_loc', 'w2_scale', 'b1_loc', 'b1_scale', 'b2_loc
', 'b2_scale']
```

For each predicition we predict the class with higher probability.

```
def predict_class(x):
    l0 = torch.zeros(len(x), dtype= float)
    l1 = (inferred_w1*x).sum(dim=1) + inferred_b1
    l2 = (inferred_w2*x).sum(dim=1) + inferred_b2
    softmax = torch.nn.Softmax(dim=1)
    v = softmax(torch.t(torch.stack([l0,l1,l2])))
    return(torch.argmax(v, dim=1))
```

Finally we can compute the overall test accuracy and class-wise accuracy for the three different flower categories.

```
correct_predictions = (predict_class(x_test) == y_test).sum().item()

print(f"test accuracy = {correct_predictions/len(x_test)*100:.2f}%")
```

```
test accuracy = 90.00%
```

```
correct_predictions_0 = ((predict_class(x_test) == y_test) &  (predict_class(x_test) ==
0)).sum().item()
print(f"class wise accuracy setosa = {correct_predictions_0/((y_test == 0).sum().item())*10
0:.2f}%")

correct_predictions_1 = ((predict_class(x_test) == y_test) &  (predict_class(x_test) ==
1)).sum().item()
print(f"class wise accuracy versicolor = {correct_predictions_1/((y_test == 1).sum().item
())*100:.2f}%")

correct_predictions_2 = ((predict_class(x_test) == y_test) &  (predict_class(x_test) ==
2)).sum().item()
print(f"class wise accuracy virginica = {correct_predictions_2/((y_test == 2).sum().item())
*100:.2f}%")
```

```
class wise accuracy setosa = 100.00%
class wise accuracy versicolor = 84.62%
class wise accuracy virginica = 83.33%
```

Solution 2

I import the dataset and I perform a random split of training and test sets

```
from sklearn import datasets
iris = datasets.load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    random_state=0,test_size=0.2)
X_train=torch.tensor(X_train)
y_train=torch.tensor(y_train)

y_test=torch.tensor(y_test)
X_test=torch.tensor(X_test)
```

Model:

$$
\begin{aligned}
w_1 &\sim \mathcal{N}(0,1)\\
w_2 &\sim \mathcal{N}(0,1)\\
b_1 &\sim \text{LogNormal}(0,1)\\
b_2 &\sim \text{LogNormal}(0,1)\\
a1 &= w_1 x + b_1\\
a2 &= w_2 x + b_2\\
\hat{\mu} &= \text{Softmax}(0, a_1, a_2)\\
y &\sim Categorical(\hat{\mu}).
\end{aligned}
$$

I compute the probabilities to be used to sample from the categorical variable using the softmax function on the vector $[0, a_1, a_2]$. The value 0 is because I've chosen the first class as baseline.

As posterior distrbution families I set a Normal distribution over w_1 and w_2 and a Log-Normal on b_1 and b_2, then I can run SVI inference on (X_{train}, y_{train}).

Also in this case I store the losses in a vector to plot them.

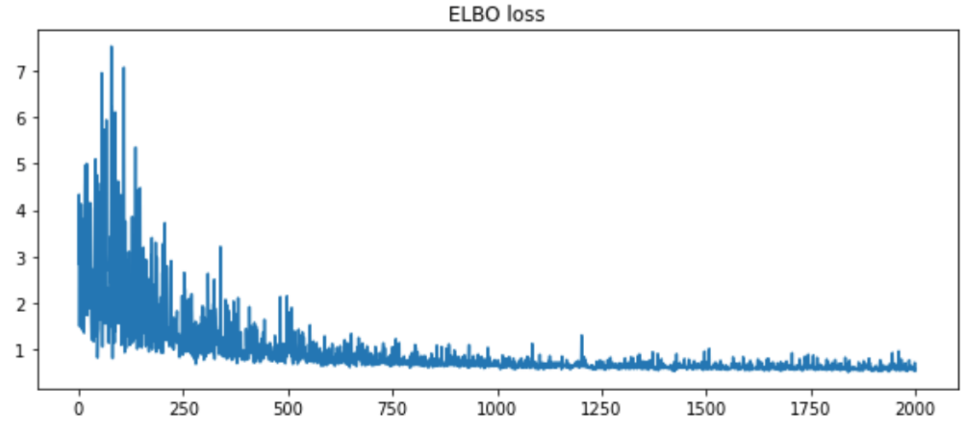
```
pyro.clear_param_store()

def iris_model(x,response):

    n_observations, n_predictors = x.shape

    w1 = pyro.sample("w1", dist.Normal(torch.zeros(n_predictors), torch.ones(n_predictor
s)))
    w2 = pyro.sample("w2", dist.Normal(torch.zeros(n_predictors), torch.ones(n_predictor
s)))

    b1 = pyro.sample("b1", dist.Normal(0.,1.))
    b2 = pyro.sample("b2", dist.Normal(0.,1.))

    a0=torch.zeros(n_observations,dtype=float)
    a1 = (w1*x).sum(dim=1)+b1
    a2 = (w2*x).sum(dim=1)+b2

    a=torch.stack([a0,a1,a2],1)
    sm=torch.nn.Softmax(dim=1)
    y_hat=sm(a)

    with pyro.plate("data", n_observations):
        y = pyro.sample("y", dist.Categorical(probs=y_hat), obs=response)
```

```python
def iris_guide(x, response=None):

        n_observations, n_predictors = x.shape

        w1_loc = pyro.param("w1_loc", torch.rand(n_predictors))
        w1_scale = pyro.param("w1_scale", torch.rand(n_predictors),constraint=constraints.p
ositive)
        w1 = pyro.sample("w1", dist.Normal(w1_loc, w1_scale))

        w2_loc = pyro.param("w2_loc", torch.rand(n_predictors))
        w2_scale = pyro.param("w2_scale", torch.rand(n_predictors),constraint=constraints.p
ositive)
        w2 = pyro.sample("w2", dist.Normal(w2_loc, w2_scale))

        b1_loc = pyro.param("b1_loc", torch.rand(1))
        b1_scale = pyro.param("b1_scale", torch.rand(1),constraint=constraints.positive)
        b1 = pyro.sample("b1", dist.Normal(b1_loc, b1_scale))

        b2_loc = pyro.param("b2_loc", torch.rand(1))
        b2_scale = pyro.param("b2_scale", torch.rand(1),constraint=constraints.positive)
        b2 = pyro.sample("b2", dist.Normal(b2_loc, b2_scale))

iris_svi = SVI(model=iris_model, guide=iris_guide,optim=optim.ClippedAdam({'lr' : 0.01}),
              loss=Trace_ELBO())

losses=[]
for step in range(2001):
    loss = iris_svi.step(X_train,y_train)/len(X_train)
    losses.append(loss)
    if step % 100 == 0:
        print(f"Step {step} : loss = {loss}")
```

```
Step 0 : loss = 2.839971592611755
Step 100 : loss = 2.698142789278957
Step 200 : loss = 2.6535023827110487
Step 300 : loss = 0.9167453812479646
Step 400 : loss = 1.4493958991851383
Step 500 : loss = 0.9687006039211691
Step 600 : loss = 0.7446013227609811
Step 700 : loss = 0.6827562880790949
Step 800 : loss = 0.8439213064780318
Step 900 : loss = 0.8417486190613934
Step 1000 : loss = 0.6409506400287354
Step 1100 : loss = 0.5799221328331167
Step 1200 : loss = 0.6401695473278596
Step 1300 : loss = 0.6363180049951657
Step 1400 : loss = 0.6067477213344212
Step 1500 : loss = 0.670007234167011
Step 1600 : loss = 0.5640109026750136
Step 1700 : loss = 0.6317755090223474
Step 1800 : loss = 0.558813151345158
Step 1900 : loss = 0.6096910369858489
Step 2000 : loss = 0.5703615015290597
```

I can see that losses decrease and stabilize around 0.6. The same behaviour can be seen in the following plot:

```python
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(losses)
ax.set_title("ELBO loss");
```



Now I can extract and print the inferred parameters. Using them to predict class for units in the test set I'm able to estimate the overall test accuracy and the accuracy for different classes:

```python
inferred_w1 = pyro.get_param_store()["w1_loc"]
inferred_w2 = pyro.get_param_store()["w2_loc"]

inferred_b1 = pyro.get_param_store()["b1_loc"]
inferred_b2 = pyro.get_param_store()["b2_loc"]

for i,w in enumerate(inferred_w1):
    print(f"w_{i} = {w.item():.8f}")
for i,w in enumerate(inferred_w2):
    print(f"w_{i} = {w.item():.8f}")
print(f"b1 = {inferred_b1.item():.8f}")
print(f"b2 = {inferred_b2.item():.8f}")
```

```
w_0 = -0.02258334
w_1 = -1.12477040
w_2 = 1.04400527
w_3 = 0.44286636
w_0 = -0.64602524
w_1 = -1.39479220
w_2 = 1.59730661
w_3 = 2.68949914
b1 = 0.21621759
b2 = -1.23895574
```

```python
def predict_class(x):
    a0=torch.zeros(x.shape[0],dtype=torch.float64)
    a1=(inferred_w1 * x).sum(dim=1) + inferred_b1
    a2=(inferred_w2 * x).sum(dim=1) + inferred_b2

    a=torch.stack([a0,a1,a2],1)
    sm=torch.nn.Softmax(dim=1)
    yhat=sm(a)
    return(torch.argmax(yhat,1))


correct_predictions = (predict_class(X_test) == y_test).sum().item()

print(f"test accuracy = {correct_predictions/len(X_test)*100:.2f}%")
```

```
test accuracy = 96.67%
```

```python
idx=torch.stack([(y_test==0),(y_test==1),(y_test==2)],0)
n=torch.sum(idx,1)

for i in range(3):
    correct_predictions=(predict_class(X_test[:][idx[i]])==i).sum().item()
    print("test accuracy for class ",i,f" = {correct_predictions/n[i].item()*100:.2f}%")
```

```
test accuracy for class  0  = 100.00%
test accuracy for class  1  = 92.31%
test accuracy for class  2  = 100.00%
```

I can see that the overall accuracy of 96.67% can be explained with more precision looking at the accuracy for each class. Our model seems to work very well to predict classes 0 and 2 while is a bit less precise predicting class 1 (but still 92.31% is good).

We have also to say that we are computing the accuracy over just a bunch of observetions for each class, we would get a better estimate of the accuracy using a larger dataset or using k-fold cross validation instead of test validation.